Derivative Grammars: A Symbolic Approach to Parsing with Derivatives

IAN HENRIKSEN, The University of Texas at Austin, U.S.A.
GIANFRANCO BILARDI, Universita di Padova, Italy
KESHAV PINGALI, The University of Texas at Austin, U.S.A.

We present a novel approach to context-free grammar parsing that is based on generating a sequence of grammars called *derivative grammars* from a given context-free grammar and input string. The generation of the derivative grammars is described by a few simple inference rules. We present an $O(n^2)$ space and $O(n^3)$ time recognition algorithm, which can be extended to generate parse trees in $O(n^3)$ time and $O(n^2 \log n)$ space. Derivative grammars can be viewed as a *symbolic* approach to implementing the notion of *derivative languages*, which was introduced by Brzozowski.

Might and others have explored an *operational* approach to implementing derivative languages in which the context-free grammar is encoded as a collection of recursive algebraic data types in a functional language like Haskell. Functional language implementation features like knot-tying and lazy evaluation are exploited to ensure that parsing is done correctly and efficiently in spite of complications like left-recursion. In contrast, our symbolic approach using inference rules can be implemented easily in any programming language and we obtain better space bounds for parsing.

Reifying derivative languages by encoding them symbolically as grammars also enables formal connections to be made for the first time between the derivatives approach and classical parsing methods like the Earley and LL/LR parsers. In particular, we show that the sets of Earley items maintained by the Earley parser implicitly encode derivative grammars and we give a procedure for producing derivative grammars from these sets. Conversely, we show that our derivative grammar recognizer can be transformed into the Earley recognizer by optimizing some of its bookkeeping. These results suggest that derivative grammars may provide a new foundation for context-free grammar recognition and parsing.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: context-free grammars, parsing with derivatives, Earley's algorithm

ACM Reference Format:
Ian Henriksen, Gianfranco Bilardi, and Keshav Pingali. 2019. Derivative Grammars: A Symbolic Approach to Parsing with Derivatives. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 127 (October 2019), 28 pages. https://doi.org/10.1145/3360553

Authors' addresses: Ian Henriksen, Oden Institute of Computational Engineering and Science, The University of Texas at Austin, 201 E. 24th Street, POB 4.102, Austin, Texas, 78712, U.S.A., ian@oden.utexas.edu; Gianfranco Bilardi, Dipartimento Ingegneria dell'Informazione, Universita di Padova, Via Gradenigo 6/b, Padova, 35131, Italy, bilardi@unipd.it; Keshav Pingali, Department of Computer Science, The University of Texas at Austin, 2317 Speedway, Austin, Texas, 78712, U.S.A., pingali@cs.utexas.edu.

2475-1421/2019/10-ART127
https://doi.org/10.1145/3360553

1 INTRODUCTION

There are many classical algorithms for recognizing and parsing context-free languages (CFLs) that are specified using context-free grammars (CFGs)[1] [Aho et al. 1986; Hopcroft and Ullman 1979; Sippu and Soisalon-Soininen 1988]. General context-free grammars, which can be ambiguous, can be recognized using Earley's algorithm and the Cocke-Younger-Kasami (CYK) algorithm [Earley 1970; Sippu and Soisalon-Soininen 1988]. These algorithms run in $O(n^3)$ time and take $O(n^2)$ space where n is the length of the input string; for unambiguous grammars, Earley's algorithm runs in $O(n^2)$ time, but the CYK algorithm still takes $O(n^3)$ time. Marpa is a highly optimized implementation of Earley parsing [Kegler 2017]. LL and LR grammars, which are subclasses of general CFGs, can be parsed in $O(n)$ time. ANTLR [Parr and Fisher 2011] supports LL(*) grammars, and Yacc and Bison support LALR(1) grammars [Aho et al. 1986].

Parsing with derivatives (PWD) [Adams et al. 2016; Brachthäuser et al. 2016; Danielsson 2010; Might et al. 2011], which is based on earlier work by Brzozowski on regular grammar recognition [Brzozowski 1964], is a relatively new approach for parsing general CFGs. The key concept is that of a *derivative language*. Informally, the derivative of a string $s = tx$ with respect to terminal symbol t is the string x. If t is not the first symbol of a string, the derivative of that string with respect to t is undefined. Derivatives can be iterated, so the derivative of a string $s = zx$ with respect to string z is x. This idea can be naturally lifted to languages, which are sets of strings.

Definition 1.1. Let T be a set of terminal symbols. Given a language $L \subseteq T^*$ and a string $z \in T^*$, the *derivative language* of L with respect to z, denoted $D_z(L)$, is the set of strings $x \in T^*$ such that zx belongs to L, i.e.:

$$D_z(L) = \{x \in T^* : zx \in L\}. \tag{1}$$

For example, $D_{ab}(\{abcba\}) = \{cba\}$, $D_\epsilon(\{abcba\})$=$\{abcba\}$, and $D_b(\{abcba, bc\}) = \{c\}$.

A straightforward consequence of Definition 1.1 is that a string $w \in L$ iff $\epsilon \in D_w(L)$. This is the basis of the parsing with derivatives approach of Might *et al.* [Might et al. 2011]. Grammars are represented in a functional language like Haskell as elements of a recursive data type with self-referencing pointers in memory. Functional language features such as knot-tying for recursive value definitions and lazy evaluation are exploited in an elegant way to perform parsing correctly and efficiently in spite of complications like left-recursive productions. We refer to this as the *operational approach* to parsing with derivatives because the grammar is encoded in a program.

1.1 Contributions of This paper

$$\begin{array}{ccccccc} \Gamma & \longrightarrow & \Gamma_{t_1} & \longrightarrow & \Gamma_{t_1 t_2} & \longrightarrow \cdots & \Gamma_{t_1 t_2 \ldots t_n} \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ L & \longrightarrow & D_{t_1}(L) & \longrightarrow & D_{t_1 t_2}(L) & \longrightarrow \cdots & D_{t_1 t_2 \ldots t_n}(L) \end{array}$$

Fig. 1. Derivative languages and derivative grammars

This paper makes three main contributions.

(1) (Sections 2 and 3) We present a novel approach to parsing general context-free grammars, based on generating a sequence of grammars that we call *derivative grammars*. The generation of the derivative grammars is described by a small number of inference rules so we call this a *symbolic* approach to parsing with derivatives. Figure 1 shows an overview of the approach.

[1]Recognition is a decision problem: given a string, determine whether the string is in a specified CFL; parsing requires in addition a derivation of that string from the start symbol of the CFG.

Given a grammar Γ and string $w = t_1 t_2 \ldots t_n$, we construct a sequence of derivative grammars $\Gamma_{t_1}, \Gamma_{t_1 t_2}, \ldots, \Gamma_{t_1 \ldots t_n}$ that generate languages $D_{t_1}(L), D_{t_1 t_2}(L), \ldots, D_{t_1 \ldots t_n}(L)$. The string w is in L if and only if ϵ can be derived from the start symbol S_w in $\Gamma_w = \Gamma_{t_1 t_2 \ldots t_n}$.
This approach, which reifies derivative languages by representing them symbolically using grammars rather than encoding them as algebraic data types within a program, is advantageous because it does not require language features like knot-tying and lazy evaluation, and can be easily implemented in any language. A baseline implementation of this approach, called the *pruned derivative grammar* (PrDA) algorithm, permits recognition in $O(n^3)$ time and $O(n^2)$ space, the same complexity as Earley, CYK and PWD.

(2) (Section 4) We show that this recognizer can be extended to a parser that requires $O(n^3)$ time and $O(n^2 \log n)$ space, improving previous results on PWD [Adams et al. 2016].

(3) (Section 5) While PWD has been studied for several years, its connection to classical parsing algorithms like Earley or LL/LR has remained a mystery. Recently Thiemann has introduced an interesting notion called *partial* derivatives to construct nondeterministic pushdown automata (NPDA) from μ-regular expressions [Thiemann 2017]. However there is no direct way to obtain a parsing algorithm starting from an NPDA (one can convert the NPDA back to a CFG and use Earley or CYK but in that case, the NPDA is not needed), so the connection between PWD and classical parsing algorithms has remained unclear.
In this paper, we establish direct and formal connections between derivative grammars and classical parsing algorithms such as the Earley algorithm. In particular, we show that the sets of Earley items maintained by the Earley recognizer implicitly encode derivative grammars and give a procedure for producing derivative grammars from Earley sets. Conversely, we show that the PrDA recognizer can be transformed into the Earley recognizer in three steps, each of which is provably correct. This establishes a formal relationship between the Earley recognizer and the derivatives approach for the first time.
These results do not imply that the Earley recognizer is the same as the PrDA recognizer; rather, they suggest that *derivative grammars may provide a new foundation for CFG recognition algorithms* in the sense that algorithms such as Earley, LL/LR and PWD can be obtained as optimized implementations of the baseline derivative grammar approach introduced in this paper.

2 INFORMAL INTRODUCTION TO DERIVATIVE GRAMMARS

E	$\to$	aEa	E_a	$\to$	Ea	E_{aa}	$\to$	$E_a a$
E	$\to$	bEb	E	$\to$	aEa	E_{aa}	$\to$	ϵ
E	$\to$	ϵ	E	$\to$	bEb	E_a	$\to$	Ea
			E	$\to$	ϵ	E	$\to$	aEa
						E	$\to$	bEb
						E	$\to$	ϵ
(a) Ψ			(b) Ψ_a			(c) Ψ_{aa}		

Fig. 2. An example of parsing with derivatives with grammar Ψ and input string $w = aa$

The grammars in Figure 2(a) and 3(a) are used to introduce the key ideas. Section 2.1 describes informally how derivative grammars are generated. Section 2.2 formalizes this using inference rules. Section 2.3 describes optimizations to this baseline approach.

$E \to E{+}E$ $E \to n$	$E_n \to E_n{+}E$ $E_n \to \epsilon$ $E \to E{+}E$ $E \to n$	$E_{n+} \to E_{n+}{+}E$ $E_{n+} \to E$ $E \to E{+}E$ $E \to n$
(a) Γ	(b) Γ_n	(c) Γ_{n+}
$E_{n+n} \to E_{n+n} + E$ $E_{n+n} \to E_n$ $E_n \to E_n{+}E$ $E_n \to \epsilon$ $E \to E{+}E$ $E \to n$	$E_{n+n+} \to E_{n+n+}{+}E$ $E_{n+n+} \to E$ $E_{n+n+} \to E_{n+}$ $E_{n+} \to E_{n+}{+}E$ $E_{n+} \to E$ $E \to E{+}E$ $E \to n$	$E_{n+n+n} \to E_{n+n+n}{+}E$ $E_{n+n+n} \to E_n$ $E_{n+n+n} \to E_{n+n}$ $E_{n+n} \to E_{n+n}{+}E$ $E_{n+n} \to E_n$ $E_n \to E_n{+}E$ $E_n \to \epsilon$ $E \to E{+}E$ $E \to n$
(d) Γ_{n+n}	(e) Γ_{n+n+}	(f) Γ_{n+n+n}

Fig. 3. An example of parsing with derivatives with grammar Γ and input string $w = t_1t_2t_3t_4t_5 = n{+}n{+}n$

2.1 Examples of the Derivative Grammar Approach

Palindrome grammar: Figure 2(a) shows an unambiguous grammar that is used to teach context-free grammars. It generates even length palindromes over the alphabet $\{a, b\}$. The string aa is in the language generated by this grammar. The successive derivative grammars in Figure 2 show how to recognize this string using the derivative grammar approach.

Call the language of palindromes M. From Definition 1.1, the only strings in M that generate strings in M_a are those that begin with a. These strings must have a derivation in Ψ that starts with the production $(E \to aEa)$. Therefore, the strings in Ψ_a must be generated from the string Ea using the productions for E. The grammar Ψ_a in Figure 2(b) formalizes this intuition by introducing the start symbol E_a with the production $(E_a \to Ea)$, together with the productions for E in the original grammar. Note that E_a is not nullable, which shows that a is not in M, as expected.

The second derivative grammar M_{aa} can be understood in a similar way. A string x in the derivative language M_{aa} must be the derivative of a string $w = ax$ in M_a. This string must have a derivation that starts with the production $(E_a \to Ea)$. There are two cases. One way a can be generated is from the nonterminal E, so we include the production $(E_{aa} \to E_a a)$ in Ψ_{aa}. The other possibility is for E to be rewritten to ϵ using the production $(E \to \epsilon)$, so we include the production $(E_{aa} \to \epsilon)$ in Ψ_{aa}. Adding all the productions in Ψ_a gives us the grammar in Figure 2. Note that E_{aa} is nullable, showing that aa is in M.

Ambiguous grammar. Figure 3 shows an ambiguous grammar Γ that generates the language $L = \{n, n{+}n, n{+}n{+}n, \ldots\}$. We use this grammar as the running example in this paper.

The derivative language $D_n(L)$ is the set of strings $\{\epsilon, {+}n, {+}n{+}n, \ldots\}$, and it is generated by the grammar Γ_n in Figure 3(b). The productions of Γ_n are derived from the productions of Γ as follows.

- $(E \to n)$: This production asserts that $n \in L$. From Definition 1.1, it follows that $\epsilon \in D_n(L)$; therefore, Γ_n contains the production $(E_n \to \epsilon)$. We refer to this production as a *D-child* (derivative child) of the production $(E \to n)$, which, in turn, will be referred to as a *D-parent* (derivative parent) of the production $(E_n \to \epsilon)$.

$$\text{(O)}\ \frac{p \in P(\Gamma)}{p \in P(\Gamma_t)} \qquad \text{(N)}\ \frac{(A \rightarrow \alpha B \beta) \in P(\Gamma) \qquad nullable(\alpha, \Gamma)}{(A_t \rightarrow B_t \beta) \in P(\Gamma_t)}$$

$$\text{(T)}\ \frac{(A \rightarrow \alpha t \beta) \in P(\Gamma) \qquad nullable(\alpha, \Gamma)}{(A_t \rightarrow \beta) \in P(\Gamma_t)}$$

Fig. 4. Inference rules for generating productions of grammar Γ_t from grammar $\Gamma = <N, T, P, S>$. Here α and β represent a possibly empty string that may contain both terminals and nonterminals.

- $(E \rightarrow E+E)$: This production asserts that if strings $x_1, x_2 \in L$, then $x_1+x_2 \in L$. Since every string in L begins with n, the derivatives $D_n(x_1)$ and $D_n(x_1+x_2)$ exist, and by Definition 1.1, they must belong to $D_n(L)$. Therefore, $D_n(x_1)+x_2$, which is equal to $D_n(x_1+x_2)$, must belong to $D_n(L)$. The production $(E_n \rightarrow E_n+E)$ in Γ_n ensures this.

The last two productions in Γ_n are the productions in Γ; these are required because in general, some D-children of productions in Γ will contain nonterminals from Γ (the production $(E_n \rightarrow E_n+E)$ in Γ_n is an example). In a derivative grammar, nonterminals that are not in the original grammar Γ are called *derivative nonterminals*. Note that, following a notational convention similar to the one used for grammars, if A is a nonterminal in the original grammar, then the nonterminal A_u (for some string u) derives the set of strings $\{v | A \Rightarrow^* uv\}$.

When a grammar has nullable nonterminals, it is possible for a production to have several D-children in a derivative grammar. This can be seen in Γ_{n+}. The production $(E_n \rightarrow E_n+E)$ in Γ_n has two D-children in Γ_{n+}, namely the productions $(E_{n+} \rightarrow E_{n+}+E)$ and $(E_{n+} \rightarrow E)$. To understand why the second production is needed, note that E_n is nullable, so adding the production $(E_n \rightarrow +E)$ to Γ_n does not change the language generated by this grammar. Therefore, L_{n+} must contain any string that can be derived from E, which is ensured by the production $(E_{n+} \rightarrow E)$ in Γ_{n+}. The other grammars in Figure 3 can be understood intuitively in the same way. In Γ_{n+n+n}, E_{n+n+n} is nullable, so we conclude $n+n+n \in L$.

2.2 Inference Rules for Derivatives

The intuitive description of derivative grammars given in Section 2.1 is formalized by the inference rules shown in Figure 4 for generating a derivative grammar Γ_t from the grammar Γ and a terminal symbol t. In Section 3, we prove the correctness of an equivalent version of these inference rules, but we consider them here first to build intuition.

- Rule (O) (for Original) asserts that every production in the original grammar Γ is in Γ_t.
- Rule (N) (for Nonterminal) creates productions with derivatives of nonterminals on the right-hand side. This rule generates the production $(E_n \rightarrow E_n+E)$ in Γ_n from the production $(E \rightarrow E+E)$ in Γ. In this case, α is ϵ. Note that only the first symbol on the right-hand side is a derivative nonterminal; all other symbols come from Γ.
- Rule (T) (for Terminal) shows how Γ_t productions are generated by productions of the form $(A \rightarrow \alpha t \beta)$ in which α is nullable and t is the terminal with respect to which derivatives are being taken. This rule generates the production $(E_{n+} \rightarrow E)$ in Γ_{n+} from the production $(E_n \rightarrow E_n+E)$ in Γ_n; in this case, α is E_n.

Just as many grammars can be used to generate a given language, in principle, it is possible to provide various derivative grammars for a single derivative language. The procedure for generating derivative grammars described here produces a specific valid sequence of derivative grammars with various properties, however that sequence of grammars is not unique.

Higher-order derivatives. In principle, higher-order derivatives can be computed by starting with Γ and iteratively applying the rules in Figure 4 to produce a sequence of derivative grammars. This naive approach leads to an exponential time algorithm since it generates many unnecessary productions. To produce Γ_{n+} from Γ_n, rule (O) would add all the productions in Γ_n to the productions in Γ_{n+} even though the nonterminal E_n and its productions are not reachable from E_{n+}.

Section 3 shows how the exponential blow up can be avoided using *pruned derivative grammars*. These grammars can be generated using a single inference rule, which is tailored to systematically avoid most unreachable productions. This inference rule is found in Equation 5 and later expanded in Figure 11a (using the shortened notation of Section 5.2). Using this rule to generate a sequence of pruned derivative grammars (as in Figure 3) leads to our baseline recognition algorithm, referred to as the *pruned derivative algorithm* (PrDA), which is time and space efficient.

2.3 Optimizations

The baseline pruned derivative grammar algorithm can be further optimized in several ways.

Exploiting the structure of derivative productions. A careful examination of the grammars in Figure 3 shows that productions for derivative grammars have a special structure: if a derivative nonterminal occurs in the right-hand side of a production, it must be the first symbol on the right-hand side. Furthermore, such productions are of the form $(A_{vu} \rightarrow B_u \beta)$ where vu and u are suffixes of the portion of the input string that has been read up to that point (a nonterminal A from the original grammar is assumed to be A_ϵ). This is proved in Lemma 3.5. This structure is used to optimize several aspects of the implementation such as the nullability computation in Section A.5.

Memoization. Figure 3 shows that a given derivative nonterminal and its associated productions may occur in many derivative grammars. For example, E_n and its productions occur in grammars Γ_n, Γ_{n+n}. Lemma 3.5 in Section 3 shows that a given nonterminal in a sequence of derivative grammars will have the same productions regardless of the grammar in which it appears. Since the length of the input string can be unbounded while the number of terminal symbols is fixed for a given grammar, these kinds of repetitions will be frequent when parsing long input strings, and will add to both the time and space overheads of parsing in practice.

One way to avoid these repetitions is *memoization* [Adams et al. 2016; Might et al. 2011]. The idea is to use a pool of nonterminal symbols common to all derivative grammars. For each nonterminal, the single-symbol derivatives of that nonterminal that have already been computed are memoized and reused when possible. For example, E_n will map the terminal symbol + to E_{n+}. If E_n already has an entry for a derivative with respect to +, the productions for E_{n+} can be reused every time E_{n+} appears in any derivative grammar. In the example of Figure 3, this optimization eliminates the generation of all productions other than the productions for the start symbol of each derivative grammar. PWD implementations generally use hash tables for memoization [Adams et al. 2016; Might et al. 2011], but case-specific optimizations for memoization were studied in [Adams et al. 2016].

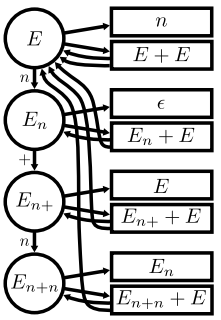
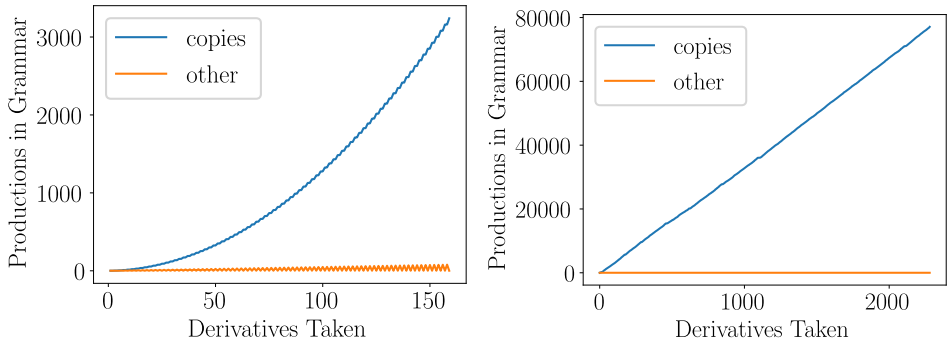


Fig. 5. Memoized version of Γ_{n+n} from example in Figure 3. Arrows between nonterminals labeled by terminal symbols represent taking a derivative with respect to the given symbol.

Figure 5 shows this data structure for the memoized version of Γ_{n+n} from Figure 3. Each nonterminal references its productions which, in turn, reference the nonterminals they contain. A new derivative grammar can be computed recursively, starting with the start symbol of the previous grammar, computing the corresponding derivative productions, then following the same process to generate the productions for any derivative nonterminal encountered that is not already present

in the table of nonterminals. When checking whether a nonterminal A_{ut_i} exists in the table it is sufficient to check if A_u has an entry for a derivative with respect to t_i.



(a) Example in Figure 3 (b) Python grammar

Fig. 6. Growth in copied productions

Eliminating copied productions. The grammars in Figure 3 show a different source of inefficiency that arises from *copied productions*. Any production $(A_{vu} \rightarrow B_u \beta)$ in a given grammar generates a D-child production $(A_{vut} \rightarrow B_{ut} \beta)$ in the succeeding grammar. We call this D-child a *copied production*. All productions with a derivative nonterminal leading the right hand side are copied productions.

In Figure 3, there are chains of copied productions like $(E \rightarrow E{+}E)$, $(E_n \rightarrow E_n{+}E)$, $(E_{n+} \rightarrow E_{n+}{+}E) \ldots$. Intuitively, copied productions are just bookkeeping until the leading derivative nonterminal on the right-hand side becomes nullable, at which point a D-child production that is not a copied production is generated (in the example chain, this happens in grammars Γ_n, Γ_{n+n} and Γ_{n+n+n}).

Although they seem innocuous, copied productions make up the vast majority of the productions generated when parsing long input strings. Figures 6a and 6b show this growth for the grammar in Figure 3 and for a Python grammar respectively (Section 6 has more details about the Python grammar and the inputs). Eliminating copied productions is important for the practical efficiency of the parser. One way to do this is to permit the recognizer to access productions in *any* previous derivative grammar and not just the current grammar at each step of recognition. In Section 5, we show how this idea can be implemented by transforming the pruned derivative algorithm inference rules. Together with two other transformations, this results in the Earley recognizer.

3 DERIVATIVE GRAMMARS: FORMALIZATION

Section 3.1 formalizes the notion of the first derivative of a grammar, using a single inference rule 2, instead of the three rules presented in Section 2, since this simplifies the proofs. Section 3.2 shows that, although higher-order derivatives of grammars can be obtained by simply iterating the first-order derivative construction, this leads to many useless productions. Therefore, Section 3.3 introduces the notion of *pruned* derivative grammars and an inference rule for their construction that avoids these useless productions. Section 3.4 provides complexity results and proves properties of derivative grammars that can be exploited to optimize the implementation, such as in the nullability computation in Section A.5.

3.1 First Derivative

Consider a language L generated by a grammar Γ and its derivative L_t with respect to some terminal t. We show how to construct a grammar Γ_t that generates L_t.

Definition 3.1. **(Derivative grammar)** Given a grammar $\Gamma = <N, T, P, S>$ and a terminal $t \in T$, the *derivative grammar* Γ_t is defined as $\Gamma_t = (N \cup N_t, T, P \cup P_t, S_t)$, where $N_t = \{A_t : A \in N\}$ and P_t is the set of the productions given by the following inference rule, where $\alpha, \beta \in (N \cup T)^*$, and $X \in (N \cup \{t\})$:

$$\frac{(A \rightarrow \alpha X \beta) \in P \quad nullable(\alpha, \Gamma)}{(A_t \rightarrow X_t \beta) \in P_t}. \tag{2}$$

If $X = B \in N$, let $X_t = B_t \in N_t$, so the inferred production becomes $(A_t \rightarrow B_t \beta)$; in this case, Equation 2 is the *N-inference* rule. If $X = t$, let $X_t = \epsilon$, so the inferred productions becomes $(A_t \rightarrow \beta)$; in this case, Equation 2 is the *T-inference* rule. The production $\pi_1 = (A \rightarrow \alpha X \beta)$ in P is called a *D-parent* of the (inferred) production $\pi_2 = (A_t \rightarrow X_t \beta)$ in P_t; conversely, π_2 is called a *D-child* of π_1.

Note that Definition 3.1 combines inference rules (N) and (T) of Figure 4 within rule 2, while it subsumes rule (O) by explicitly making P a subset of the productions of Γ_t. The next result shows that the derivative grammar is in fact a grammar for the derivative language (proof in Section A.1).

Theorem 3.2. *The language generated by Γ_t is the derivative with respect to t of the language generated by Γ, i.e.:*

$$L(\Gamma_t) = D_t(L(\Gamma)). \tag{3}$$

3.2 Higher-Order Derivatives

We now extend the notion of derivative from a single terminal to a string of terminals, resulting from the iterated application of the inference rule in Definition 3.1.

Definition 3.3. **(Iterated derivative grammars)** Given a grammar $\Gamma = <N, T, P, S>$ and a terminal string $w = t_1 t_2 \dots t_n \in T^n$, the *iterated derivative grammar* $\Gamma'_{t_1 \dots t_j}$ is inductively defined by letting $\Gamma'_\epsilon = \Gamma$ and, for $j = 1, \dots, n$, $\Gamma'_{t_1 \dots t_j} = (\Gamma'_{t_1 \dots t_{j-1}})_{t_j}$, where the subscript t_j denotes differentiation in the sense of Definition 3.1.

A consequence of Definition 1.1 is that $D_{t_1 \dots t_j}(L) = D_{t_j}(D_{t_1 \dots t_{j-1}}(L))$. Then, a straightforward inductive argument based on the length of w and using Theorem 3.2 establishes that the language generated by Γ'_w is the derivative with respect to w of the language generated by Γ, i.e.:

$$L(\Gamma'_w) = D_w(L(\Gamma)). \tag{4}$$

One serious drawback of the grammars obtained by the iterated derivative method is that taking the derivative with respect to one terminal can lead to a constant factor increase in the number of productions; therefore, this number can grow exponentially with n. In fact, with reference to inference rule 2, each production in P can generate, in the worst case, a number of productions in P_t equal to one plus the length of its right hand side. (Specifically, such worst case will occur if all symbols on the right hand side are nullable nonterminals.)

Fortunately, most of the productions in grammar Γ'_w are useless, since they are not reachable from the start symbol, S_w. To gain some intuition on unreachable productions, consider $\Gamma'_{t_1 t_2} = (\Gamma'_{t_1})_{t_2}$. Starting from $\Gamma'_{t_1} = (N \cup N_{t_1}, T, P \cup P_{t_1}, S_{t_1})$, we can write $\Gamma'_{t_1 t_2} = (N \cup N_{t_1} \cup N_{t_1 t_2}, T, P \cup P_{t_1} \cup P_{t_1 t_2}, S_{t_1 t_2})$, where $P_{t_1 t_2}$ is the set of those productions resulting from the application of inference rule 2 to the productions in $P \cup P_{t_1}$. We will now argue that the set of productions P_{t_1} can be removed from the expression for $\Gamma'_{t_1 t_2}$ without affecting the language generated by the grammar. If $t_2 = t_1$, then $P_{t_1} \subseteq P_{t_1 t_2}$, hence its inclusion is redundant. Else ($t_2 \neq t_1$), we will show that productions in P_{t_1} are

unreachable. In fact, considering the structure of inference rule 2, productions in P_{t_1} will have one of the following forms: $(A_{t_1} \rightarrow B_{t_1}\beta)$; or $(A_{t_1} \rightarrow \beta)$; where $\beta \in (N \cup T)^*$. On the other hand, productions in $P_{t_1t_2}$ will have one of the following forms: $(A_{t_1t_2} \rightarrow B_{t_1t_2}\beta)$; or $(A_{t_1t_2} \rightarrow B_{t_2}\beta)$; or $(A_{t_1t_2} \rightarrow \beta)$; where $\beta \in (N \cup T)^*$. Therefore, a derivation starting with $S_{t_1t_2}$ will never include a symbol of the form B_{t_1}, hence never use a production in P_{t_1}. In conclusion, $L_{t_1t_2}$ can be generated by the "pruned" grammar $\Gamma_{t_1t_2} = (N \cup N_{t_1t_2}, T, P \cup P_{t_1t_2}, S_{t_1t_2})$. The considerations illustrated by this example lead to the definition of pruned derivative grammar developed in the next subsection.

3.3 Pruned Derivatives

In this subsection, we provide a definition of the pruned derivative Γ_w of a grammar Γ with respect to a string $w = t_1t_2 \ldots t_n$ and then we show that $L(\Gamma_w) = D_w(L(\Gamma))$. The definition builds a sequence of $n+1$ grammars, corresponding to increasing prefixes of w, from ϵ to w itself. Each grammar $\Gamma_{t_1t_2\ldots t_i}$ in the sequence can be viewed as an augmentation of the given grammar Γ with new productions obtained by applying inference rule 2 to the productions of the previous grammar $\Gamma_{t_1t_2\ldots t_{i-1}}$ in the sequence. Observe that, in addition to the productions obtained by the inference rule, (by Definition 3.1) the first derivative $(\Gamma_{t_1t_2\ldots t_{i-1}})_{t_i}$ would also include all the productions of $\Gamma_{t_1t_2\ldots t_{i-1}}$. In contrast, $\Gamma_{t_1t_2\ldots t_i}$ will not include the productions that are in $\Gamma_{t_1t_2\ldots t_{i-1}}$ but not in Γ, which are "pruned" away. The impact of pruning on the size of the grammars is significant and builds up across the iterations.

In the grammar Γ_z, associated with the prefix $z = t_1t_2 \ldots t_i$ of w, the new nonterminals are of the form A_v, with $v = t_h \ldots t_i$ a suffix of z. The productions that are added to those of Γ ensure (as will be proven) that the strings that can be derived from A_v form exactly the derivative with respect to v of the set of strings that can be derived from A. In particular, S_w generates $D_w(L)$.

This pruned approach does not eliminate all unreachable nonterminals, however it does remove all productions containing derivative symbols with subscripts that are not suffixes of the current prefix z since they are guaranteed not to be reachable. See Lemma 3.5 for a list of properties the remaining productions must satisfy. Though some unreachable productions may remain, this reduction in the number of productions suffices for the complexity bounds proved in Theorem 3.7.

Definition 3.4. **(Pruned derivative grammars)** Given a grammar $\Gamma = <N, T, P, S>$ and a string $w = t_1t_2 \ldots t_n$, for each prefix $z \in \{\epsilon, t_1, \ldots, t_1t_2 \ldots t_n\}$, we define a grammar

$$\Gamma_z = <N \cup N_z, T, P \cup P_z, S_z>,$$

where

- by convention, for every $A \in N$, $A_\epsilon = A$; $N_\epsilon = N$, $P_\epsilon = P$, and $S_\epsilon = S$, so that $\Gamma_\epsilon = \Gamma$;
- for $i = 1, 2, \ldots n$ and letting $z = t_1 \ldots t_i$, the sets N_z contain nonterminals of the form A_v with v a nonempty suffix of z (that is, $v \in \{t_i, t_{i-1}t_i, \ldots, t_1 \ldots t_i\}$);
- for $i = 1, 2, \ldots n$ and letting $z = t_1 \ldots t_{i-1}t_i = z't_i$, the set of productions P_z is obtained from $P \cup P_{z'}$ according to the following inference rule, where $\alpha, \beta \in (N \cup N_{z'} \cup T)^*$, and $X \in (N \cup N_{z'} \cup \{t_i\})$:

$$\frac{(A_v \rightarrow \alpha X \beta) \in P \cup P_{z'} \quad nullable(\alpha, \Gamma_{z'})}{(A_{vt_i} \rightarrow X_{t_i}\beta) \in P_z}. \tag{5}$$

If $X = B_u \in (N \cup N_{z'})$, let $X_{t_i} = B_{ut_i} \in N_z$, so the inferred production becomes $(A_{vt_i} \rightarrow B_{ut_i}\beta)$ and Equation 5 is called an *N-inference* rule. If $X = t_i$, let $X_{t_i} = \epsilon$, so the inferred production becomes $(A_{vt_i} \rightarrow \beta)$ and Equation 5 is called a *T-inference* rule.
The production $\pi_1 = (A_v \rightarrow \alpha X \beta)$ in $P \cup P_{z'}$ is called a *D-parent* of the (inferred) production $\pi_2 = (A_{vt_i} \rightarrow X_{t_i}\beta)$ in P_z; conversely, π_2 is called a *D-child* of π_1.

We observe that if the subscript v of the left-hand side nonterminal A_v of a D-parent production is a suffix of $z' = t_1 \dots t_{i-1}$, then the subscript vt_i of the left-hand side nonterminal A_{vt_i} of the D-child production is a nonempty suffix of $z = t_1 \dots t_i$, as required by the condition for A_{vt_i} to be in N_z of Definition 3.4.

3.4 Properties of Pruned Derivative Grammars

Productions and derivations within the pruned derivative grammars exhibit various properties that play a useful role in both connecting these grammars with derivative languages and establishing the parsing algorithm based on these grammars.

Lemma 3.5. *The pruned derivative grammars of Definition 3.4 satisfy the following properties.*

(1) If $W \rightarrow Y\xi$ is a production in any derivative grammar, then $\xi \in (N \cup T)^*$, that is, all symbols in the right-hand side of the production, except possibly for the first one (Y), are symbols of the original grammar Γ.
(2) Let $W \rightarrow Y\xi$ be a production in P_z where Y is a nonterminal. If $W = C_r$ and $Y = D_s$ (with $C, D \in N$), then s is a suffix of r (including the cases $s = \epsilon$ and $s = r$), and both r and s are suffixes of z.
(3) If $C_r \Rightarrow^*_{\Gamma_z} Y\xi$ is a derivation within grammar Γ_z with $C_r \in N \cup N_z$, (r a suffix of z), then $\xi \in (N \cup T)^*$ and, if $Y = D_s$ is a nonterminal, then s is a suffix of r (hence of z).
(4) Let $z = pr$ and $C \in N$. The set of productions for C_r in grammar Γ_z equals the set of productions for C_r in grammar Γ_r (i.e., this set of productions is independent of p).

Theorem 3.6. *With reference to the notation introduced in Definition 3.4, for each prefix z of w, the pruned derivative grammar Γ_z generates the derivative with respect to z of the language generated by Γ, i.e.:*

$$L(\Gamma_z) = D_z(L(\Gamma)). \tag{6}$$

The proofs of Lemma 3.5 and Theorem 3.6 are in Sections A.2 and A.3 of the appendix.

Theorem 3.7. *Given Γ and $w = t_1 t_2 \dots t_n$, grammar Γ_w can be stored in space $O(n^2)$ and constructed in time $O(n^3)$.*

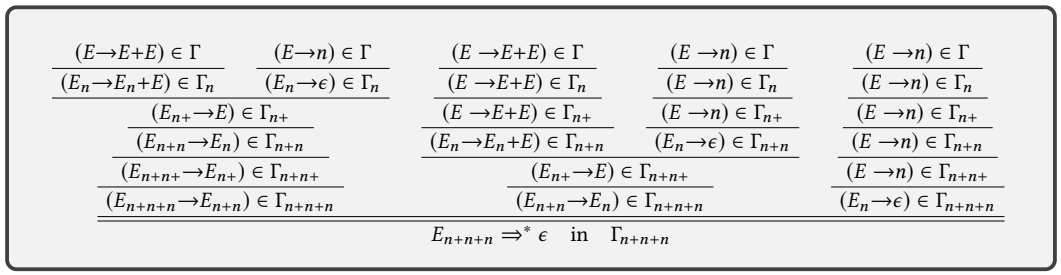
The proof of Theorem 3.7 is in Section A.4 of the appendix.

4 CONSTRUCTING PARSE TREES USING BACKWARD INFERENCE

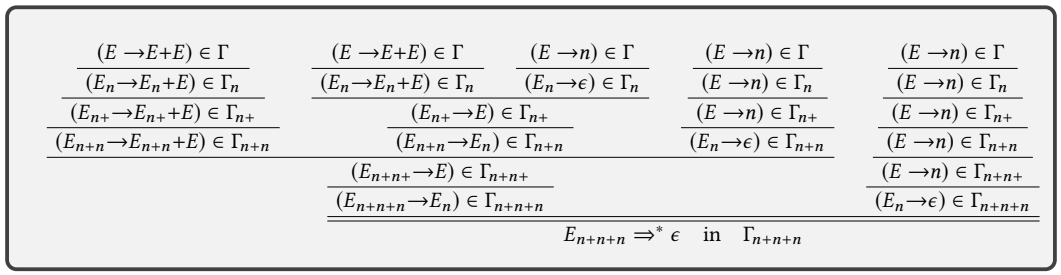
The recognizer in Section 3 can be extended to produce a parse tree for a string of length n in time $O(n^3)$ and space $O(n^2 \log n)$, improving the $O(n^3)$ time and space bounds of Adams *et al.* [Adams et al. 2016]. The basic idea is the following.

The generation of the sequence of derivative grammars for a given grammar Γ and input string $w \in L(\Gamma)$ uses *forward* inference since the productions in each derivative grammar $\Gamma_{t_1 \dots t_{i+1}}$ are generated by applying inference rules to the productions in the previous grammar $\Gamma_{t_1 \dots t_i}$.

The generation of parse trees from the sequence of derivative grammars is *backward* inference: we first construct a proof of nullability of the start symbol S_w of the last grammar Γ_w using the productions of Γ_w, and then repeatedly replace productions in each derivative grammar $\Gamma_{t_1 \dots t_{i+1}}$ with the productions in $\Gamma_{t_1 \dots t_i}$ that generated it during the forward inference. This backward inference process stops when all the productions are in Γ, producing a derivation of w in Γ. We focus on leftmost derivations, from which it is straightforward to obtain a parse tree [Aho et al. 1986]; other derivation orders can easily be obtained by modifications of this algorithm.



(a) First inference tree



(b) Second inference tree

Fig. 7. Two inference trees for string n+n+n.

4.1 Example of Backward Inference for Parsing

Figure 7 shows two inference trees that produce leftmost derivations $E \Rightarrow^* n$+n+n from the sequence of derivative grammars of Figure 3 (there are two leftmost derivations because the grammar Γ is ambiguous). Both inference trees start with a leftmost derivation of $E_{n+n+n} \Rightarrow^* \epsilon$ in Γ_{n+n+n}. In particular, Figure 7(a) starts with the leftmost derivation $E_{n+n+n} \Rightarrow E_{n+n} \Rightarrow E_n \Rightarrow \epsilon$. The sequence of productions used in this derivation is $(E_{n+n+n} \to E_{n+n})$, $(E_{n+n} \to E_n)$, $(E_n \to \epsilon)$, shown in the assumptions of the last inference in Figure 7(a). Intuitively, this sequence of productions can be viewed as an encoding of the proof that $E_{n+n+n} \Rightarrow^* \epsilon$ in grammar Γ_{n+n+n}. In Figure 7(a), this inference is shown as a double line to distinguish it from other inferences in this inference tree, which are instances of inference rule (5) as discussed below.

At the leaves of the tree in Figure 7(a), there are productions from Γ. Using these productions in left-to-right order in a leftmost derivation starting at E, we get the string n+n+n, so the sequence of productions at the leaves of the tree in left to right order can be viewed as an encoding of the proof that $E \Rightarrow^* n$+n+n in grammar Γ.

To obtain this sequence of productions from the sequence of productions used in the proof that $E_{n+n+n} \Rightarrow^* \epsilon$ in grammar Γ_{n+n+n}, we proceed iteratively up from the root, replacing each production in a given level with its D-parent in the grammar at the previous level; this procedure will be extended below to account for the nullability assumption in inference rule (5). In Figure 7(a), the first step of this backward inference generates the productions $(E_{n+n+} \to E_{n+})$, $(E_{n+} \to E)$, $(E \to n)$ (in left to right order) in grammar Γ_{n+n+}. Note that if these productions are used in a leftmost derivation starting at E_{n+n+}, we obtain the string n, which is the suffix of the input string generated by the derivative grammar Γ_{n+n+}, as expected.

This procedure of backward inference using the D-parent relation must be extended to account for the nullability computation in inference rule (5). When the production $(E_{n+} \to E)$ in Γ_{n+n+} is

considered, we see that its D-parent is $(E_n \rightarrow E_n + E)$, but the generation of this production when inference rule (5) was applied required the assumption $nullable(E_n)$. Therefore in the backward inference process, a production must be replaced not just with its D-parent production but its D-parent followed by the productions used in a leftmost derivation $\alpha \Rightarrow^* \epsilon$ in the grammar $\Gamma_{z'}$ in Inference Rule (5). These productions are computed for each nullable nonterminal during the generation of the derivative grammars, as described in Section A.5. In the running example, this generates the sequence $(E_n \rightarrow E_n + E), (E_n \rightarrow \epsilon)$ as expected.

In general, with reference to inference rule (5), the *P-parent relation* (parsing parent relation) associates with a given production $A_{vt_i} \rightarrow X_{t_i}\beta$ in the derivative $\Gamma_{z't_i}$ of a grammar $\Gamma_{z'}$ its D-parent production $A_v \rightarrow \alpha X \beta$ as well as the productions in $\Gamma_{z'}$ used in a derivation $\alpha \Rightarrow^*_{\Gamma_{z'}} \epsilon$ (that is, the productions used in the proof of the precondition $nullable(\alpha, \Gamma_{z'})$ of inference rule (5)). A *P-parent sequence* is the sequence formed by the D-parent followed by the remaining productions of the P-parent relation, in the order in which they are used in a leftmost derivation of $\alpha \Rightarrow^*_{\Gamma_{z'}} \epsilon$.

Given this definition, the general procedure for generating inference trees can be described simply as follows: *starting from the sequence of productions used in a leftmost derivation $S_w \Rightarrow^* \epsilon$ in Γ_w, replace each production in a given level with its P-parent sequence in the grammar at the previous level until all productions are from Γ.*

Turning to the inference tree in Figure 7(b), we see that it uses a different proof of nullability of E_{n+n+n}, namely $E_{n+n+n} \Rightarrow E_n \Rightarrow \epsilon$. In general, there may be many inference trees for a given word if the grammar is ambiguous. Multiple trees may arise because there are multiple proofs of nullability of S_w in Γ_w or because one or more productions at intermediate nodes in the tree have multiple P-parent sequences. For simplicity, we focus below on generating a single inference tree even if the grammar is ambiguous. It is not difficult to extend this procedure to generate a *shared packed parse forest* (SPPF), which is a compact representation of multiple parse trees, similar to AND-OR trees [Scott 2008].

4.2 An Efficient Procedure for Constructing Parse Trees

The approach described informally above performs a breadth-first expansion of the inference tree. While this is easy to understand, a more space-efficient construction is obtained by expanding the inference tree in a depth-first order from right to left, and building the parse tree directly during this traversal. Figure 8 shows the pseudocode for such a parser. It maintains two stacks, a stack of productions called ProdStack and a stack of parse tree fragments called TreeStack. The production stack is pushed whenever a node with multiple predecessors in the inference tree is encountered. This can happen at the root node and at internal nodes that have P-parent sequences with multiple productions, as is the case with the node $(E_{n+} \rightarrow E) \in \Gamma_{n+}$ in Figure 7(a). These predecessors are visited in right to left order. Fragments of the parse tree are constructed whenever a leaf node of the inference tree is visited: if the righthand side of the production has no nonterminals, the tree for that production can be constructed directly, and if the righthand side has k nonterminals, trees for these nonterminals are popped from the TreeStack and the tree for that production is constructed. For example, when the rightmost occurrence of $(E \rightarrow E + E) \in \Gamma$ in Figure 7(a) is encountered, the trees for the two occurrences of E on the righthand side have been constructed during the visits to the two leaves to the right of this leaf, and they are popped from TreeStack. In both cases, the resulting tree is pushed on the TreeStack.

The pseudocode in Figure 8 incorporates one additional optimization. When a production in the original grammar Γ is encountered at an internal node in the inference tree, it is not necessary to generate the chain of nodes from that internal node to the leaf of the inference tree; instead the fragment of the parse tree is generated immediately and the traversal backs out of that node.

```
Given: grammar Γ = <N, T, P, S>, word w = t1 t2 ... tn
       generated by Γ.
Returns: a parse tree for w.

ProdStack = [ ]; // stack of productions
TreeStack = [ ]; // stack of parse trees

Construct sequence of derivative grammars Γ_t1, ..., Γ_t1 t2 .. tn;
Construct a leftmost derivation S_w ⇒* ϵ in Γ_t1,..,tn;
Push productions in derivation on ProdStack in the order used in derivation;

while (! ProdStack.empty()) {
  (A→α) = ProdStack.pop(); // for some production (A→α) with α ∈ (N ∪ T)*
  if (A→α) ∈ Γ // generate parse tree fragment and back out of node
    {if (α is ϵ or α has only terminal symbols)
       Construct tree for (A→α) and push on TreeStack;
     else{
       // α contains nonterminals and their trees are on top of TreeStack
       k = number of nonterminals in α;
       Pop k trees from TreeStack;
       Construct tree for (A→α) using these as subtrees and push on TreeStack;
       }
    }
  else // push P−parent production sequence from previous grammar
    push P−parents of (A→α) on ProdStack in leftmost derivation order;
}
return TreeStack.pop();
```

Fig. 8. Generating a parse tree from a grammar Γ and a word $w \in L(\Gamma)$

For example, when the production $(E \rightarrow n) \in \Gamma_{n+n+}$ in the rightmost branch of the inference tree of Figure 7(a) is encountered, the tree for the production $(E \rightarrow n)$ can be generated right away without pushing and popping the nodes on the chain from the internal node all the way to the leaf.

Theorem 4.1 provides time and space complexity bounds for parsing with derivatives. The corresponding proof in Section A.6 shows that a variant of the parsing procedure described here can run in the given time and space bounds, which improve on the time and space bounds given by Adams *et al.* [Adams et al. 2016].

Theorem 4.1. *Given a grammar Γ, a parse tree for a string $w = t_1 t_2 \ldots t_n \in L(\Gamma)$ can be constructed in time $O(n^3)$ and space $O(n^2 \log n)$.*

4.3 Discussion

The parse tree construction algorithm discussed in this section constructs a leftmost derivation of the input word, but it can be modified easily to construct other derivation orders such as rightmost derivations. The derivation $S_w \Rightarrow^* \epsilon$ and P-parent sequences must be modified to use the desired derivation order, but the rest of the algorithm remains the same.

5 RELATING CLASSICAL PARSING ALGORITHMS TO DERIVATIVE GRAMMARS

To connect classical recognition algorithms like LL, LR, Earley, and CYK to PWD, we observe that, other than the CYK algorithm, these algorithms are *online* in the sense that they process the input string from left to right without backtracking[2]. After a prefix z of the input string has been processed, the state of an online recognizer must encode enough information to decide whether a string zx is in the given language for any string x of terminal symbols. *In other words, the state of such a recognizer must encode the derivative language of L with respect to string z.*

Online recognizers differ in (i) how they internally represent the derivative language, (ii) how they update this representation upon processing a new terminal, and (iii) how they test for $\epsilon \in D_w(L)$.

For example, deterministic pushdown automata (DPDA) such as those for LL and LR recognizers encode the derivative language via the state of their finite control and the content of their stack, and the grammar for L is encoded in the transition function of the automaton. The test for $\epsilon \in D_w(L)$ is implemented in a DPDA either by acceptance by empty stack or acceptance by final state.

The most complicated classical recognizer is the Earley recognizer (Section 5.1). We show how to transform the pruned derivative grammar recognizer (PrDA) into the Earley recognizer in three steps, each of which is provably correct. Conversely, we show that the sets of Earley items in the Earley recognizer implicitly encode derivative grammars and give a procedure for generating derivative grammars from Earley sets (Section 5.2). We also show briefly how LL and LR parsing can be described using the derivatives-based approach (Section 5.3).

Non-negative integers: $\mathbb{N}$
Σ-set: $\mathscr{P}\left(\{A \to \alpha.\beta \mid A \in N \text{ and } \alpha, \beta \in (N \cup T)^*\} \times \mathbb{N}\right)$
State: Σ^+
Acceptance: $<S \to \sigma\bullet, 0> \in \Sigma_{|w|}$

$$\text{INIT}\ \frac{(S \to \alpha) \in P(\Gamma)}{<S \to \bullet\alpha, 0> \in \Sigma_0} \qquad \text{COMPLETION}\ \frac{<B \to \gamma\bullet, h> \in \Sigma_i \qquad <A \to \eta\bullet B\beta, g> \in \Sigma_h}{<A \to \eta B\bullet\beta, g> \in \Sigma_i}$$

$$\text{PREDICTION}\ \frac{<A \to \eta\bullet B\beta, h> \in \Sigma_i \qquad (B \to \gamma) \in P(\Gamma)}{<B \to \bullet\gamma, i> \in \Sigma_i} \qquad \text{SCAN}\ \frac{<A \to \eta\bullet t_i\beta, h> \in \Sigma_{i-1}}{<A \to \eta t_i\bullet\beta, h> \in \Sigma_i}$$

Fig. 9. Earley's algorithm: grammar is $<N, T, P, S>$, input word is w, with $\alpha, \beta, \gamma, \nu \in (N \cup T)^*$

5.1 Inference Rules for the Earley Parser

Given an input word $w = t_1 \ldots t_n$, Earley's algorithm constructs a sequence of *frames* (also known as Σ-sets) labeled $\Sigma_0, \Sigma_1, \ldots, \Sigma_n$, where Σ_i depends only on $t_1 \ldots t_i$. Each frame Σ_i is a set of *Earley entries* of the form $<A \to \eta\bullet\zeta, h>$, where $(A \to \eta\zeta)$ is a production of the given grammar Γ and h is an integer such that $0 \le h \le i$. We call $A \to \eta\bullet\zeta$ the *item*, i the *frame number*, and h the *tag* of entry $<A \to \eta\bullet\zeta, h> \in \Sigma_i$. Tags correspond to frame numbers, and a necessary (although not sufficient) condition for EA to generate such an entry is that $\eta \Rightarrow_\Gamma^* t_{h+1} \ldots t_i$.

Earley's algorithm is described by the inference rules in Figure 9. Figure 10 shows the sequence of frames constructed by this algorithm for the example in Figure 3. Entries in Σ sets are assertions about how the recognizer is attempting to use various substrings of the input, as described below.

[2]CYK can be easily modified to work online but in its usual presentation, it is not an online algorithm.

- An entry of the form $<B \rightarrow \bullet\gamma, i>$ in frame Σ_i indicates that the algorithm is attempting to use the production $(B \rightarrow \gamma)$ as the root production in a derivation of a substring of the input starting at (and including) t_{i+1}.
- An entry $<B \rightarrow \gamma\bullet, h>$ in frame Σ_i indicates that the algorithm has succeeded in using the production $(B \rightarrow \gamma)$ as the root production in a derivation of the substring $t_{h+1} \ldots t_i$.
- When the dot in an item is somewhere in the middle of the righthand side of a production, such as $<A \rightarrow \eta\bullet\zeta, h>$ in Σ_i, it indicates that the algorithm is attempting to use the production $(A \rightarrow \eta\zeta)$ as the root production in a derivation of a substring starting at t_{h+1}, and the substring $t_{h+1} \ldots t_i$ can be generated from η.

Entries in the first frame Σ_0 are inserted by the INIT rule. The PREDICT rule is triggered when the dot in an item is before a nonterminal symbol B, and the algorithm tries to use all the productions for B to parse substrings starting at t_{i+1}. It is useful to think of this as starting an "invocation" of B at this point. In Figure 10, Σ_2 contains an entry $<E \rightarrow E+\bullet E, 0>$, so the PREDICT rule is used to add the entries $<E \rightarrow \bullet n, 2>$ and $<E \rightarrow \bullet E+E, 2>$. The COMPLETION rule performs the lookback into previous Σ sets to see how to continue parsing when a production $B \rightarrow \gamma$ completes. The tag for this entry shows the Σ set at which the invocation of B was made, and the entries in this Σ set that have a dot before a B on the righthand side show how to continue the recognition. In Figure 10, Σ_3 has an entry $<E \rightarrow E+E\bullet, 0>$. The tag shows that this invocation of E was started in Σ_0, so we look back into this Σ set and find the entry $<E \rightarrow \bullet E+E, 0>$. The call to E has completed, so we add the entry $<E \rightarrow E\bullet+E, 0>$ to continue the parse. The SCAN rule simply consumes a terminal symbol from the input and continues the recognition. The string w is accepted if the entire string can be read in and the last Σ set contains an entry $<S \rightarrow \sigma\bullet, 0>$; intuitively, this means that an invocation of S that started in frame 0 completes after consuming the entire input string.

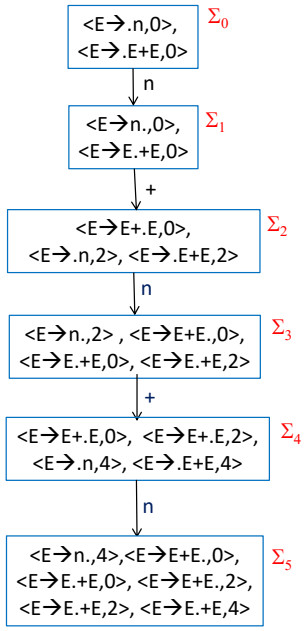


Fig. 10. Σ sets for Figure 3 example

Even at a superficial level, we see a similarity between the PrDA recognizer of Section 3.3 and the Earley recognizer: if the input string has length n, one uses $n + 1$ grammars while the other uses $n + 1$ frames. The next section establishes a deeper connection between frame entries and productions in derivative grammars.

5.2 Transforming the PrDA Recognizer to the Earley Recognizer

There are three transformation steps to get to Earley from PrDA. The result of applying each transformation is captured using inference rules in Figure 11.

1. Unlike PrDA, the Earley recognizer does not require an explicit check for nullability of nonterminals. Section 5.2.1 shows that for a class of grammars called *ϵ-reduced grammars*, nullability of nonterminal A is equivalent to the existence of a production $(A \rightarrow \epsilon)$. The inference rules in Figure 11b generate ϵ-reduced derivative grammars. The SCAN and COPY rules are also simpler for these grammars.

2. Section 5.2.2 introduces the COMPLETION rule, which permits the recognizer to look up productions in any previous derivative grammar, eliminating the need for copied productions and the COPY rule (Figure 11c).
3. The ORIGINAL rule introduces all productions of the original grammar Γ into each derivative grammar. Section 5.2.3 exploits a simple reachability check to reduce the number of these productions introduced into each grammar (Figure 11d).

Section 5.2.4 shows that, with a change in notation, this final recognizer is the Earley recognizer.

5.2.1 Nullability Computation. If a grammar contains productions $(A \rightarrow B\beta)$ and $(B \rightarrow \epsilon)$, we can obviously add the production $(A \rightarrow \beta)$ (Where $\beta \in (N \cup T)^*$) to the grammar without changing the generated language. This process can be applied recursively if β itself starts with a nonterminal that has an ϵ-production. The following lemma, proved in Appendix A.8, describes properties of grammars generated by this construction, which we call *ϵ-reduction*. A grammar G is said to be *ϵ-reduced* if the ϵ-reduction of G is G itself.

Lemma 5.1. *Given grammar $G = <N, T, P, S>$, the ϵ-reduction of G is the grammar $G_\epsilon = <N, T, P_\epsilon, S>$ in which the productions of P_ϵ are defined by the following rules:*

$$\text{ORIGINAL } \frac{p \in P}{p \in P_\epsilon} \qquad \text{EPSILON } \frac{(A \rightarrow B\beta) \in P_\epsilon \qquad (B \rightarrow \epsilon) \in P_\epsilon}{(A \rightarrow \beta) \in P_\epsilon}$$

Then (i) G and G_ϵ generate the same language, (ii) if A is nullable in G, P_ϵ contains the production $(A \rightarrow \epsilon)$, and (iii) if $(A \rightarrow \alpha\beta) \in P$ and nullable(α), then $(A \rightarrow \beta) \in P_\epsilon$.

In Figure 11b, the EPSILON rule has been added to the pruned derivative grammar rules of Figure 11a. The SCAN and COPY rules have been simplified; like in the Earley recognizer, there is no explicit check for nullability since the "heavy lifting" is done by the EPSILON rule. The new sequence of derivative grammars is named Δ_i. Lemma 5.1 and Theorem 5.2 are proved in the appendix in sections A.7 and A.8 respectively. As is already the case in Figure 11a, we use Γ_i rather than $\Gamma_{t_1 \ldots t_i}$ to denote the i-th grammar in the sequence built by the PrDA recognizer.

Theorem 5.2. *Given a grammar Γ and string $w = t_1 t_2 \ldots t_n$, let the sequences of derivative grammars produced by the rules in Figure 11a and Figure 11b be $\Gamma_0, \Gamma_1 \ldots \Gamma_n$ and $\Delta_0, \Delta_1 \ldots \Delta_n$ respectively. Δ_i is the ϵ-reduction of Γ_i and therefore generates the same language.*

5.2.2 Eliminating the COPY Rule. As discussed in Section 2, eliminating copied productions is important for efficiency. The next transformation eliminates the COPY rule by permitting the recognizer to access productions in *any* previous derivative grammar. There are two steps.

First, we prove that the EPSILON rule in Figure 11b can be generalized to the COMPLETION rule as shown in Figure 11c without changing the generated grammars. This is proved in Appendix A.9 by showing that if $(B_u \rightarrow \epsilon) \in P(\Phi_i)$ (the common assumption of the EPSILON and COMPLETION rules) and u is $t_{h+1} \ldots t_i$, $(A_{vu} \rightarrow B_u\beta) \in P(\Delta_i)$ iff $(A_v \rightarrow B\beta) \in P(\Delta_h)$. Therefore, the assumptions of the two rules are equivalent, and the EPSILON rule can be replaced with the COMPLETION rule without changing the generated grammars. Notice that the COMPLETION rule requires the ability to "lookback" to productions in Δ_h when generating productions in Δ_i, where $i \geq h$.

Next, we observe that this makes the COPY rule redundant in the following sense. In the inference rules of Figure 11c, only the COPY rule assumes and generates copied productions. If a string $w = t_1 \ldots t_n$ is in the language $L(\Gamma)$, Φ_n will contain $(S_{t_1 \ldots t_n} \rightarrow \epsilon)$ since the grammar is ϵ-reduced. This is not a copied production. Therefore, if we perform a backward inference starting at this production, similar to the procedure for constructing parse trees described in Section 4 but for the rules in Figure 11c, we will not encounter a copied production. Intuitively, with the COMPLETION

$$\text{ORIGINAL}\ \frac{p \in P(\Gamma)}{p \in P(\Gamma_i)} \qquad \text{COPY}\ \frac{(A_{vu} \rightarrow \alpha B_u \beta) \in P(\Gamma_{i-1}) \qquad \mathit{nullable}(\alpha, \Gamma_{i-1})}{(A_{vut_i} \rightarrow B_{ut_i} \beta) \in P(\Gamma_i)}$$

$$\text{SCAN}\ \frac{(A_u \rightarrow \alpha t_i \beta) \in P(\Gamma_{i-1}) \qquad \mathit{nullable}(\alpha, \Gamma_{i-1})}{(A_{ut_i} \rightarrow \beta) \in P(\Gamma_i)}$$

(a) Pruned derivative grammar rules for generating grammar Γ_i from grammar Γ_{i-1}

$$\text{ORIGINAL}\ \frac{p \in P(\Gamma)}{p \in P(\Delta_i)} \qquad \text{EPSILON}\ \frac{(B_u \rightarrow \epsilon) \in P(\Delta_i) \qquad (A_{vu} \rightarrow B_u \beta) \in P(\Delta_i)}{(A_{vu} \rightarrow \beta) \in P(\Delta_i)}$$

$$\text{COPY}\ \frac{(A_{vu} \rightarrow B_u \beta) \in P(\Delta_{i-1})}{(A_{vut_i} \rightarrow B_{ut_i} \beta) \in P(\Delta_i)} \qquad \text{SCAN}\ \frac{(A_u \rightarrow t_i \beta) \in P(\Delta_{i-1})}{(A_{ut_i} \rightarrow \beta) \in P(\Delta_i)}$$

(b) ϵ-reduced derivative grammars $\Delta_0, \ldots \Delta_n$

$$\text{ORIGINAL}\ \frac{p \in P(\Gamma)}{p \in P(\Phi_i)}$$

$$\text{COMPLETION}\ \frac{(B_u \rightarrow \epsilon) \in P(\Phi_i) \qquad u = t_{h+1} \ldots t_i \qquad (A_v \rightarrow B\beta) \in P(\Phi_h)}{(A_{vu} \rightarrow \beta) \in P(\Phi_i)}$$

$$\text{SCAN}\ \frac{(A_u \rightarrow t_i \beta) \in P(\Phi_{i-1})}{(A_{ut_i} \rightarrow \beta) \in P(\Phi_i)} \qquad \boxed{\text{COPY}\ \frac{(A_{vu} \rightarrow B_u \beta) \in P(\Phi_{i-1})}{(A_{vut_i} \rightarrow B_{ut_i} \beta) \in P(\Phi_i)}}$$

(c) COPY-optimized derivative grammars $\Phi_0 \ldots \Phi_n$

$$\text{INIT}\ \frac{(S \rightarrow \alpha) \in P(\Gamma)}{(S \rightarrow \alpha) \in \Sigma_0} \qquad \text{COMPLETION}\ \frac{(B_u \rightarrow \epsilon) \in \Sigma_i \qquad u = t_{h+1} \ldots t_i \qquad (A_v \rightarrow B\beta) \in \Sigma_h}{(A_{vu} \rightarrow \beta) \in \Sigma_i}$$

$$\text{PREDICTION}\ \frac{(A_v \rightarrow B\beta) \in \Sigma_i \qquad (B \rightarrow \gamma) \in P(\Gamma)}{(B \rightarrow \gamma) \in \Sigma_i} \qquad \text{SCAN}\ \frac{(A_u \rightarrow t_i \beta) \in \Sigma_{i-1}}{(A_{ut_i} \rightarrow \beta) \in \Sigma_i}$$

(d) On-demand expansion of nonterminals generating the sets of productions $\Sigma_0 \ldots \Sigma_n$

Fig. 11. Deriving the Earley recognizer from pruned derivative grammar rules. A is the same as A_ϵ. α, β, and γ are all in $(N \cup T)^*$. For brevity we use Γ_i, Δ_i, and Φ_i instead of $\Gamma_{t_1 \ldots t_i}$, $\Delta_{t_1 \ldots t_i}$, and $\Phi_{t_1 \ldots t_i}$ respectively.

rule, copied productions are like dead code in program optimization, so the COPY rule can be eliminated. In Figure 11c, the COPY rule has been sequestered in a box to highlight this point. If it is removed, the resulting sets of productions are no longer grammars since some productions are missing, but the remaining productions can be interpreted as assertions about how the recognizer is attempting to use substrings of the input string just as in the Earley recognizer.

5.2.3 On-Demand Introduction of Nonterminals. The ORIGINAL rule in Figure 11c introduces productions for all nonterminals of the original grammar Γ into each derivative grammar, but

this is usually not needed. In general, a production $q = (B_u \rightarrow \epsilon) \in \Phi_i$ is useful if (i) it is the final production $(S_w \rightarrow \epsilon) \in \Phi_n$ or (ii) $u = t_{h+1} \ldots t_i$ and there is a production $(A_v \rightarrow B\beta)$ in Φ_h so the COMPLETION rule can be applied to production q. Otherwise production q is useless since it is not of interest in itself nor is it the assumption of another inference rule. Looking back where the B production was introduced, this means that a production $p = (B \rightarrow \alpha)$ is needed in Φ_h only if (i) $p = (S \rightarrow \alpha)$ and $h = 0$, or (ii) there is a production $(A_v \rightarrow B\beta) \in \Phi_h$ in which B is the leftmost symbol on the righthand side. In Figure 11d, the ORIGINAL rule has been replaced by the INIT and PREDICTION rules to handle these two cases.

As mentioned above, the sets of productions constructed by these rules do not necessarily form complete grammars, so we denote these sets as Σ_i rather than as $P(\Sigma_i)$ in Figure 11d. Note that a valid sequence of derivative grammars can be generated by adding the productions that would otherwise be generated by the ORIGINAL and COPY rules. *This gives a procedure for recovering a sequence of derivative grammars from the state of the Earley recognizer at any point during recognition.*

5.2.4 Conversion to Earley Notation. Comparing the inference rules of Figure 11d with the rules for the Earley recognizer in Figure 9, we see that transforming the pruned derivative grammar rules as described in this section has produced the Earley recognizer, with two bookkeeping differences.

(i) Subscripts of derivative nonterminals in Figure 11d are substrings of the input string of the form $t_{h+1} \ldots t_i$. Since these substrings are always suffixes of the input string consumed up to that point, it is sufficient to track the starting position of this substring (h) instead of keeping the entire string. This is the tag in Earley entries. Alternatively, an Earley entry $<A \rightarrow \alpha \bullet \beta, h>$ in Σ_i can be written in derivative form as $(A_{t_{h+1} \ldots t_i} \rightarrow \beta)$.

(ii) The derivative grammar rules do not track the portion of the Earley items to the left of the dot since this plays no role in derivative-based recognition (or in the Earley recognizer for that matter). It is easy to alter the COMPLETION and SCAN rules in Figure 11d to track these symbols.

5.3 LL and LR grammars in the derivative approach

$$
\begin{array}{lcl}
E & \rightarrow & (+\ E\ E) \\
E & \rightarrow & (-\ E\ E) \\
E & \rightarrow & n
\end{array}
$$

(a) Γ

$$
\begin{array}{lcl}
E_{(} & \rightarrow & +\ E\ E) \\
E_{(} & \rightarrow & -\ E\ E) \leftarrow \text{ignored} \\
E & \rightarrow & (+\ E\ E) \\
E & \rightarrow & (-\ E\ E) \\
E & \rightarrow & n
\end{array}
$$

(b) $\Gamma_{(}$

$$
\begin{array}{lcl}
E_{(+} & \rightarrow & E\ E) \\
E & \rightarrow & (+\ E\ E) \\
E & \rightarrow & (-\ E\ E) \\
E & \rightarrow & n
\end{array}
$$

(c) $\Gamma_{(+}$

$$
\begin{array}{lcl}
E_{(+n} & \rightarrow & E_n\ E) \\
E_n & \rightarrow & \epsilon \\
E & \rightarrow & (+\ E\ E) \\
E & \rightarrow & (-\ E\ E) \\
E & \rightarrow & n
\end{array}
$$

(d) $\Gamma_{(+n}$

$$
\begin{array}{lcl}
E_{(+nn} & \rightarrow & E_n) \\
E_n & \rightarrow & \epsilon \\
E & \rightarrow & (+\ E\ E) \\
E & \rightarrow & (-\ E\ E) \\
E & \rightarrow & n
\end{array}
$$

(e) $\Gamma_{(+nn}$

$$
\begin{array}{lcl}
E_{(+nn)} & \rightarrow & \epsilon \\
E & \rightarrow & (+\ E\ E) \\
E & \rightarrow & (-\ E\ E) \\
E & \rightarrow & n
\end{array}
$$

(f) $\Gamma_{(+nn)}$

Fig. 12. SLL(2) parsing with derivative grammars: grammar is Γ and input string is $w = (+\ n\ n)$

LL and LR parsers use lookahead to limit the exploration of possible derivations during parsing. In particular, LL(k) parsers (more precisely, SLL(k) parsers) use k symbols of lookahead to choose

a unique production to expand the leftmost nonterminal in a left-sentential form. The derivative grammar approach can be extended to use lookahead in the same way: for an SLL(k) grammar, lookahead is used to pick a unique production for differentiating a nonterminal with respect to the input symbol, as explained in more detail below. Parsing of LL(k) grammars is usually implemented by reducing it to SLL(k) parsing: given an LL(k) grammar Γ, a process similar to procedure cloning in programming languages clones nonterminals and their productions to generate an equivalent SLL(k) grammar Γ', for which an SLL(k) parser can be used. This same preprocessing step can be used in the derivative grammar approach as well.

Figure 12 shows a small example that illustrates how lookahead can be used to implement SLL(k) grammars in the derivative grammar approach. The grammar Γ is an SLL(2) grammar: given 2 symbols of lookahead, a recursive descent parser can determine which "E" production to use to expand the leftmost E in a sentential form: if the lookahead symbols are "(+", use the first production; if they are "(-", use the second production; and otherwise, use the third production.

This can be described using the derivative grammar approach by using lookahead to pick a unique production when differentiating a nonterminal with respect to the input symbol. This is shown in the first step of Figure 12: the canonical approach described in the paper would differentiate both the first and second productions with respect to '(' and produce two productions for $E_($ as shown in Figure 12(b). Using lookahead, the second of these productions would not be generated. For the remaining derivative grammars in this example, one symbol of lookahead is enough as the derivative grammar approach implicitly incorporates this already.

Although this example is simple, it illustrates how lookahead can be used when parsing with SLL(k) grammars using the derivative approach. The copy optimization described in Section 5.2 can be used to eliminate the generation of copied productions.

A similar approach can be used to leverage lookahead for parsing with SLR(k) grammars. In standard shift-reduce parsers for SLR(k) grammars, lookahead is used to resolve reduce-reduce and shift-reduce conflicts. The equivalent of "reduce" in the derivative grammar approach is a nullable nonterminal since intuitively, a nullable nonterminal indicates that the end of a production has been reached. In the derivative grammar approach, we compute nullable nonterminals at each step and can use lookahead to remove some productions from further consideration when generating the next derivative grammar. Disambiguating ambiguous grammars using precedence and associativity, as is done in shift-reduce parsers like Yacc, can be described using this idea.

6 EXPERIMENTAL RESULTS

While the focus of this paper is on derivative grammars and their relationship to context-free grammar parsing algorithms, we present an experimental study using derp-3 (an existing PWD parser), Marpa (a highly optimized Earley parser), and ANTLR (which only supports LL(*) grammars). The study was motivated by an online discussion with the author of Marpa, Jeffrey Kegler, who noted that although both PWD and Marpa can handle general—possibly ambiguous—context-free grammars, no one had yet published an experimental comparison of the PWD approach with an optimized Earley parser like Marpa. The results in this section are not meant to be exhaustive but they show broad trends with a simple message.

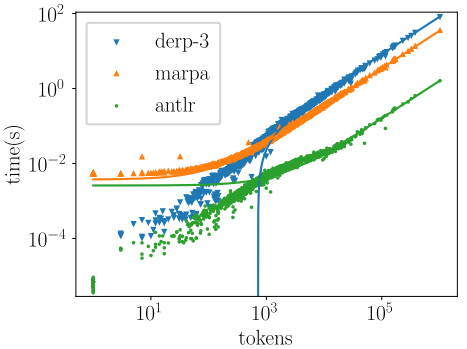


Fig. 13. Times for parsing the Python files in the SymPy python library using derp-3, Marpa, and ANTLR. Times are the average of 20 runs.

Figure 13 shows the time spent to parse input files using the grammar for Python 3.6. derp-3, Marpa R2 version 4.0, and ANTLR 4.7.1 were used for these experiments. derp-3 is the prototype provided to demonstrate the work in [Adams et al. 2016]. Racket 7.1 [Flatt and PLT 2010] was used to run derp-3. Marpa is an industrial grade implementation of a variant of Earley's algorithm [Kegler 2017][Leo 1991]. Perl 5.22.2 [Wall 2000] was used to run Marpa. ANTLR [Parr and Fisher 2011] is an industrial grade parser generator for LL(*) parsers that can generate parsers in a variety of languages. The C++ ANTLR backend was used for these tests.

The parsers were run on preprocessed versions of all the Python files in the SymPy 1.3 source repository [Meurer et al. 2017]. SymPy is a computer algebra system written exclusively in Python. The Python "tokenize" library was used to lex the input files into tokens so that the parsers could process files containing lexed versions of their inputs without relying on additional custom code for lexing the original Python files. The takeaway points from Figure 13 are the following.

- Marpa shows some constant time overhead for small inputs, but outperforms derp-3 by roughly a factor of 2 for large inputs. One caveat in the case of derp-3 is that, as Adams *et al.* [Adams et al. 2016] note, there is a significant cost in using Racket as opposed to a lower-level compiled language. Similarly, our study used Marpa's Perl interface, which incurs some overheads beyond what would be expected when using libmarpa (the C library forming the core of Marpa's implementation) or a similar compiled Earley parser. These results suggest that PWD is competitive with Earley parsing for general context-free grammars.
- ANTLR required a slower first run to fill in some of the metadata within the parser, but it outperforms both of the other parsers tested in the slower first run of the parser and also in subsequent runs. The times reported for ANTLR are the average of 20 runs after the slow first run. For large programs, ANTLR runs roughly 10-20 times faster than derp-3 and Marpa. ANTLR accepts LL(*) grammars, which are a subset of general context-free grammars, so it is more specialized than PWD and Earley parsers but it can exploit this specialization to obtain better performance.

7 CONCLUSIONS

In this paper, we argued that the notion of derivative languages underlies all online context-free grammar recognizers that process the input string from left to right without backtracking. This is a very broad class that includes the LL, LR and Earley recognizers.

Prior work [Adams et al. 2016; Brachthäuser et al. 2016; Danielsson 2010; Might et al. 2011] has described derivative languages in operational terms by encoding the grammar as a collection of mutually recursive algebraic data types in a functional language like Haskell. Although this approach is elegant, it is difficult to reason about it abstractly; for example, the connection between derivative languages and classical parsing methods has remained unclear.

The symbolic approach to derivative languages proposed in this paper uses a small number of inference rules to generate a sequence of derivative grammars from the original grammar, under the control of the input string. We showed that formulating recognition and parsing in terms of derivative grammars and inference rules has several advantages. Parsing can be done in $O(n^2 log(n))$ space and $O(n^3)$ time for a string of length n, improving prior bounds in the literature. Furthermore, formal connections can be made with classical recognition algorithms like Earley.

A APPENDIX

A.1 Proof of Theorem 3.2

THEOREM. *The language generated by* Γ_t *is the derivative with respect to* t *of the language generated by* Γ*, i.e.:*

$$L(\Gamma_t) = D_t(L(\Gamma)). \tag{7}$$

PROOF. By Definition 1.1, the theorem is equivalent to the statement that $x \in L(\Gamma_t)$ if and only if $tx \in L(\Gamma)$. We establish the stronger statement that, for any nonterminal $A \in N$ and any string $x \in T^*$, $A_t \Rightarrow^*_{\Gamma_t} x$ if and only if $A \Rightarrow^*_{\Gamma} tx$, which yields the desired conclusion by setting $A = S$.

Part I: If $A_t \Rightarrow^*_{\Gamma_t} x$ then $A \Rightarrow^*_{\Gamma} tx$. The proof is by induction on the number of steps, r, in the derivation of x from A_t.

Base case ($r = 1$). If the derivation $A_t \Rightarrow^*_{\Gamma_t} x$ has exactly one step, that step must be based on a production $(A_t \rightarrow x)$ in P_t. By Definition 3.1 of Γ_t, there exists in P a D-parent production $(A \rightarrow \alpha X \beta)$, where α is nullable, $X = t$, and $\beta = x$. Then, we have: $A \Rightarrow_{\Gamma} \alpha X \beta \Rightarrow^*_{\Gamma} \epsilon X \beta = t\beta = tx$, which establishes the claim.

Inductive step (from $r-1$ to $r > 1$). Assume that $A_t \Rightarrow^*_{\Gamma_t} x$ is a derivation of $r > 1$ steps. Let the production used in the first step of this derivation be $(A_t \rightarrow X_t \beta)$, which is necessarily in P_t and has a D-parent production, in P, of the form $(A \rightarrow \alpha X \beta)$ where, by inference rule 2, we have $nullable(\alpha, \Gamma)$.

In the given derivation, let $X_t \Rightarrow^*_{\Gamma_t} x'$ and $\beta \Rightarrow^*_{\Gamma} x''$, with $x'x'' = x$. We see that

$$A \Rightarrow^*_{\Gamma} \alpha X \beta \Rightarrow^*_{\Gamma} X\beta \Rightarrow^*_{\Gamma} Xx''.$$

Recalling that $X \in N \cup \{t\}$, we separately consider two cases.
Case $X \in N$. Observe that $X_t \Rightarrow^*_{\Gamma_t} x'$ by a sequence of fewer than r steps. Then, by the inductive hypothesis, we have $X \Rightarrow^*_{\Gamma} tx'$, hence $A \Rightarrow^*_{\Gamma} Xx'' \Rightarrow^*_{\Gamma} tx'x'' = tx$.
Case $X = t$. Here, $X_t = \epsilon$, so that $x = x''$. Moreover,
$A \Rightarrow^*_{\Gamma} Xx'' \Rightarrow^*_{\Gamma} tx'' = tx$.

Part II: If $A \Rightarrow^*_{\Gamma} tx$, then $A_t \Rightarrow^*_{\Gamma_t} x$. The argument is by induction on the number s of steps in the derivation of tx from A.

Base case ($s = 1$). If derivation $A \Rightarrow^*_{\Gamma} tx$ includes exactly one step, that step can only be based on a production $(A \rightarrow tx)$ (in P). Then, P_t will contain the production $(A_t \rightarrow x)$, which provides a derivation $A_t \Rightarrow^*_{\Gamma_t} x$.

Inductive step (from $s-1$ to $s > 1$). Assume that $A \Rightarrow^*_{\Gamma} tx$, according to a derivation in $s > 1$ steps. Let the production used in the first step be $(A \rightarrow \alpha B \beta)$, where $\alpha \Rightarrow^*_{\Gamma} \epsilon$, $B \Rightarrow^*_{\Gamma} tx'$, and $\beta \Rightarrow^*_{\Gamma} x''$, with $x'x'' = x$. Since α is nullable, by Definition 3.1, P_t will contain production $(A_t \rightarrow B_t \beta)$. By the inductive hypothesis, $B \Rightarrow^*_{\Gamma} tx'$ (in fewer than s steps) implies that $B_t \Rightarrow^*_{\Gamma_t} x'$. Furthermore, $\beta \Rightarrow^*_{\Gamma} x''$ implies $\beta \Rightarrow^*_{\Gamma_t} x''$, since all productions in Γ are also productions in Γ_t. Therefore $A_t \Rightarrow^*_{\Gamma_t} B_t\beta \Rightarrow^*_{\Gamma_t} x'x'' = x$. □

A.2 Proof of Lemma 3.5

LEMMA. *The pruned derivative grammars of Definition 3.4 satisfy the following properties.*

(1) If $(W \rightarrow Y\xi)$ is a production in any derivative grammar, then $\xi \in (N \cup T)^*$, that is, all symbols in the righthand side of the production, except possibly for the first one (Y), are symbols of the original grammar Γ.

(2) Let $(W \to Y\xi)$ be a production in P_z where Y is a nonterminal. If $W = C_r$ and $Y = D_s$ (with $C, D \in N$), then s is a suffix of r (including the cases $s = \epsilon$ and $s = r$), and both r and s are suffixes of z.

(3) If $C_r \Rightarrow^*_{\Gamma_z} Y\xi$ is a derivation within grammar Γ_z with $C_r \in N \cup N_z$, (r a suffix of z), then $\xi \in (N \cup T)^*$ and, if $Y = D_s$ is a nonterminal, then s is a suffix of r (hence of z).

(4) Let $z = pr$ and $C \in N$. The set of productions for C_r in grammar Γ_z equals the set of productions for C_r in grammar Γ_r. (I.e., this set of productions is independent of p).

Proof. We separately argue each property below.

(1) Property (1) is trivially true for productions of $\Gamma_\epsilon = \Gamma$. Assuming it inductively true for productions of $\Gamma_{z'}$, it is established for $\Gamma_z = \Gamma_{z't_i}$, by considering inference rule 5. In fact, by the inductive assumption, $\beta \in (N \cup T)^*$, hence the inferred production $(A_{vt_i} \to X_{t_i}\beta) \in P_z$ also satisfies Property (1).

(2) Property (2) is trivially true for productions of $\Gamma_\epsilon = \Gamma$. Assuming it is inductively true for productions of $\Gamma_{z'}$, it is established for $\Gamma_z = \Gamma_{z't_i}$, by considering inference rule 5 and analyzing the following exhaustive cases.
(i) If $X = B_u \in (N \cup N_{z'})$ and $\alpha = \epsilon$ then, by the inductive hypothesis, u is a suffix of v and they are both suffixes of z'. The inferred production is $\left(A_{vt_i} \to B_{ut_i}\beta\right)$, so that Property (2) holds with $C_r = A_{vt_i}$ and $D_s = B_{ut_i}$. Clearly, $r = vt_i$ and $s = ut_i$ are both suffixes of $z = z't_i$.
(ii) If $X = B_u \in (N \cup N_{z'})$ and $\alpha \neq \epsilon$, then $u = \epsilon$, by Property (1). The inferred production is $\left(A_{vt_i} \to B_{t_i}\beta\right)$, so that property (2) is satisfied with $C_r = A_{vt_i}$ and $D_s = B_{t_i}$. Clearly, $r = vt_i$ and $s = t_i$ are both suffixes of $z = z't_i$.
(iii) If $X = t_i$, then $X_{t_i} = \epsilon$ and the inferred production is $\left(A_{vt_i} \to \beta\right)$. By property (1), $\beta \in (N \cup T)^*$, thus if the first symbol of β is a non teminal, it is of the form D_ϵ. Property (2) is then satisfied with $C_r = A_{vt_i}$ and $D_s = D_\epsilon$. Again, inductively v is a suffix of z' hence $r = vt_i$ and $s = \epsilon$ are both suffixes of $z = z't_i$.

(3) Property (3) is a simple corollary of Properties (1) and (2) and the definition of derivation within a grammar.

(4) The proof is trivial if $r = \epsilon$. Otherwise, let $r = r't_i$. We assume inductively that the set of productions for $C_{r'}$ is the same in $\Gamma_{pr'}$ and in $\Gamma_{r'}$. Moreover, as a consequence of Property (2), a production in $\Gamma_{pr'}$ of the form $(C_{r'} \to \alpha X\beta)$ cannot include symbols that are not also in $\Gamma_{r'}$. Hence, α is nullable in $\Gamma_{pr'}$ if and only if it is nullable in $\Gamma_{r'}$. Therefore, the D-children of said production generated by inference rule 5 (which are of the form $\left(C_r \to X_{t_i}\beta\right)$) are the same in Γ_{pr} and in Γ_r, as claimed.

□

A.3 Proof of Theorem 3.6

Theorem. *With reference to the notation introduced in Definition 3.4, for each prefix z of w, the pruned derivative grammar Γ_z generates the derivative with respect to z of the language generated by Γ, i.e.:*

$$L(\Gamma_z) = D_z(L(\Gamma)). \tag{8}$$

Proof. The proof will proceed by induction on the length i of z.

Base cases ($i = 0, 1$). For $i = 0$, i.e., $w = \epsilon$, Equation 6 specializes to $L(\Gamma_\epsilon) = D_\epsilon(L(\Gamma))$, which is a simple consequence of the facts that $\Gamma_\epsilon = \Gamma$ (according to Definition 3.4) and that $D_\epsilon(L(\Gamma)) = L(\Gamma)$, (by Definition 1.1). For $i = 1$, i.e., $w = t_1$, it is easy to verify that Definition 3.4 of the pruned derivative grammar Γ_{t_1} actually yields the same derivative grammar as Definition 3.1. Thus, Equation 6 follows from Theorem 3.2.

Inductive step (from $i-1$ to $i > 1$). Let $z = t_1 \dots t_{i-1} t_i = z' t_i$ and let $\Gamma_{z'} = <N \cup N_{z'}, T, P \cup P_{z'}, S_{z'}>$. By the inductive hypothesis, $L(\Gamma_{z'}) = D_{z'}(L(\Gamma))$. Consider now the grammar $\hat{\Gamma} = (\Gamma_{z'})_{t_i}$, that is, the first derivative of $\Gamma_{z'}$ with respect to t_i, according to Definition 1.1. Clearly, by Theorem 3.2, we have $L(\hat{\Gamma}) = D_{t_i}(D_{z'}(L(\Gamma))) = D_z(L(\Gamma))$. Then, the desired conclusion is an immediate corollary of the following claim.

Claim: Grammars Γ_z and $\hat{\Gamma}$ generate the same language, i.e., $L(\Gamma_z) = L(\hat{\Gamma})$.

Proof of Claim. In order to compare the languages generated by Γ_z and $\hat{\Gamma}$, we trace how each of these two grammars is obtained from $\Gamma_{z'} = <N \cup N_{z'}, T, P \cup P_{z'}, S_{z'}>$. Applying Definition 3.4, we see that $\Gamma_z = <N \cup N_z, T, P \cup P_z, S_z>$, where P_z is the set of those productions resulting from the application of inference rule 5 to the productions in $P \cup P_{z'}$. Applying Definition 1.1 and considering that, when applied to $P \cup P_{z'}$, inference rule 2 has the same effect as inference rule 5, we see that $\hat{\Gamma} = <N \cup N_{z'} \cup N_z, T, P \cup P_{z'} \cup P_z, S_z>$. We will now argue that the set of productions $P_{z'}$ and the set of nonterminals $N_{z'}$ can be removed from the expression for $\hat{\Gamma}$ without affecting the language generated by the grammar. We begin by observing that, by Property (3) of Lemma 3.5, any nonterminal symbol included in a derivation starting with S_z will be of the form D_s, where $D \in N$ and s is a suffix of z. If either $s = \epsilon$ or s is not a suffix of z', then by Property (2) of Lemma 3.5, $P_{z'}$ will contain no production for D_s, hence no production of $P_{z'}$ can be used when expanding D_s. Consider then the remaining case where s is a nonempty suffix of z' (it is easy to see that $s = (t_i)^k$, for some integer $k \geq 1$). Then, by Property (4) of Lemma 3.5, any production for D_s in $P_{z'}$ is also in P_z, hence the inclusion of $P_{z'}$ in the expression for the productions of Γ_z is redundant. □

A.4 Proof of Theorem 3.7

Theorem. *Given Γ and $w = t_1 t_2 \dots t_n$, grammar Γ_w can be stored in space $O(n^2)$ and constructed in time $O(n^3)$.*

Proof. To analyze space and time for building the pruned derivative grammars, it is useful to derive an upper bound to the number of their productions. To this end, observe that each production in

Rule 5. As a corollary of Properties (1) and (2) of Lemma 3.5, a production in P_z is either of the form $(C_r \rightarrow D_s \xi)$ or of the form $(C_r \rightarrow \xi)$, where $\xi \in (N \cup T)^*$. It is easy to see that the root production is of the form $(C \rightarrow D\xi)$ in the first case and $(C \rightarrow \eta\xi)$ in the second case, where $\eta \in (N \cup T)^*$. Given $z = t_1 \dots t_i$, we have

$$|P_z| \leq |P| \ell(\Gamma)(i+1)i/2 = O(i^2), \tag{9}$$

where $\ell(\Gamma)$ denotes the maximum number of symbols in the righthand side of any production of Γ. In fact, for each production in P, there are at most $\ell(\Gamma)(i+1)i/2$ descendants in P_z, where the factor $\ell(\Gamma)$ is an upper bound to the number of prefixes η and the factor $(i+1)i/2$ is an upper bound to the number of (r, s) pairs, considering that s is a suffix of r, r is a suffix of z, and z has i characters.

At any stage i of the construction of the pruned derivative grammars, it is sufficient to store grammars $\Gamma_{t_1 \dots t_{i-1}}$, and $\Gamma_{t_1 \dots t_i}$, as well as some auxiliary information of size proportional to that of the grammars. By Equation 9, we see that $O(i^2) = O(n^2)$ space is sufficient.

As part of the construction, the set of nullable nonterminals must be computed for each grammar. This can be accomplished in time proportional to the size of the grammar using well-known nullability algorithms [Aho et al. 1986; Sippu and Soisalon-Soininen 1988], briefly reviewed also in Section A.5. Once the nullability information is available for $\Gamma_{z'} = \Gamma_{t_1, \dots, t_{i-1}}$, it is straightforward to build $\Gamma_z = \Gamma_{t_1 \dots t_i}$, in time linear in the size of $\Gamma_{z'}$, that is, in time $O(i^2) = O(n^2)$. Thus, the construction of the n grammars leading to Γ_w can be completed in time $T = O(n^3)$. □

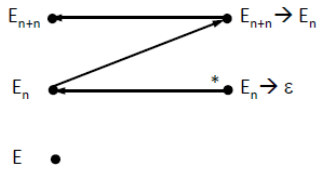
A.5 Nullability Computation

Finding nullable nonterminals is a standard procedure for CFGs [Aho et al. 1986; Sippu and Soisalon-Soininen 1988] but the structure of derivative grammars allows additional optimizations.

It is useful to build a bipartite graph with directed edges, $\mathcal{B}_{null} = (N, P_N, E_{NP} + E_{PN})$, where (i) the nodes in N are the nonterminals; (ii) the nodes in $P_N \subseteq P$ are the productions whose righthand side does not contain any terminal; (iii) for each production $p = (A \rightarrow \alpha)$, there is an edge in E_{NP} from each nonterminal of α to p (there are no edges if $\alpha = \epsilon$)and an edge in E_{PN} from p to A.

Nullability information flows along the edges according to the following procedure. Nodes for ϵ-productions are initially marked as nullable; then a loop is executed where (a) the E_{PN}-neighbors of nullable productions are marked as nullable and (b) the productions with all their E_{NP}-neighbors nullable are marked as nullable, until convergence (*i.e.*, until no new node is marked during an iteration). This procedure can be easily implemented in $O(|G|)$ time and space. Figure 14(a) shows the bipartite graph for grammar Γ_{n+n} in Figure 3 with nullability marked initially with a "*".

Although nullability is enough for recognition, parsing requires a derivation $A \Rightarrow^{+} \epsilon$, for every nullable A. The nullability procedure can be extended to mark, for any nullable A, the E_{PN} edge (p, A) that (first) leads to marking A. An inductive argument shows that a leftmost null derivation for A can be constructed, starting with $p = (A \rightarrow \alpha)$ and continuing with the leftmost null derivations of the nonterminals in α (from left to right).



(a) Bipartite graph for nullability

Nonterminal	Production
E_{n+n}	$E_{n+n} \rightarrow E_n$
E_n	$E_n \rightarrow \varepsilon$

(b) Map for ε derivations

Fig. 14. Nullability computation in grammar E_{n+n} of Fig. 3

Optimizations for derivative grammars. Due to Property 4 of Lemma 3.5, although a given nonterminal C_r may belong to derivative grammar Γ_z for different prefixes $z = pr$ of the input string w, the nullability of C_r is independent of p (and z). Thus, whether C_r is nullable and, if so, a corresponding ϵ-derivation, can be memoized whenever the derivative nonterminals and productions are.

Second, nullability computation for derivative nonterminals can be simplified by exploiting Property 1 of Lemma 3.5. This property implies that a production p for a derivative nonterminal can play a role in an ϵ-derivation only if it is of the form (0) $(C_r \rightarrow \xi)$ or of the form (1) $(C_r \rightarrow D_s \xi)$, with ξ a sequence of nullable nonterminals of Γ, and s a suffix of r. A production of form (0) implies that C_r is nullable, while a production of form (1) imples that, if D_s is nullable, so is C_r. We can then build a directed graph whose nodes are the derivative nonterminals (of a given grammar Γ_z), with an edge from D_s to C_r for every production of form (1), where each lefthand side of a production of form (0) is marked. It is easy to see that nullable nonterminals are exactly those reachable from some marked terminal. Labelling each edge with the corresponding production; marking, for each nullable nonterminal, the first edge by which it is reached; and utilizing the null derivations for nonterminals in Γ, one can easily reconstruct an ϵ-derivation for a nullable derivative nonterminal.

A.6 Space and Time Complexity of Parsing by Pruned Derivative Grammars

Theorem A.1. *Given a grammar Γ, a parse tree for a string $w = t_1 t_2 \ldots t_n \in L(\Gamma)$ can be constructed in time $O(n^3)$ and space $O(n^2 \log n)$.*

Proof. (Sketch) Throughout this proof, we adopt the following notation for the grammars constructed by the recognizer and used by the parser: $\Gamma_0 = \Gamma$ and $\Gamma_i = \Gamma_{t_1 \ldots t_i}$, for $i = 1, \ldots, n$. As

argued in the proof of Theorem 3.7, Γ_i can be stored in space $O(i^2+1) = O(n^2)$. In addition, let D_n denote the derivation $S_{t_1 \ldots tn} \Rightarrow^*_{\Gamma_i} \epsilon$, constructed by the recognizer, and for $i = 0, \ldots, n-1$, let D_i denote the derivation $S_{t_1 \ldots t_i} \Rightarrow^*_{\Gamma_i} t_{1+1} \ldots t_n$, constructed by the parser.

For convenience, we separately analyze the following components of the complete parsing procedure, using the abbreviations in parentheses as subscripts of the corresponding time and space requirements: (rc) recognition, which outputs derivation D_0; (gr) the construction of the grammars in the order $\Gamma_n, \ldots, \Gamma_1$ useful to the parser; (dr) the construction of derivations $D_{n-1}, \ldots, D_0$, starting from D_n; and (tr) the conversion of D_0 to a parse tree.

From Theorem 3.7, $T_{rc} = O(n^3)$ and $S_{rc} = O(n^2)$. The recognizer actually constructs all the necessary grammars, but only stores two of them at the time; if all grammars were simultaneously stored, the space would become $O(n^3)$. This space can be substantially reduced, at the expense of some recomputation, which however increases time only by a constant factor. This goal is accomplished by the following recursive procedure $reverse(i, j)$, which takes as input Γ_{i-1} and produces as outputs the sequence $\Gamma_j, \ldots, \Gamma_i$. The basis of the recursion, for $j = i$, obtains Γ_i from Γ_{i-1}, as the recognizer would do. In general, setting $k = \lfloor (i+j)/2 \rfloor$, $reverse(i, j)$ internally stores the input Γ_{i-1}; computes and stores Γ_{k-1} (as the recognizer would do, in time $O(n^2(k-i)$, that is $O(n^2)$ per grammar, and $O(n^2)$ space); makes the recursive call $reverse(k, j)$; releases the storage for Γ_{k-1}; makes the call $reverse(i, k-1)$; and releases the storage for Γ_{i-1}. The root call made by the parser is $reverse(1, n)$, given Γ_0 as the input. Standard techniques to analyze divide and conquer algorithms lead to the formulation and solution of recurrence relations for both time and space:

$$\begin{aligned} T_{gr}(n) &= 2T_{gr}(n/2) + O(n^3) = O(n^3); \\ S_{gr}(n) &= O(g) + S_{gr}(n/2) = O(g \log n) = O(n^2 \log n); \end{aligned}$$

where $g = O(n^2)$ is an upper bound to the space for one grammar; since g is independent of the size of the subproblems, its dependence upon n is plugged in only after solving the recurrence.

Although, for convenience, we have described *reverse* as a stand alone procedure, in an implementation of the parser its execution must be interleaved with the procedure that, at a stage where Γ_i and Γ_{i-1} are available, constructs D_{i-1} from D_i, based on P-parent sequences. Denoting by $|D_i|$ the number of productions in derivation D_i, we can easily see that

$$\begin{aligned} T_{dr}(n) &= O(\sum_0^n |D_i|) = O(n^2); \\ S_{gr}(n) &= O(\max_0^n |D_i|) = O(n). \end{aligned}$$

It remains to analyze the size of the D_i's to show that $D_i = O(n)$, as assumed in the above equations. We begin by establishing the bound $D_n = O(|N|n)$. In the shortest epsilon derivation the number of productions with derivative nonterminals is at most $|N|n$, since no nonterminal can appear twice with the same index and the length of the index cannot increase (see Lemma 3.5). Thus, a run with the same index can include at most $|N|$ different nonterminals and there are at most n distinct runs. A derivation without repeated derivative nonterminals can be obtained from the data structures introduced in Section A.5. A detailed analysis shows that size increase during backward inference, where productions are replaced by their P-parent sequences, follows the bound $|D_i| = O(n)$.

The conversion of D_0 into a parse tree can easily be done is linear time and space, that is, $T_{tr}, S_{tr} = O(|D_0|) = O(n)$. Putting the pieces together, we obtain the claimed complexity results:

$$\begin{aligned} T(n) &= T_{rc}(n) + T_{gr}(n) + T_{dr}(n) + T_{tr}(n) = O(n^3 + n^3 + n^2 + n^2) = O(n^3); \\ S(n) &= S_{rc}(n) + S_{gr}(n) + S_{dr}(n) + S_{tr}(n) = O(n^2 + n^2 \log n + n + n) = O(n^2 \log n). \end{aligned}$$

□

A.7 Proof of Lemma 5.1

Lemma. *Given grammar $G = <N, T, P, S>$, the ϵ-reduction of G is the grammar $G_\epsilon = <N, T, P_\epsilon, S>$ in which the productions of P_ϵ are defined by the following rules:*

$$\text{ORIGINAL}\ \frac{p \in P}{p \in P_\epsilon} \qquad \text{EPSILON}\ \frac{(A \rightarrow B\beta) \in P_\epsilon \qquad (B \rightarrow \epsilon) \in P_\epsilon}{(A \rightarrow \beta) \in P_\epsilon}$$

Then (i) G and G_ϵ generate the same language, (ii) if A is nullable in G, P_ϵ contains the production $(A \rightarrow \epsilon)$, and (iii) if $(A \rightarrow \alpha\beta) \in P$ and nullable(α), then $(A \rightarrow \beta) \in P_\epsilon$.

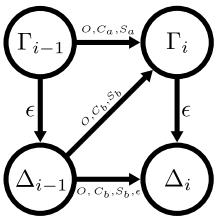


Fig. 15. Commuting diagram used to prove Theorem 5.2. ϵ refers to the EPSILON inference rule of Lemma 5.1, O, C_a, S_a, C_b, and S_b refer to the rules from Figures 11a and 11b.

Unlike the inference rules used to define grammars up to this point in the paper, the EPSILON rule is recursive because the grammar G_ϵ is used in both the assumptions and conclusion. To prove Lemma 5.1, we can construct a sequence of grammars $G = G_0, G_1, \ldots, G_q = G_\epsilon$ in which each successive grammar G_{i+1} contains all the productions in the previous grammar G_i as well as new productions obtained from G_i by the following *non-recursive* rule:

$$\text{EPSILON}_1\ \frac{(A \rightarrow B\beta) \in P(G_i) \qquad (B \rightarrow \epsilon) \in P(G_i)}{(A \rightarrow \beta) \in P(G_{i+1})}$$

The sequence of grammars $G_0, \ldots, G_q$ is finite because (i) a production can be used at most once in generating this sequence, and (ii) the production $(A \rightarrow \beta)$ in the conclusion of the EPSILON rule is strictly smaller in length than the production $A \rightarrow B\beta$ in the assumption, so an unbounded number of new productions cannot be generated.

Figure 11b shows how ϵ-reduction can be used to simplify the COPY and SCAN rules of Figure 11a. Instead of computing *nullable* to advance over nullable nonterminals when applying the COPY and SCAN rules as is done in Figure 11a, the inference rules in Figure 11b strip these nonterminals from the productions using the EPSILON rule before applying the simplified COPY and SCAN rules.

Proof. (i) Consider the grammar sequence $G = G_0, G_1, \ldots, G_q = G_\epsilon$. The proof is an easy induction on the length of this sequence, and the inductive step is proved by noticing that given a leftmost derivation of a terminal string in G_i, the use of the productions $(A \rightarrow B\beta)$ and $(B \rightarrow \beta)$ in sequence in such a derivation can be replaced by the production $(A \rightarrow \beta)$ in G_{i+1} and vice versa.

(ii) If A is nullable in G, it must be nullable in G_ϵ since every production in G is also in G_ϵ. Consider a leftmost derivation of ϵ from A in G_ϵ (say $A \rightarrow \alpha_1 \rightarrow \alpha_2 \ldots \rightarrow \epsilon$). Note that the productions used in this derivation must be of the form $(B \rightarrow \epsilon)$, $(B \rightarrow C)$ or $(B \rightarrow \beta)$ where β is a string of two or more nonterminals. Given such a leftmost derivation, the proof below constructs a derivation $A \Rightarrow^* \epsilon$, showing that there must be a production $A \rightarrow \epsilon$ in G_ϵ.

In the given derivation, the first string A has one symbol and the last string ϵ has zero symbols, so there must at least one place in this derivation where the length of the string is reduced after the derivation step. Consider the first place in the derivation where this happens.

If this happens at the beginning of the derivation, there is a production $(A \rightarrow \epsilon)$. Otherwise, suppose this happens at $i \geq 1$. Then the successive strings in the derivation can be written as follows: $\alpha_{i-1} = C\delta \rightarrow D\gamma\delta \rightarrow \gamma\delta = \alpha_{i+1}$. In this sequence, the production $C \rightarrow D\gamma$ was applied first and then the production $(D \rightarrow \epsilon)$. Since G_ϵ is closed under the EPSILON rule, there must be a production

$(C \rightarrow \gamma)$. Using this rule to rewrite C in α_{i-1}, we get a shorter leftmost derivation of ϵ from A in G_ϵ. Repeating this argument, we ultimately produce a derivation $A \rightarrow \epsilon$, from which the result follows.

(iii) Since $(A \rightarrow \alpha\beta) \in P$, it follows that $(A \rightarrow \alpha\beta) \in P_\epsilon$. Let $\alpha = B_1 B_2 \ldots B_k$. Since α is nullable in P, the nonterminals B_i are nullable in P. From part (ii), this means that $(B_i \rightarrow \epsilon) \in P_\epsilon$. Since P_ϵ is closed under the EPSILON rule, it follows that $(A \rightarrow \epsilon) \in P_\epsilon$. □

A.8 Proof of Theorem 5.2

Theorem. *Given a grammar Γ and an input string $w = t_1 t_2 \ldots t_n$, let the sequences of derivative grammars produced by the rules in Figure 11a and Figure 11b be $\Gamma_0, \Gamma_1 \ldots \Gamma_n$ and $\Delta_0, \Delta_1, \ldots \Delta_n$ respectively. Γ_i and Δ_i generate the same language.*

Proof. We show by induction on i that Δ_i is the ϵ-reduction of Γ_i, from which the required result follows by Lemma 5.1. For $i = 0$, the result follows from Lemma 5.1. The proof strategy for the inductive step is shown in Figure 15.

First we show that the upper triangle in this diagram commutes; that is, (i) applying the rules of Figure 11a to Γ_{i-1} produces the same grammar as (ii) applying the ORIGINAL, COPY and SCAN rules of Figure 11b to the ϵ-reduction of Γ_{i-1}. The proof considers the ORIGINAL, COPY and SCAN rules in the two sets of rules, and is a routine application of Lemma 5.1, so it is omitted.

Second we show that the bottom triangle in Figure 15 commutes; that is, (i) applying the ORIGINAL, COPY and SCAN rules of Figure 11b to Δ_{i-1} and then taking the ϵ-reduction of the resulting grammar produces the same grammar as (ii) applying the rules of Figure 11b to Δ_{i-1}. This follows from the fact that the assumptions in the ORIGINAL, COPY and SCAN rules in Figure 11b do not refer to Δ_i, whereas the conclusion of EPSILON rule refers only to Δ_i.

Putting the two triangles of Figure 15 together proves the inductive step. □

A.9 Correctness of Copy Elimination

The following result was needed in Section 5.2.2 to prove the correctness of copy elimination.

Theorem. *In Figure 11b, the EPSILON rule can be replaced with the COMPLETION rule from Figure 11c without changing the generated grammar.*

We show that in the sequence of derivative grammars $\Delta_0, \Delta_1, \ldots \Delta_n$ generated by the inference rules of Figure 11b, $(A_{vu} \rightarrow B_u \beta) \in P(\Delta_i)$ iff $(A_v \rightarrow B\beta) \in P(\Delta_h)$, proving the theorem.

Proof. If $u = \epsilon$, this is obviously true. Otherwise $u = t_{h+1} \ldots t_i \neq \epsilon$ and by common assumption in the EPSILON and COMPLETION rules, $(B_u \rightarrow \epsilon) \in P(\Delta_i)$.

($\Longrightarrow$) Suppose $(A_{vu} \rightarrow B_u \beta) \in P(\Delta_i)$. Examining the inference rules in Figure 11b, we see that since $u \neq \epsilon$, this production can only be generated by the COPY rule since it is the only rule that generates productions with a leading derivative nonterminal on the righthand side. Thus the D-parent production is $(A_{v t_{h+1} \ldots t_{i-1}} \rightarrow B_{t_{h+1} \ldots t_{i-1}} \beta)$ in $P(\Delta_{i-1})$. If the leading nonterminal on righthand side of this production is a derivative nonterminal, this production also must have been generated by the COPY rule so we look at its D-parent and so on. Each backward inference step peels off one terminal symbol from u, so in the end, we reach the production $(A_v \rightarrow B\beta)$ in $P(\Delta_h)$.

($\Longleftarrow$) Conversely, suppose $(A_v \rightarrow B\beta) \in P(\Delta_h)$. Applying the COPY rule repeatedly for $u = t_{h+1} \ldots t_i$, we see $(A_{vu} \rightarrow B_u \beta) \in P(\Delta_i)$. Therefore, in Figure 11b, the EPSILON rule can be replaced with the COMPLETION rule without changing the generated grammar. □

ACKNOWLEDGMENTS

This research was supported by NSF grants 1406355, 1618425, 1705092, and 1725322, and by DARPA contracts FA8750-16-2-0004 and FA8650-15-C-7563.

REFERENCES

Michael D. Adams, Celeste Hollenbeck, and Matthew Might. 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)*.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: principles, techniques, and tools.* Addison Wesley.

Jonathan Immanuel Brachthäuser, Tillmann Rendel, and Klaus Ostermann. 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)*.

Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *Journal of the ACM* 11, 4 (October 1964), 481–494.

Nils Anders Danielsson. 2010. Total Parser Combinators. *SIGPLAN Not.* 45, 9 (Sept. 2010).

Jay Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM* 13, 2 (Feb. 1970), 94–102.

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computability* (1st ed.). Addison-Wesley Publishing Co., Inc., Boston, MA, USA.

Jeffrey Kegler. 2017. Marpa–R2. https://github.com/jeffreykegler/Marpa--R2.

Joop M.I.M. Leo. 1991. A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead. *Theoretical Computer Science* 82, 1 (1991), 165 – 176. https://doi.org/10.1016/0304-3975(91)90180-A

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: A functional pearl. In *International Conference on Functional Programming*.

Terence Parr and Kathleen Fisher. 2011. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 425–436. https://doi.org/10.1145/1993498.1993548

Elizabeth Scott. 2008. SPPF-Style Parsing From Earley Recognisers. *Electronic Notes in Theoretical Computer Science* 203, 2 (2008), 53 – 67.

Seppo Sippu and Eljas Soisalon-Soininen. 1988. *Parsing theory.* Springer-Verlag.

Peter Thiemann. 2017. Partial Derivatives for Context-Free Languages. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*. Springer-Verlag New York, Inc., New York, NY, USA, 248–264. https://doi.org/10.1007/978-3-662-54458-7_15

Larry Wall. 2000. *Programming Perl* (3rd ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.