Toward a Theory of Input-Driven Locally Parsable Languages[☆,☆☆]

Stefano Crespi Reghizzi[a,c], Violetta Lonati[b], Dino Mandrioli[a], Matteo Pradella[a,c]

[a] *DEIB - Politecnico di Milano, P.zza L. da Vinci, 32, 20133 Milano, Italy*
[b] *DI - Università degli Studi di Milano, via Comelico 39/41, Milano, Italy*
[c] *IEIIT - CNR, Milano*

Abstract

If a context-free language enjoys the local parsability property then, no matter how the source string is segmented, each segment can be parsed independently, and an efficient parallel parsing algorithm becomes possible. The new class of locally chain parsable languages (LCPLs), included in the deterministic context-free language family, is here defined by means of the chain-driven automaton and characterized by decidable properties of grammar derivations. Such automaton decides whether to reduce or not a substring in a way purely driven by the terminal characters, thus extending the well-known concept of input-driven (ID) alias visibly pushdown machines. The LCPL family extends and improves the practically relevant Floyd's operator-precedence (OP) languages which are known to strictly include the ID languages, and for which a parallel-parser generator exists.

Keywords: Operator Precedence languages, Input-driven languages, Visibly Pushdown languages, Parallel Parsing

1. Introduction

Syntax analysis or parsing of context-free languages (CFLs) is a mature research area, and good parsing algorithms are available for the whole CFL family and for the deterministic (DCFL) subfamily that is of concern here. Yet the classical parsers are strictly serial and cannot profit from the parallelism of current computers. An exception is the parallel deterministic parser [4, 3] based on Floyd's [13] operator-precedence grammars (OPGs) and their languages (OPLs), which are included in the DCFL family. This is a data-parallel algorithm that is based on a theoretical property of OPGs, called *local parsability*: any arbitrary substring of a sentence can be deterministically parsed, returning the unique partial syntax-tree whose frontier matches the input string.

[☆] Work partially supported by PRIN 2010LYA9RH-006.
[☆☆] Preliminary versions of the results of this paper appeared in [8], and were presented at ICTCS'15.
Email addresses: stefano.crespireghizzi@polimi.it (Stefano Crespi Reghizzi), lonati@di.unimi.it (Violetta Lonati), dino.mandrioli@polimi.it (Dino Mandrioli), matteo.pradella@polimi.it (Matteo Pradella)

LL(k) and LR(k) grammars do not have this property, and their parsers must scan the input left-to-right to build leftmost derivations (or reversed-rightmost ones). On the contrary, the abstract recognizer of a locally parsable language, called a *local parser*, repeatedly looks in some arbitrary position inside the input string for a rule right-hand side (r.h.s.) and reduces it. The local parsability property ensures the correctness of the syntax tree thus obtained, no matter where and in which order reductions are applied.

The informal idea of local parsability is occasionally mentioned in old research on parallel parsing, and has been formalized for OPGs in [4]. Our contribution is the definition of a new and more general class of locally parsable languages: the family of languages to be called *Locally Chain Parsable* (LCPLs), which gains in generative capacity and bypasses some inconveniences of OPGs. We remark that OPLs in turn are a generalization of the well-known family of input-driven (alias visibly pushdown) languages (IDLs) [23, 2, 9], which are characterized by pushdown machines that choose to perform a push/pop/stay operation depending on the alphabetic class (opening/closing/internal) of the current input character, without a need to check the top of stack symbol.

To understand in what sense our LCPLs are input-driven, we first recall that IDLs generalize parenthesis languages, by taking the opening/closing characters as parentheses to be balanced, while the internal characters are handled by a finite-state automaton. It suffices a little thought to see that IDLs have the local parsability property, which also stems from the fact that IDLs are included in the OPL family. Yet, the rigid alphabetic 3-partition severely reduces their generative capacity. If we allow the parser decision whether to push, pop, or stay, to be based on a *pair* of adjacent terminal characters (more precisely on the precedence relation $\lessdot, \gtrdot, \doteq$ between them), instead of just one as in the IDLs, we obtain the OPL family, which has essentially the same closure and decidability properties [9, 18]. Loosely speaking, we may say that the input that drives the automaton for OPLs is a terminal string of length two.

With the LCPL definition, we move further: the automaton bases its decision whether to reduce or not a substring (which may contain nonterminals) on the purely terminal string orderly containing: the preceding terminal, the terminals of the substring, and the following terminal. Such triplet will be called a *chain* and the machine a *chain-driven automaton* (CDA).

The main results of this paper are presented along the following organization. After the Preliminaries, Section 3 introduces the chain-driven machine as a recognizer for all context-free languages. Section 4 defines local chain parsability for chain-driven automata and for grammars, and proves the two notions to be equivalent. Section 5 extends the definition of chains from embracing a single r.h.s. to representing portions of a whole derivation, and formulates a decidability condition for local chain parsability based on the absence of conflicts between chain sets. Section 6 proves structural properties of LCPLs, the strict inclusion thereof in the DCFL family, and investigates the behavior of the class with respect to classical language operations; precisely, it shows that, under suitable hypotheses of structural compatibility, the application of Boolean operations, but in general not concatenation and Kleene *, to two LCPLs produces a new LCPL; as a corollary, the inclusion problem between structurally compatible LCPLs is decidable, a key property for possible application of model checking techniques. Section 7 establishes the strict inclusion of the OPL (and hence also IDL)

family within LCPLs, and claims through practical examples that LCPGs are more suitable than OPG for specifying real programming languages. Section 8 compares our new family of languages with similar families introduced in previous literature. Finally, Section 9 draws some conclusions and outlines several goals for future research.

2. Preliminaries

For terms not defined here, we refer to any textbook on formal languages, e.g. [16]. The *terminal* alphabet is denoted by Σ; it includes the letter # used as start and end of text. Let Δ be an alphabet disjoint from Σ. A string $\beta \in (\Sigma \cup \Delta)^* \, \Sigma \, (\Sigma \cup \Delta)^* \setminus (\Sigma \cup \Delta)^* \, \Delta\Delta \, (\Sigma \cup \Delta)^*$ is in *operator form*; in words, β contains at least one terminal and does not contain adjacent symbols from Δ. OF(Δ) denotes the set of all operator form strings over $\Sigma \cup \Delta$.

The following *naming conventions* are adopted for letters and strings, unless otherwise specified: lowercase Latin letters $a, b, \ldots$ denote terminal characters; uppercase Latin letters $A, B, \ldots$ denote characters in Δ; lowercase Latin letters $x, y, z \ldots$ denote terminal strings; and Greek lowercase letters $\alpha, \ldots, \omega$ denote strings over $\Sigma \cup \Delta$.

Within the preceding convention, symbols in ***bold*** denote strings over an alphabet that includes, as extra symbols, the square brackets, e.g. $\boldsymbol{x} \in (\Sigma \cup \{[,]\})^*, \boldsymbol{\alpha} \in (\Sigma \cup \Delta \cup \{ [,] \})^*$.

We introduce the following short notation for frequently used operations based on alphabetic projections:

- for erasing all nonterminal symbols in a string $\boldsymbol{\alpha}$, we write $\widehat{\boldsymbol{\alpha}}$;
- for erasing all square brackets, we write $\widetilde{\boldsymbol{\alpha}}$;
- moreover, $\boldsymbol{\alpha} \mathrel{\hat{=}} \boldsymbol{\beta}$ stands for $\widehat{\boldsymbol{\alpha}} = \widehat{\boldsymbol{\beta}}$ and $\boldsymbol{\alpha} \mathrel{\tilde{=}} \boldsymbol{\beta}$ stands for $\widetilde{\boldsymbol{\alpha}} = \widetilde{\boldsymbol{\beta}}$.

A *context-free grammar* is a 4-tuple $G = (V_N, \Sigma, P, S)$, where V_N is the nonterminal alphabet, P the set of rules, and $S \subseteq V_N$ is the set of axioms. V denotes the set $V_N \cup \Sigma$. For a rule $A \to \alpha \in P$, $A \in V_N$ is the left-hand side (l.h.s.) and $\alpha \in V^*$ is the right-hand side (r.h.s.).

Let H be a new symbol, $H \notin V$, and $\sigma : V \to \{H\}$ be the homomorphism that maps every nonterminal to H: for every $X \in V_N, \sigma(X) = H$, otherwise $\sigma(a) = a$. The *stencil* of a rule $A \to \alpha$ is the rule $H \to \sigma(\alpha)$.

The *derivation relation* for a grammar G is denoted as usual by $\Rightarrow_G$ and its reflexive and transitive closure by $\overset{*}{\Rightarrow}_G$. A *sentential form* generated by G is any string $\#\alpha\# \in V^*$ such that $T \overset{*}{\Rightarrow}_G \alpha$ with $T \in S$, and the language generated by G is the set $L(G)$ of strings $x \in \Sigma^*$ such that $\#x\#$ is a sentential form.

A grammar is *invertible* if any two rules differ in their r.h.s. A grammar is an *operator grammar* (OG) if all r.h.s.'s are in the operator form OF(V_N); clearly, every sentential form of an OP grammar is in OF(V_N). Any context-free grammar that does not generate ε admits an equivalent OG (Theorem 4.8.1 of [16]). In this paper we deal only with OG, and assume them to be *reduced*, i.e., such that every rule is used in at least one derivation of a string belonging to its language.

For a context-free grammar G, the associated *parenthesis grammar*, denoted by $[G]$, is obtained by bracketing with '[' and ']' each r.h.s. of a rule of G. A grammar G is *structurally ambiguous* if there exists $\boldsymbol{x_1} \neq \boldsymbol{x_2} \in L([G])$ such that $\boldsymbol{x_1} \doteq \boldsymbol{x_2}$. Two grammars G, G' are *structurally equivalent* if $L([G]) = L([G'])$.

3. Chain-driven automata

In this section, we present the core formalism of this paper, i.e., the chain-driven automaton (CDA), that can be seen as an abstract parser for context-free languages. Unlike traditional parsers which operate left-to-right and build leftmost grammar derivations or reverse-rightmost ones, CDAs may proceed bottom-up starting from any position of the input string and building (in the reverse order) any derivation thereof: a CDA repeatedly and nondeterministically looks inside the input string for a grammar's r.h.s. and proceeds with a reduction.

As stated in the introduction such type of abstract parser is particularly well-suited to support parallel implementation: these automata, in fact, have no memory of the portion of string at the left (and right) of their current position; thus, it is easy to realize parallel parsers that consist of several "instances" of such automata. We have already proved in [3] for a less powerful class of languages, that this approach becomes extremely effective when such a type of bottom-up parsing can be done deterministically.

Next we formally define chain-driven automata and illustrate them by means of a few examples, then we prove the equivalence between chain-driven automata and context-free grammars.

The key driver in the search for a string to be reduced is the concept of chain – therefrom the name of the automaton. In accordance with the general philosophy of IDLs and OPLs, where the parsing actions by the recognizing automata are determined exclusively on the basis of terminal characters, the chains driving our automata contain only terminal characters: intuitively, a chain is the terminal projection of a string, enclosed within a suitable context, candidate to be reduced by the automaton.

Definition 1. *A* chain *is a triple $a\langle y\rangle b$ with $a, b \in \Sigma$ and $y \in \Sigma^+$; (a, b) is the* context *and y the* body *of the chain.*

A CDA works by reducing the input string through a sequence of reductions driven by a given set of chains; the automaton finds a given chain within the input string and replaces its body with a state; then the mechanism is applied recursively to the obtained string. Hence during the reduction steps the input string is shortened and simultaneously enriched by the computed states; chains being defined over the input alphabet, the portion of the input substring to be reduced is detected depending on terminal symbols only; enriching states are used then to (nondeterministically) determine which state will replace the detected substring but do not affect the choice of the chain on which to operate.

Definition 2. *A* chain-driven automaton *(CDA) $\mathcal{A}$ is a tuple (Σ, Q, δ, F) where*

- Σ *is the input alphabet;*

- Q *is a finite set of states;*
- $\delta : \Sigma \times \mathrm{OF}(Q) \times \Sigma \to \mathcal{P}(Q)$ *is the* reduce *function;*
- $F \subseteq Q$ *is the set of final states.*

If $\delta(a, \gamma, b) \neq \emptyset$ *implies* $|\delta(a, \gamma, b)| = 1$*, for all* $a, b \in \Sigma, \gamma \in \mathrm{OF}(Q)$*, then* $\mathcal{A}$ *is called* single valued*.*
For expressiveness, we also say that automaton $\mathcal{A}$ *is driven by the set of* chains $C(\mathcal{A}) := \{a\langle\widehat{\gamma}\rangle b \mid \delta(a, \gamma, b) \neq \emptyset\}$.

A *configuration* of the automaton is a string $\kappa \in \mathrm{OF}(Q)$. The *initial* configuration on input $x \in \Sigma^*$ is defined as $\#x\#$; a configuration $\#q\#$ with $q \in F$ is called an *accepting* configuration.

The elementary operation of the automaton, named *reduction move* and denoted by $\overset{a\langle y\rangle b}{\longmapsto}$, where $a\langle y\rangle b$ is a chain in $C(\mathcal{A})$, is defined for two configurations κ_1, κ_2, as:

$$\kappa_1 \overset{a\langle y\rangle b}{\longmapsto} \kappa_2 \text{ if, and only if, } \kappa_1 = \alpha\, a\gamma b\, \beta,\ \kappa_2 = \alpha\, aqb\, \beta, \text{ where } \widehat{\gamma} = y \text{ and } q \in \delta(a, \gamma, b).$$

We say that the string γ is *reduced* in such move. When not relevant, we omit the chain and write simply $\kappa_1 \longmapsto \kappa_2$. In words, a move deletes a substring in $\mathrm{OF}(Q)$ (corresponding to the body y of the chain, possibly enriched with states in Q) and replaces it with a state.

A *computation* is a sequence of reduction moves $\kappa_0 \overset{c_1}{\longmapsto} \kappa_1 \overset{c_2}{\longmapsto} \cdots \overset{c_n}{\longmapsto} \kappa_n$ where κ_i are configurations and c_i are chains. As usual $\overset{*}{\longmapsto}$ denotes the reflexive and transitive closure of $\longmapsto$. The *language accepted* by the automaton is defined as $L(\mathcal{A}) = \{x \in \Sigma^* \mid \#x\# \overset{*}{\longmapsto} \#q\# \text{ with } q \in F\}$.

In the sequel we assume, without loss of generality, that, for all chain-driven automata considered, every chain in $C(\mathcal{A})$ is used in some accepting computation.

Example 1. *Consider the language* $L_{ab} = \{a^n b^n \mid n \geq 1\}$*.* L_{ab} *can be recognized by the CDA* $\mathcal{A}_{ab}$ *with one state q, the set of chains*

$$\{\#\langle ab\rangle\#,\ a\langle ab\rangle b\},$$

and the reduction function defined by setting

$$\delta(\#, ab, \#) = \delta(\#, aqb, \#) = \delta(a, ab, b) = \delta(a, aqb, b) = \{q\}.$$

The (only) accepting computation on input string a^3b^3 *is the following:*

$$\#aaabbb\# \overset{a\langle ab\rangle b}{\longmapsto} \#aaqbb\# \overset{a\langle ab\rangle b}{\longmapsto} \#aqb\# \overset{\#\langle ab\rangle\#}{\longmapsto} \#q\#.$$

Similarly, the language $L_{abb} = \{a^n b^{2n} \mid n \geq 1\}$ *can be recognized by a chain driven automaton* $\mathcal{A}_{abb}$ *by simply changing the set of chains to* $\{\#\langle abb\rangle\#,\ a\langle abb\rangle b\}$ *and modifying the reduction function accordingly.*

Then, the language $L_{ab} \cup L_{abb}$, is recognized by the "union" of automata $\mathcal{A}_{ab}$ and $\mathcal{A}_{abb}$ which is built in a fairly natural way, i.e., by using the union of the two chain sets and two states $\{q_1, q_2\}$; the reduction function is also naturally defined as follows:

$$\begin{aligned} \delta(a, ab, b) = \delta(a, aq_1b, b) = \delta(\#, ab, \#) = \delta(\#, aq_1b, \#) &= \{q_1\} \\ \delta(a, abb, b) = \delta(a, aq_2bb, b) = \delta(\#, abb, \#) = \delta(\#, aq_2bb, \#) &= \{q_2\}. \end{aligned}$$

Example 2. *Consider now the language of arithmetic expressions on $\{e, +, *\}$ with the obvious meaning of symbols. A CDA $\mathcal{A}$ recognizing such expressions is defined as follows: $C(\mathcal{A})$ contains chains $\#\langle+\rangle\#$, $\#\langle+\rangle+$, $\#\langle*\rangle\#$, $\#\langle*\rangle+$, $+\langle*\rangle\#$, $+\langle*\rangle+$, $+\langle*\rangle*$, $\#\langle*\rangle*$, and all chains $a\langle e\rangle b$ with $a, b \in \{\#, *, +\}$; $Q = \{q_e, q_+, q_*\}$, $F = Q$, and δ is given in the following table, where the first column collects the contexts (a, b), and the second row specifies the strings in* $\mathrm{OF}(Q)$ *gathered according to their projection.*

context	$\widehat{\gamma} = *$	$\widehat{\gamma} = +$	$\widehat{\gamma} = e$
	$q_e{*}q_e \;\; q_*{*}q_e$	$q_e{+}q_e \;\; q_*{+}q_e \;\; q_+{+}q_e$ $q_e{+}q_* \;\; q_*{+}q_* \;\; q_+{+}q_*$	e
$(\#,\#)\;\; (\#,+)$	q_*	q_+	q_e
$(+,\#)\;\; (+,+)\;\; (+,*)\;\; (\#,*)$	q_*		q_e
$(*,\#)\;\; (*,*)\;\; (*,+)$			q_e

(1)

For instance, the bottom row specifies, among others, $\delta(*, e, +) = q_e$.

Two accepting computations for $e + e * e$ are:

$$\#e{+}e{*}e\# \overset{\#\langle e\rangle+}{\vdash} \#q_e{+}e{*}e\# \overset{+\langle e\rangle*}{\vdash} \#q_e{+}q_e{*}e\# \overset{*\langle e\rangle\#}{\vdash} \#q_e{+}q_e{*}q_e\# \overset{+\langle*\rangle\#}{\vdash} \#q_e{+}q_*\# \overset{\#\langle+\rangle\#}{\vdash} \#q_+\#;$$

$$\#e{+}e{*}e\# \overset{+\langle e\rangle*}{\vdash} \#e{+}q_e{*}e\# \overset{*\langle e\rangle\#}{\vdash} \#e{+}q_e{*}q_e\# \overset{\#\langle e\rangle+}{\vdash} \#q_e{+}q_e{*}q_e\# \overset{+\langle*\rangle\#}{\vdash} \#q_e{+}q_*\# \overset{\#\langle+\rangle\#}{\vdash} \#q_+\#.$$

Both computations are forced to reduce the $*$ operator before the $+$, in agreement with the usual syntactic structure of arithmetic expressions that gives precedence to $*$ over $+$. The intuitive reason for this behavior is to be found in the fact that a body with $+$ cannot be reduced in a context containing $*$; on the other hand, a body with $*$ can be reduced in a context with $+$.

Intuitively, the automata of the above examples provide a unique syntactic structure to their accepted input strings, despite the fact that several computations may recognize the same string. In general, however, different computations may associate different structures with the same accepted string.

As for the classical parenthesis grammars, therefore, we next formalize the notion of structure and structural ambiguity by introducing parenthesis automata.

Definition 3. *For a CDA $\mathcal{A} = (\Sigma, Q, \delta, F)$, the associated* parenthesis CDA *is $[\mathcal{A}] = (\Sigma \cup \{[,]\}, Q, \delta', F)$ where δ' is defined by setting $\delta'(a, [\gamma], b) = \delta(a, \gamma, b)$. A CDA $\mathcal{A}$ is* structurally ambiguous *if $L([\mathcal{A}])$ contains two strings $\boldsymbol{x_1} \neq \boldsymbol{x_2}$ such that $\boldsymbol{x_1} \,\tilde{=}\, \boldsymbol{x_2}$.*

For instance, consider a variant of the automaton of Example 2, where in (1) the context $(+, \#)$ is moved up one row, and $q_e + q_+$ is added to the column for $\widehat{\gamma} = +$. This automaton performs two structurally different computations for the same input:

$$\#e + e + e\# \overset{*}{\vdash} \#q_e + q_e + q_e\# \overset{\#\langle+\rangle+}{\vdash} \#q_+ + q_e\# \overset{\#\langle+\rangle\#}{\vdash} \#q_+\#,$$

which on the parenthesis CDA corresponds to the computation $\#[[[e]+[e]]+[e]]\# \overset{*}{\longmapsto} \#q_+\#$, and

$$\#e+e+e\# \overset{*}{\longmapsto} \#q_e+q_e+q_e\# \overset{+\langle+\rangle\#}{\longmapsto} \#q_e+q_+\# \overset{\#\langle+\rangle\#}{\longmapsto} \#q_+\#,$$

where the structure is $[[e]+[[e]+[e]]]$.

Not surprisingly, we are going to see that CDAs recognize exactly context-free languages, acting as parsers for their grammars: states of the automaton correspond to nonterminals of the grammar; any string reduced by the automaton corresponds to the r.h.s. of some grammar rules, and the set of states computed by the reduction function corresponds to the set of the l.h.s., i.e., nonterminals of those rules.

Definition 4. *The chain $a\langle y\rangle b$ is a* grammatical chain *associated with a context-free grammar G if there exists a derivation*

$$\#T\# \underset{G}{\overset{*}{\Longrightarrow}} \alpha\, aAb\, \beta \underset{G}{\Longrightarrow} \alpha\, a\gamma b\, \beta$$

with $T \in S$, $\widehat{\gamma} = y$. The set of grammatical chains associated with G is written C_G.

Example 3. *Consider the grammar with one nonterminal and axiom T and rules $T \to aTb \mid ab$; the grammar generates the language L_{ab} defined in Example 1. The grammatical chains are $\#\langle ab\rangle\#$ and $a\langle ab\rangle b$. The latter is defined, e.g., by the derivation $\#T\# \Rightarrow \#aTb\# \Rightarrow \#aabb\#$.*

Theorem 1. *Chain-driven automata recognize the family of context-free languages.*

Proof. We prove that the language recognized by any CDA can be generated by an OG, and vice versa. We first need the concept of a labeled transition system (LTS), which is a triple (S, Λ, τ) where S is an infinite set of LTS states, Λ is a set of labels, and τ is a set of labeled state transitions (i.e., $\tau \subseteq S \times \Lambda \times S$).

Notice that both grammars and chain-driven automata can be seen as LTS. Formally, a grammar can be seen as the LTS $(\mathrm{OF}(V_N), C, \Longleftarrow)$ where the LTS states are all strings in operator forms, the labels are all the possible chains over Σ, and $\Longleftarrow$ is defined by setting $\alpha a\gamma b\beta \overset{c}{\Longleftarrow} \alpha aAb\beta$ where $c = a\langle y\rangle b$, $A \to \gamma$ is a rule of G and $\widehat{\gamma} = y$. A CDA can be seen as the LTS $(\mathrm{OF}(Q), C, \longmapsto)$ where labels are the chains that drive the automaton, and $\longmapsto$ is the relation defined by the reduction moves.

Let $G = (V_N, \Sigma, P, S)$, and C_G be the set of grammatical chains associated with G. Define the CDA $\mathcal{A}_G = (\Sigma, Q, \delta, F)$ where: $Q = V_N$; $F = S$ is the set of axioms; δ is defined by setting $B \in \delta(a, \gamma, b)$ for each rule $B \to \gamma$ such that $a\langle\hat{\gamma}\rangle b \in C_G$. Notice that $C(\mathcal{A}_G) = C_G$. Both G and $\mathcal{A}_G$ define the same LTS. In particular, this means that the derivations $\#T\# \overset{*}{\Rightarrow} \#x\#$ of G with $T \in S$ are in bijection with the computations $\#x\# \overset{*}{\longmapsto} \#T\#$ and this implies that $L(\mathcal{A}_G) = L(G)$.

Conversely, let $\mathcal{A} = (\Sigma, Q, \delta, F)$. Define the grammar $G_{\mathcal{A}} = (V_N, \Sigma, P, S)$ where: $V_N = \Sigma \times Q \times \Sigma$, $S = \{(\#, q, \#) \mid q \in F\}$, and P is the set of rules $(a_0, q, a_{n+1}) \to \gamma$ where $q \in \delta(a_0, \gamma, a_{n+1})$. Notice that the set of $G_{\mathcal{A}}$'s chains coincides with $C(\mathcal{A})$. Both $\mathcal{A}$ and $G_{\mathcal{A}}$ define the same LTS, except that for $\mathcal{A}$ the LTS states are configurations

$a_0q_0a_1q_1a_2 \dots a_nq_na_{n+1}$ (any q_i may be missing), whereas for $G_{\mathcal{A}}$ the LTS states are written in the form $a_0(a_0, q_0, a_1)a_1(a_1, q_1, a_2)a_2 \dots a_{n-1}(a_n, q_n, a_{n+1})a_{n+1}$. In particular, this means that the computations $\#x\# \overset{*}{\longmapsto} \#q\#$ of $\mathcal{A}$ with $q \in F$ are in bijection with the derivations $\#(\#, q, \#)\# \overset{*}{\Rightarrow} \#x\#$ of G and this implies that $L(G_{\mathcal{A}}) = L(\mathcal{A})$. □

In summary, whereas traditional pushdown automata and context-free parsers always proceed left to right and produce a unique representation of the syntax trees associated with the input string, our chain-driven automata may nondeterministically produce any bottom-up possible traversal of the grammar's trees, as it is illustrated by the automaton of Example 2 and by its structurally ambiguous modification.

Clearly, $\mathcal{A}$ is structurally unambiguous if, and only if, the equivalent grammar $G_{\mathcal{A}}$ defined in the proof of Theorem 1 is structurally unambiguous, and vice versa. Also, $\mathcal{A}$ is single valued if, and only if, the equivalent grammar $G_{\mathcal{A}}$ is invertible, and vice versa.

4. Locally chain-parsable languages

In this section we introduce the key notion of *local chain parsability* (LCP) and define it as a property of both chain-driven automata and context-free grammars. Intuitively, a local chain-driven automaton, once it has found a chain while parsing a valid string, can reduce it with the certainty that the reduction is part of an accepting computation; thus, the risk of roll-back, typical of nondeterministic parsing, is avoided for these automata. To be more precise, a certain level of nondeterminism remains embedded in the reduce function when the chain-driven automaton is not single valued, but we will see at the end of the section that this choice too can be removed from the automaton definition.

Definition 5. *A CDA is a* local chain parser *(LCPA) if, for every chain $a\langle y\rangle b$, the following condition holds: if $\widehat{\gamma} = y$, then every computation $\alpha\, a\gamma b\, \beta \overset{*}{\longmapsto} \#q_F\#$ with $q_F \in F$ can be written as*

$$\alpha\, a\gamma b\, \beta \overset{*}{\longmapsto} \alpha'\, a\gamma b\, \beta' \longmapsto \alpha'\, aqb\, \beta' \overset{*}{\longmapsto} \#q_F\#$$

for some state q and strings α', β', such that $\alpha \overset{*}{\longmapsto} \alpha'$, $\beta \overset{*}{\longmapsto} \beta'$.

Informally, for a CDA to be an LCPA, we require that, if $a\langle y\rangle b$ is a chain, then every γ, with $\widehat{\gamma} = y$, appearing in a context (a, b), must be reduced with a single move.

Example 4. *Consider the languages and corresponding automata defined in Example 1. It is easy to verify that both $\mathcal{A}_{ab}$ and $\mathcal{A}_{abb}$ are local chain parsers.*

The automaton recognizing $L_{ab} \cup L_{abb}$, instead, is not an LCPA. For instance, consider the chain $a\langle abb\rangle b$ and the accepting computation (on an input in L_{ab})

$$\#aaabbb\# \overset{a\langle ab\rangle b}{\longmapsto} \#aaq_1bb\# \overset{a\langle ab\rangle b}{\longmapsto} \#aq_1b\# \overset{\#\langle ab\rangle\#}{\longmapsto} \#q_1\#$$

We can identify portions $\alpha = \#a$, $\gamma = abb$, and $\beta = \#$ that do not satisfy the condition of Definition 5, since γ is not reduced in a single move: first the prefix ab is reduced by applying $\delta(a, ab, b) \ni q_1$, and then the suffix b is reduced, together with another a, by applying $\delta(a, aq_1b, b) \ni q_1$.

Similarly, one can verify that the automaton described in Example 2 is an LCPA but its subsequent variant is not.

Example 5. *Consider now the language* $L_{aba} = \{a^n b a^n \mid n \geq 1\}$. L_{aba} *can be recognized by a chain driven automaton with one state* q*, the set of chains*

$$\{a\langle b\rangle a,\ \#\langle aa\rangle\#,\ a\langle aa\rangle a\},$$

and the reduction function defined by setting

$$\delta(a, b, a) = \delta(a, aqa, a) = \delta(\#, aqa, \#) = \{q\}.$$

Perhaps surprisingly, this automaton is not an LCPA. For instance, consider the chain $a\langle aa\rangle a$ *and the accepting computation*

$$\#aaabaaa\# \overset{a\langle b\rangle a}{\vdash\!\!\dashv} \underbrace{\#aa}_{\alpha}\, a\, \underbrace{qaa}_{\gamma}\, a\, \underbrace{\#}_{\beta} \overset{a\langle aa\rangle a}{\vdash\!\!\dashv} \#aaqaa\# \overset{a\langle aa\rangle a}{\vdash\!\!\dashv} \#aqa\# \overset{\#\langle aa\rangle\#}{\vdash\!\!\dashv} \#q\#.$$

The evidenced substrings α, γ*, and* β *do not satisfy the condition of Definition 5, since* γ *is not reduced in a single move: first the prefix* qa *is reduced, together with the preceding* a*, by applying* $\delta(a, aqa, a) \ni q$*, and then the suffix* a *is reduced, together with the* aq *to its left, by reapplying the same reduction rule.*

The previous example illustrates the fact that Definition 5 is based on a purely input-driven condition for local parsability. Indeed, the presence of state q would allow to locally recognize the right portion of string to reduce (aqa and not aaq, qaa, nor aa); however, this cannot be done considering only terminals. To stress this fact, in the definition we qualified our parser as "local" and driven by "chains" (which are composed only by terminals).

Thus, our Definition 5 is conceptually different from other formalizations of the intuitive notion of local parsability, in particular, Floyd's bounded-context parsing based on full r.h.s. including terminals and nonterminals [14], discussed in Section 8; the unavoidable loss in terms of generative power is, in our opinion, compensated by the more complete algebraic properties typical of input-driven languages as we will show in Section 6.

Definition 6. *A grammar is* locally chain parsable *(LCPG) if, for every grammatical chain* $a\langle y\rangle b$*, the following condition holds: if* $\widehat{\gamma} = y$*, then each derivation* $\#T\# \overset{*}{\Longrightarrow} \alpha a\gamma b\beta$ *with* $T \in S$ *consists of steps* $\#T\# \overset{*}{\Longrightarrow} \alpha' aAb\,\beta' \Longrightarrow \alpha' a\gamma b\,\beta' \overset{*}{\Longrightarrow} \alpha\, a\gamma b\,\beta$*, where* $\alpha' \overset{*}{\Longrightarrow} \alpha$*, and* $\beta' \overset{*}{\Longrightarrow} \beta$*. A language* L *is locally chain parsable (LCPL) if it is generated by an LCPG.*

In other terms, for a grammar to be an LCPG, we require what follows: for every γ appearing with context (a, b) in some derivation starting from $\#T\#$, γ has to be generated with a single rule $A \to \gamma$ and such a rule has to be applied to a string where the nonterminal A has (a, b) as context.

Example 6. *The following grammar G_1, which generates the same arithmetic expressions recognized by the CDA of Example 2, is an LCPG.*

$$\begin{array}{ll} E \to E + T \mid T * F \mid e & F \to e \\ T \to T * F \mid e & S = \{E, T, F\} \end{array}$$

The set of grammatical chains associated with this grammar is exactly $C(\mathcal{A})$ as defined in Example 2. Consider a derivation such as

$$\#E\# \Rightarrow \#E+T\# \Rightarrow \#E+T+T\# \Rightarrow \#E+T*F+T\# \overset{*}{\Rightarrow} \#e+e*F+e\# \Rightarrow \#e+e*e+e\#.$$

The result of any derivation step is such that each terminal character is enclosed within a context of a pair of terminals which univocally determines the stencil of the last step of the derivation that produced it, independently on the non-terminals involved in the derivation: e.g., every e can only be produced by a rule with stencil $H \to e$; the only $*$ in the context $(+,+)$ can only be produced through a rule with stencil $H \to H * H$; the first + is produced by the rule $E \to E + T$ in the context $(\#,+)$ but there is no way to produce the second + within any of the contexts $(+,\#)$, $(*,\#)$, $(*,e)$, and (e,e), by means of an immediate derivation with stencil $H \to H + H$.

Thus, a bottom-up parser can always decide which terminal part of any r.h.s. to reduce by only inspecting the terminal parts of any sentential form of length 3 plus its context: if it finds the terminal part $\widehat{\alpha}$ of a rule $A \to \alpha$ within a context where G can generate any β with $\beta \hat{=} \alpha$ through an immediate step of derivation $B \Rightarrow \beta$, then it can reduce the r.h.s. to the corresponding l.h.s. with the certainty that the same $\widehat{\alpha}$ cannot be obtained as part of a more complex derivation that does not produce it in a single step; notice also that the reduction is unique if G is invertible.

On the contrary, the following grammar G_2, generating only additive expressions, is not LCP.

$$\begin{array}{ll} X \to E + X \mid E + E & E \to e \\ Y \to Y + E \mid E + E & S = \{X, Y\} \end{array}$$

The associated grammatical chains are: $\#\langle+\rangle\#$, $\#\langle+\rangle+$, $\#\langle e\rangle+$, $+\langle e\rangle+$, $+\langle e\rangle\#$, $+\langle+\rangle\#$. For instance, chain $+\langle+\rangle\#$ is obtained by applying rule $X \to E + E$ in the last step of the derivation: $\#X\# \overset{*}{\Rightarrow} \#E + E + X\# \Rightarrow \#E + E + E + E\#$. But in the derivation:

$$\#Y\# \Rightarrow \#Y + E\# \Rightarrow \#Y + E + E\# \Rightarrow \#E + E + E + E\#$$

the substring $\gamma = E + E$ occurs in context $(+,\#)$ but it is not produced in any step of the derivation. Hence, G_2 is not locally parsable. As a consequence, a possible parser, after having reduced all terminals e to E, would be confronted with the sentential form $\#E + E + E + E\#$ and left without an indication whether to apply $X \to E + E$ reducing the last +, or to apply $Y \to E + E$ reducing the leftmost +.

Theorem 2. *A language is locally chain parsable if, and only if, it is recognized by a local chain parser.*

Proof. The statement is a consequence of the fact the both constructions in the proof of Theorem 1 preserve locality properties. □

Remark 1. *If $\mathcal{A}$ is a local chain parser, then it admits an equivalent single valued LCPA; this fact can be proved with the usual power set construction, since it preserves the LCP property. Analogously, if G is locally chain parsable, then it admits an equivalent invertible LCPG, since the construction that transforms a grammar into a structurally equivalent invertible one [16] also preserves the LCP property. These facts are not surprising, and can be seen as a direct generalization of the analogous McNaughton property for structured languages [21]. Notice the difference w.r.t. the case of DCFLs: whereas building a deterministic pushdown automaton from an LR grammar and conversely are fairly intricate constructions, the same constructions from chain-driven automata to CF grammars and conversely apply as well to LCPAs and LCPGs.*

5. Extended chains, conflicts and decidability of the LCP property

Both an LCPA and an LCPG assign a unique structure to each accepted string. To formalize this point we first introduce the notion of extended chain that generalizes Definition 1. To avoid clashes, we use different notations, so that chains and extended chains are disjoint.

Definition 7. Structured strings *are special well-parenthesized strings over $\Sigma \cup \{[,]\}$, defined recursively as follows:*

- *$y \in \Sigma^+$ are structured strings;*
- *if $a_i \in \Sigma$ and $\boldsymbol{y_i} = \varepsilon$ or $\boldsymbol{y_i} = [\boldsymbol{v_i}]$ for some structured strings $\boldsymbol{v_i}$, excluding that all $\boldsymbol{y_i}$ are empty, then $\boldsymbol{y_0} a_1 \boldsymbol{y_1} a_2 \dots a_n \boldsymbol{y_n}$ is a structured string.*

An extended chain *(briefly* xchain*) is a string $\#[\boldsymbol{y}]\#$ where $\boldsymbol{y}$ is a structured string which is called the* body *of the xchain.*

Any grammar/chain-driven automaton determines a set of xchains, which have an important role to assess the LCP property.

Definition 8. *Let $\mathcal{A}$ be a chain-driven automaton and G a grammar. An xchain $\#[\boldsymbol{y}]\#$ is an $\mathcal{A}$-*xchain*, or a G-*xchain*, if there exist $\boldsymbol{\gamma}$ such that $\widehat{\boldsymbol{\gamma}} = \boldsymbol{y}$ and, respectively,*

$$\#[\boldsymbol{\gamma}]\# \overset{*}{\underset{[\mathcal{A}]}{\vdash}} \#q_F\# \text{ with } q_F \in F \qquad \text{or} \qquad \#T\# \overset{*}{\underset{[G]}{\Longrightarrow}} \#[\boldsymbol{\gamma}]\# \text{ with } T \in S.$$

The sets of $\mathcal{A}$-xchains and G-xchains are denoted respectively by $\mathcal{X}_{\mathcal{A}}$ and $\mathcal{X}_G$.

Remark 2. *The proof of Theorem 1 defines the CDA equivalent to a given grammar, and, conversely, the grammar equivalent to a given CDA. In both cases, the CDA and the grammar have identical xchains, i.e., $\mathcal{X}_{\mathcal{A}_G} = \mathcal{X}_G$ and $\mathcal{X}_{G_{\mathcal{A}}} = \mathcal{X}_{\mathcal{A}}$.*

Example 7. *Consider grammar G_1 of Example 6. The sentential form $\#[E+[[e]*[e]]]\#$ is generated by parenthesis grammar $[G_1]$ with the following derivation*

$$\#E\# \Rightarrow \#[E + T]\# \Rightarrow \#[E + [T * F]]\# \Rightarrow \#[E + [T * [e]]]\# \Rightarrow \#[E + [[e] * [e]]]\#$$

hence #[+[[e] ∗ [e]]]# *is a* G_1*-xchain. Other* G_1*-xchains are* #[[+] + [∗[e]]]# , #[[[e] ∗ [e]] ∗ [e]]# , #[+[[∗[e]]∗]]#.
Similarly, let $\mathcal{A}$ *be the CDA of Example 2; both computations for the string* $e + e * e$ *presented in the same example define the xchain* #[[e] + [[e] ∗ [e]]]#. *One can easily guess that* $\mathcal{A}$*-xchains are the same as* G_1*'s ones. Notice also that both* G_1 *and* $\mathcal{A}$ *are such that, for each valid string* y*, there is only one* $G_1/\mathcal{A}$*-xchain* $\mathbf{y}$ *such that* $\widetilde{\mathbf{y}} = y$.

The next definition introduces the concept of conflict between an xchain and a chain. Intuitively, an xchain c conflicts with a chain $s = a\langle y\rangle b$ if $\widetilde{c}$ contains the string ayb but such occurrence of y does not correspond to the body of a "subchain" that occurs in s.

Definition 9. *An xchain* conflicts *with a chain* $a\langle y\rangle b$ *if, and only if, it can be decomposed as* $\mathbf{x}a\mathbf{y}b\mathbf{z}$ *where* $\widetilde{\mathbf{y}} = y$ *and* $\mathbf{y} \notin [^+y]^+$. *A set* $\mathcal{X}$ *of xchains and a set* C *of chains are* conflictual *if there is an xchain in* $\mathcal{X}$ *that conflicts with some chain in* C.

Example 8. *The xchain* #[+[+[+[+]]]]# *conflicts with the chain* #⟨+⟩+ *since the prefix* #[+[+ *of the xchain projects onto* # + +, *but the first occurrence of symbol* + *in the xchain is not bracketed; formally, the definition is satisfied with* $\mathbf{x} = \varepsilon$, $\mathbf{y} =$ [+[, *and* $\mathbf{z} =$ [+[+]]]#. *This implies that the sets* $\mathcal{X}_{G_2}$ *and* C_{G_2} *associated with the grammar* G_2 *defined in Example 6 are conflictual.*

On the contrary, with a little patience it can be verified that, for the grammar G_1 *in the same example, the set of* G_1*-xchains does not exhibit any conflict with* C_{G_1}. *For instance,* #[[[+]+]+]+]# *does not conflict with* #⟨+⟩+ *since, whenever* # + + *occurs in the xchain (only once, as a prefix), the first occurrence of symbol* + *is bracketed.*

Also, G_1 *is LCP, whereas* G_2 *is not. These remarks lead to the main property stated in Theorem 4.*

The next lemma, which easily follows from Definition 9, illustrates all the cases where a conflict may occur.

Lemma 1. *If an xchain* c *conflicts with a chain* $a\langle y\rangle b$*, then there are four possibilities, represented in the table:*

Conflict type	$\mathbf{x}$	a	$\mathbf{y}$	b	$\mathbf{z}$	*conditions*
Left conflict	$\mathbf{x_1}[x_2$	a	y_1] $\mathbf{y_2}$	b	$\mathbf{z}$	
Right conflict	$\mathbf{x}$	a	$\mathbf{y_1}$ [y_2	b	z_1] $\mathbf{z_2}$	
Inner conflict	$\mathbf{x}$	a	$\mathbf{y_1}[y_2]\mathbf{y_3}$	b	$\mathbf{z}$	$\widetilde{\mathbf{y_1}\mathbf{y_3}} \neq \varepsilon$
Outer conflict	$\mathbf{x_1}[x_2$	a	y	b	z_1] $\mathbf{z_2}$	

These cases are not mutually exclusive: an xchain may exhibit more that one conflict with the same chain.

We next show that the property of having nonconflictual $\mathcal{X}_G$ and C_G is decidable for any grammar G (and is computed by an automatic tool[1]).

[1] `https://github.com/bzoto/chainsaw`.

Theorem 3. *For every grammar G and automaton $\mathcal{A}$, it is decidable whether the pairs of sets, respectively, $\mathcal{X}_G$, C_G, and $\mathcal{X}_{\mathcal{A}}$, $C(\mathcal{A})$ are nonconflictual.*

Proof. Let G be (V_N, Σ, P, S). We first introduce a grammar $G' = (V_N \cup \{T'\}, \Sigma \cup \{[,]\}, P', \{T'\})$, such that $T' \notin V_N$ and

$$P' = \left\{ A \to [\alpha'] \;\middle|\; \begin{array}{l} A \to \alpha \in P, \text{ where } \alpha' = \alpha \text{ or } \alpha' \text{ is obtained from } \alpha \\ \text{by erasing some (or every) nonterminals} \end{array} \right\} \cup \{T' \to \#[T]\# \mid T \in S\}.$$

It is easy to see that G' defines the language of all the G-xchains, because its rules have the same structure as those of G and are marked by explicit brackets. Nonterminals have to be erased to take into account that G-xchains are defined on sentential forms, by discarding all nonterminals.
For a grammatical chain $c = a\langle y \rangle b$, we define the regular language

$$R(c) = (\Sigma \cup \{[,]\})^* \cdot a \cdot \{\boldsymbol{y} \in \neg([^+ \cdot y \cdot]^+) \mid \widetilde{\boldsymbol{y}} = y\} \cdot b \cdot (\Sigma \cup \{[,]\})^*$$

that is the language of all the words having a substring which conflicts with c. Clearly, $R' := \bigcup_{c \in C_G} R(c)$ is also a regular language. G is conflictual if, and only if, $L(G') \cap R' \neq \emptyset$; since $L(G') \cap R'$ is CF, its emptiness problem is decidable.

The statement for CDAs follows from Theorem 1 and Remark 2. □

Theorem 4. *A chain-driven automaton $\mathcal{A}$ is a local chain parser (respectively, a grammar G is locally chain parsable) if, and only if, $\mathcal{X}_{\mathcal{A}}$ and $C(\mathcal{A})$ (respectively, $\mathcal{X}_G$ and C_G) are nonconflictual.*

Proof. We prove the statement concerning automata; the analogous statement for grammars follows from Theorem 2 and Remark 2.

First, we prove that if $\mathcal{X}_{\mathcal{A}}$ and $C(\mathcal{A})$ are not conflictual, then $\mathcal{A}$ is a local chain parser. By contradiction we assume that $\mathcal{A}$ is not a local parser. Then, there exist a chain $a\langle y \rangle b \in C(\mathcal{A})$, a string $\boldsymbol{\gamma}$ such that y is the projection of $\boldsymbol{\gamma}$ onto Σ, a final state q_F, and a computation

$$\boldsymbol{\alpha}\, a\boldsymbol{\gamma} b\, \boldsymbol{\beta} \overset{*}{\underset{[\mathcal{A}]}{\vdash}} \#q_F\#$$

where $\boldsymbol{\gamma}$ is not reduced in a move driven by $a\langle y \rangle b$. Then we show that there exists an $\mathcal{A}$-xchain conflicting with $a\langle y \rangle b$.

We may assume that the first move of the computation involves $\boldsymbol{\gamma}$ (otherwise we can ignore the previous moves not involving $\boldsymbol{\gamma}$ and consider the remaining part of the computation). Then there are only four possibilities.

1. The computation can be decomposed as

$$\boldsymbol{\alpha_1}\, [\alpha_2 a \gamma_1]\, \boldsymbol{\gamma_2} b \boldsymbol{\beta} \underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-} \boldsymbol{\alpha_1}\, q\, \boldsymbol{\gamma_2} b \boldsymbol{\beta} \overset{*}{\underset{[\mathcal{A}]}{\vdash}} \#q_F\#$$

   where $\boldsymbol{\gamma} = \gamma_1]\boldsymbol{\gamma_2}$ and $\boldsymbol{\alpha} = \boldsymbol{\alpha_1}[\alpha_2$. Then $\widehat{\boldsymbol{\alpha_1}}[\widehat{\alpha_2} a \widehat{\gamma_1}]\widehat{\boldsymbol{\gamma_2}} b \widehat{\boldsymbol{\beta}}$ is an $\mathcal{A}$-xchain that conflicts with $a\langle y \rangle b$: it is a left conflict, since Lemma 1 is satisfied with $\boldsymbol{y} = \widehat{\gamma_1}]\widehat{\boldsymbol{\gamma_2}}$.

2. The computation can be symmetrically decomposed with $\boldsymbol{\gamma} = \boldsymbol{\gamma_1}[\gamma_2$ and $\boldsymbol{\beta} = \beta_1]\,\boldsymbol{\beta_2}$. Then one obtains a right conflict.

3. The computation can be decomposed as

$$\boldsymbol{\alpha} a \boldsymbol{\gamma_1}\ [\gamma_2]\ \boldsymbol{\gamma_3} b \boldsymbol{\beta} \underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-} \boldsymbol{\alpha} a \boldsymbol{\gamma_1}\ q\ \boldsymbol{\gamma_3} b \boldsymbol{\beta} \overset{*}{\underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-}} \#q_F\#$$

where $\boldsymbol{\gamma} = \boldsymbol{\gamma_1}[\gamma_2]\boldsymbol{\gamma_3}$, with $\boldsymbol{\gamma_1}\boldsymbol{\gamma_3} \tilde{\neq} \varepsilon$ and $\gamma_2 \neq \varepsilon$. Then $\widehat{\boldsymbol{\alpha}} a \widehat{\boldsymbol{\gamma_1}}[\widehat{\gamma_2}]\widehat{\boldsymbol{\gamma_3}} b \widehat{\boldsymbol{\beta}}$ is an $\mathcal{A}$-xchain that conflicts with $a\langle y\rangle b$: it is an inner conflict, since Lemma 1 is satisfied with $\boldsymbol{y} = \widehat{\boldsymbol{\gamma_1}}[\widehat{\gamma_2}]\widehat{\boldsymbol{\gamma_3}}$.

4. The computation can be decomposed as

$$\boldsymbol{\alpha_1}\ [\alpha_2 a \gamma b \beta_1]\ \boldsymbol{\beta_2} \underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-} \boldsymbol{\alpha_1}\ q\ \boldsymbol{\beta_2} \overset{*}{\underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-}} \#q_F\#$$

where $\boldsymbol{\alpha} = \boldsymbol{\alpha_1}[\alpha_2, \boldsymbol{\beta} = \beta_1]\,\boldsymbol{\beta_2}$. Then $\widehat{\boldsymbol{\alpha_1}}[\widehat{\alpha_2} a \widehat{\gamma} b \widehat{\beta_1}]\widehat{\boldsymbol{\beta_2}}$ is an $\mathcal{A}$-xchain that conflicts with $a\langle y\rangle b$: it is an outer conflict, since Lemma 1 is satisfied with $\boldsymbol{y} = \widehat{\gamma}$.

Vice versa, we prove that if $\mathcal{A}$ is a local chain parser, then $\mathcal{X}_{\mathcal{A}}$ and $C(\mathcal{A})$ are nonconflictual. Again, we reason by contradiction and assume that there exists an $\mathcal{A}$-xchain that conflicts with a chain $a\langle y\rangle b$. The xchain can be decomposed as $\boldsymbol{x} a \boldsymbol{y} b \boldsymbol{z}$ with $\widetilde{\boldsymbol{y}} = y$ and one of the four cases in Lemma 1 holds.

One can verify that, in each case, there exists a computation

$$\alpha\ a\gamma b\ \beta \overset{*}{\underset{\mathcal{A}}{\vdash\!\!\!-\!\!\!-}} \#q_F\#$$

with q_F final and $\widehat{\gamma} = y$, where γ is not reduced in a single move, thus contradicting the hypothesis. We only discuss one case since the others are similar. Consider the case of left conflict, where the conflicting xchain can be decomposed as $\boldsymbol{x_1}[x_2 a y_1]\boldsymbol{y_2} b \boldsymbol{z}$. Then by definition of $\mathcal{A}$-xchain, there exist $\boldsymbol{\alpha_1} \hat{=} \boldsymbol{x_1}$, $\alpha_2 \hat{=} x_2$, $\gamma_1 \hat{=} y_1$ $\boldsymbol{\gamma_2} \hat{=} \boldsymbol{y_2}$, and $\boldsymbol{\beta} \hat{=} \boldsymbol{z}$ such that

$$\boldsymbol{\alpha_1}\ [\alpha_2 a \gamma_1]\ \boldsymbol{\gamma_2} b \boldsymbol{\beta} \underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-} \boldsymbol{\alpha_1}\ q\ \boldsymbol{\gamma_2} b \boldsymbol{\beta} \overset{*}{\underset{[\mathcal{A}]}{\vdash\!\!\!-\!\!\!-}} \#q_F\#$$

is a computation of $[\mathcal{A}]$; in the corresponding computation of $\mathcal{A}$, the substring $\gamma = \gamma_1\widetilde{\boldsymbol{\gamma_2}}$ is not reduced in any move driven by $a\langle y\rangle b$ (notice that $\widetilde{\boldsymbol{\gamma_2}}$ cannot be deleted independently of q also when $\gamma_1 = \varepsilon$). □

Theorems 4 and 3 imply the following result.

Corollary 1. *It is decidable whether a grammar is locally chain parsable and whether a chain-driven automaton is a local chain parser.*

6. Basic properties of local chain parsable languages

In this section we prove the strict inclusion of LCPLs within the DCFL family, and we investigate their closure under the fundamental language operations; we point out

similarities and differences w.r.t. other input-driven language families: we show that several closure properties enjoyed by other families do not extend to LCPLs; nevertheless, under suitable hypotheses of chain compatibility, Boolean operations between LCPLs preserve local parsability, so that e.g. the containment problem is decidable for pair of languages satisfying such hypothesis. As a preliminary step, we briefly discuss structural non-ambiguity.

Remark 3. *Every LCPG and LCPA assigns to each valid string x a unique structure represented by an xchain $\#[\boldsymbol{x}]\#$ where $\widetilde{\boldsymbol{x}} = x$. In other words, LCPGs and LCPAs are structurally unambiguous. Both properties can be proved similarly reasoning by contradiction and building an xchain that necessarily conflicts with a chain.*

Moreover, if an LCPA is single valued (or, equivalently, an LCPG is invertible) then it is unambiguous, in that also the labels of the syntactic trees are uniquely determined.

Finally, independent moves of an LCPA (i.e., moves that reduce non-overlapping substrings of the input) can be applied in any order, without altering the result.

Example 9. *As a counterexample, grammar G_2 of Example 6 is not an LCPG; in fact it produces non-homomorphic syntactic trees for the same sentence $\#e + e + e\#$, namely those corresponding to xchains*

$$\#[[e] + [[e] + [[e] + [[e] + [e]]]]]\# \qquad \textit{and} \qquad \#[[[[[e] + [e]] + [e]] + [e]] + [e]]\#.$$

Next, we move to a hardly surprising inclusion property, where *right-to-left deterministic context-free languages* are those defined by deterministic pushdown automata that work reading their input going from right to left. Then, we show that the LCPL family is incomparable with another classical subfamily of DCFLs, namely the LL one.

Theorem 5. *The LCPL family is strictly included in the DCFL family and in the right-to-left deterministic context-free language family.*

Proof. By Remark 1 we know that any LCPL can be recognized by a single valued LCPA. Then, considering only the set of computations of a single valued LCPA that correspond to the reverse of the rightmost visit of syntax trees, one obtains a traditional deterministic pushdown automaton. Thus, LCPLs are DCFLs.

The inclusion is strict:

$$L = \{0a^n b^n \mid n \geq 1\} \cup \{1a^n b^{2n} \mid n \geq 1\} \tag{2}$$

is a deterministic context-free language that cannot be generated by an LCPG; we prove that, for any grammar G recognizing L, the set $\mathcal{X}_G$ and $\mathcal{C}_G$ must be conflictual. First notice that if $\boldsymbol{x} \in \mathcal{X}_G$ and $\boldsymbol{x}$ contains a substring $a[^+y]^+b$ with $y \in \Sigma^+$, then $a\langle y\rangle b \in \mathcal{C}_G$.

To generate strings like $a^n b^n$, it is necessary to have derivations of the type $A \overset{*}{\Rightarrow} a^k A b^k$ for some $k > 0$, which implies that $\mathcal{X}_G$ contains an xchain $\boldsymbol{x}a[\boldsymbol{y}]b\boldsymbol{z}$ with $\widetilde{\boldsymbol{y}} = a^k b^k$. Splittings like $B \overset{*}{\Rightarrow} a^h C$, $C \overset{*}{\Rightarrow} B b^h$ would lead to conflicts, since we have one of the following cases.

- If $B \stackrel{*}{\Rightarrow} a^hC$ contains both right linear and left linear steps, then it produces xchains containing $\boldsymbol{x_1}[[a^{j_1}]a^{k_1}]z_1$ with $\widetilde{\boldsymbol{x_1}}, \widetilde{z_1} \in a^+$, and $\boldsymbol{x_2}[a^{k_2}[a^{j_2}]]z_2$ with $\widetilde{\boldsymbol{x_2}}, \widetilde{z_2} \in a^+$ that conflict with its chains $a\langle a^{j_2}\rangle a$ and $a\langle a^{j_1}\rangle a$, respectively; similarly for derivation $C \stackrel{*}{\Rightarrow} Bb^h$.
- Otherwise, if $B \stackrel{*}{\Rightarrow} a^hC$ is right linear and $C \stackrel{*}{\Rightarrow} Bb^h$ is left linear, then they produce xchains containing $\boldsymbol{x}[a^j]z$ and $\boldsymbol{x'}[b^i]z'$ for some $i, j > 0$, $\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{x'}} \in a^+$, $\widetilde{z}, \widetilde{z'} \in b^+$, and hence the chains $a\langle a^j\rangle b$ and $a\langle b^i\rangle b$. Thus, $\boldsymbol{x}[a^j]z$ (with long enough z) conflicts with $a\langle b^i\rangle b$, and $\boldsymbol{x'}[b^i]z'$ (with long enough $\boldsymbol{x}$) conflicts with $a\langle a^j\rangle b$.

Similarly, and more generally, we must exclude splittings like $B \stackrel{*}{\Rightarrow} a^hb^jC$, $C \stackrel{*}{\Rightarrow} Bb^{h-j}$ (or the symmetric ones). Hence, all steps in derivation $A \stackrel{*}{\Rightarrow} a^kAb^k$ are balanced and we must have a chain $a\langle a^hb^h\rangle b$ for some h.

Analogously, considering strings like $1a^nb^{2n}$, $\mathcal{X}_G$ must contain an xchain $\boldsymbol{x'}a[\boldsymbol{y'}]bz'$ with $\widetilde{\boldsymbol{y'}} = a^ib^{2i}$ for some i. Since this xchain conflicts with $a\langle a^hb^h\rangle b$, G is not a locally chain parsable grammar. □

Theorem 6. *The LCPL family is incomparable with the family of languages generated by LL(k) grammars.*

Proof. The witness that proves LL(1) $\not\subseteq$ LCPL is the language in Eq. (2). The relation LL(1) $\not\supseteq$ LCPL is proved by the OPL language $\{a^nb^n \mid n \geq 1\} \cup \{a^nc^n \mid n \geq 1\}$ which is easily generated by an LCPG but is not in LL(k) [24]. □

Remark 4. *Although Theorem 5 does not say anything about the parsing complexity of a local parser, its proof shows that an LCPA can be used to recognize any string in linear time: this result can be achieved considering only the computations of a single valued LCPA that correspond to rightmost visits of syntax trees.*

Closure properties

When dealing with closure properties, we notice that they change dramatically whether we refer to unstructured context-free languages or to structured ones; for instance both DCFLs and general CFLs enjoy (differently from each other) some closure properties but not all of them. On the other hand, the languages generated by parenthesis grammars are structured in the sense that their strings immediately and uniquely represent the syntax tree associated with them. They enjoy closure under all Boolean operations and make a Boolean algebra where the top or universal element is the language generated by the stencil grammar obtained from the original ones. The various language families that may be considered input-driven are naturally structured too; for instance the original IDLs (alias visibly pushdown languages) generalize parenthesis languages in that the so named *call*/*return* terminals play the role of open/closed parentheses. Operator precedence languages, to be reconsidered in Section 7, too are structured, although their structure is less perspicuous in the terminal strings [9]. For such language families, the investigation of closure properties has always been more profitable when referred to the sentence structures rather than simply to the strings; for

instance, IDL closure properties assume that the partitioning of the alphabet into calls, returns, and internal is fixed *a priori*.

Since LCPLs too are input-driven and have a structure determined by their chains, it seems appropriate to investigate their closure properties with reference to the languages that share a predefined structure. Chains, however, unlike the alphabet partitioning of IDLs and the operator precedence matrices of OPLs, defined in Section 7, are not enough by themselves to give a unique structure to all languages sharing them: it may even happen that of two grammars with identical chains one is LCP and the other is not. For instance the two grammars below

$$G_1: \quad A \to bBa \quad B \to ab \quad S = \{A\}$$

$$G_2: \quad C \to bDa \quad D \to Dab \mid ab \quad S = \{C\}$$

have the chains $\#\langle ba\rangle\#$, $b\langle ab\rangle a$, but, whereas G_1 clearly has no conflicts, G_2 does, e.g., $b\langle ab\rangle a$ vs. $\#[b[[[ab]ab]ab]a]\#$. Thus, rather than referring to a whole family qualified by a unique set of chains, we restrict our investigation to pairs of automata or grammars exhibiting sets of compatible xchains and chains, as defined next.

Definition 10. *Two LCPAs $\mathcal{A}_1 = (\Sigma, Q_1, \delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, \delta_2, F_2)$ are structurally compatible –briefly, compatible– if the union of their xchains, $\mathcal{X}_{\mathcal{A}_1} \cup \mathcal{X}_{\mathcal{A}_2}$, and the union of their chains, $C(\mathcal{A}_1) \cup C(\mathcal{A}_2)$, are not conflictual. Two LCPLs L_1 and L_2 are* compatible *if they are recognized by two compatible LCPAs.*

The Boolean closure properties under union, intersection and difference are next proved for any pair of compatible LCPLs.

Theorem 7. *Let L_1 and L_2 be compatible LCPLs. Then $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 \setminus L_2$ are LCP, and are still compatible with L_1 and L_2.*

Proof. Let $\mathcal{A}_1 = (\Sigma, Q_1, \delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, \delta_2, F_2)$ be compatible LCPAs recognizing respectively L_1 and L_2.

We may safely assume that the set of states Q_1 and Q_2 are disjoint. Let $Q = (Q_1 \cup \{\bot, q_{\text{err}}\}) \times (Q_2 \cup \{\bot, q_{\text{err}}\})$, with $\bot, q_{\text{err}} \notin Q_1 \cup Q_2$. For each string $\gamma \in \text{OF}(Q)$, say $\gamma = q_0 a_1 q_1 a_2 q_2 \cdots a_n q_n$ with $q_i \in Q \cup \{\varepsilon\}$, define $\gamma_1 = p_0 a_1 p_1 a_2 p_2 \cdots a_n p_n$ where p_i is empty whenever q_i is empty or has $\bot$ as first component, and p_i is the first component of q_i in the other cases; γ_2 is defined symmetrically. Then $\delta(a, \gamma, b)$ is as follows:

- $\delta(a, \gamma, b)$ is the set of all pairs (q_1, q_2) with $q_1 \in \delta_1(a, \gamma_1, b)$ and $q_2 \in \delta_2(a, \gamma_2, b)$, if both $\delta_1(a, \gamma_1, b)$ and $\delta_2(a, \gamma_2, b)$ are nonempty;
- $\delta(a, \gamma, b) = \emptyset$, if both $\delta_1(a, \gamma_1, b)$ and $\delta_2(a, \gamma_2, b)$ are undefined or empty;
- $\delta(a, \gamma, b)$ is the set of all pairs (q_{err}, q_2) with $q_2 \in \delta_2(a, \gamma_2, b)$, if only $\delta_1(a, \gamma_1, b)$ is undefined or empty, and similarly for the symmetric case.

For $\diamond \in \{\cap, \cup, \setminus\}$, the automaton for $L_1 \diamond L_2$ is given by $(\Sigma, Q, \delta, F_\diamond)$, where: $F_\cap = F_1 \times F_2$, $F_\cup = F_1 \times (Q_2 \cup \{q_{err}\}) \cup (Q_1 \cup \{q_{err}\}) \times F_2)$, $F_\setminus = F_1 \times (Q \setminus F_2 \cup \{q_{err}\})$. Notice that the above construction does not produce new xchains besides $\mathcal{X}_{\mathcal{A}_1} \cup \mathcal{X}_{\mathcal{A}_2}$. □

Corollary 2. *The inclusion problem between compatible LCPLs is decidable.*

Proof. The statement is a direct consequence of the decidability of the emptiness problem for CFLs and the closure between compatible LCPLs under set difference. □

These properties make locally chain parsable languages suitable for devising automatic verification techniques, most notably model checking.

The neutrality of LCPA computations with respect to the direction of moves leads to the next property, stating that a language L is an LCPL if, and only if, its reversal is an LCPL. Notice that L^R and L need not to be compatible.

Theorem 8. *Let L be an LCPL. Then its reversal L^R is an LCPL.*

Proof. Consider a local chain parser $\mathcal{A} = (\Sigma, Q, \delta, F)$ such that $L(\mathcal{A}) = L$. A local chain parser for L^R is $\mathcal{A}_R = (\Sigma, Q, \delta_R, F)$, with $\delta_R(a, \gamma, b) = \delta(b, \gamma^R, a)$, for every a, b, γ such that $\delta(a, \gamma^R, b) \neq \emptyset$. It is easy to see that $\mathcal{A}_R$'s xchains are like those of $\mathcal{A}$, but reversed. □

On the contrary, we are going to prove that the hypothesis of structural compatibility assumed in Theorem 7 does not suffice to ensure the closure under the operations of concatenation and star. This is to be expected: in fact, concatenation necessarily alters any chain of any grammar generating L_1 that has the form $a\langle y\rangle\#$, because the # marking the end of L_1 disappears; the alteration may in some cases cause a conflict. We also show that homomorphism does not preserve local parsability, which is a constant property of all families of input-driven languages.

Theorem 9. *There exist compatible locally chain parsable languages L_1 and L_2 such that $L_1 \cdot L_2$ is not LCPLs.*

Proof. Consider $L_1 = a^+ \cup \left\{a^{2n+1}b^{2n+1} \mid n \geq 0\right\}$ and $L_2 = c^+ \cup \left\{b^{2n+1}c^{2n+1} \mid n \geq 0\right\}$. It is easy to see that they are LCPL and compatible.

We prove that

$$L_1 \cdot L_2 = \quad a^+c^+ \cup a^+\left\{b^{2n+1}c^{2n+1} \mid n \geq 0\right\} \cup \left\{a^{2n+1}b^{2n+1} \mid n \geq 0\right\}c^+ \cup \left\{a^{2n+1}b^{2n+1} \mid n \geq 0\right\}\left\{b^{2n+1}c^{2n+1} \mid n \geq 0\right\}$$

is not deterministic, hence cannot be an LCPL because of Theorem 5.
Let R be $a(aa)^*b(bb)^*c(cc)^*$. The language

$$L_R := (L_1 \cdot L_2) \cap R = a(aa)^*\left\{b^{2n+1}c^{2n+1} \mid n \geq 0\right\} \cup \left\{a^{2n+1}b^{2n+1} \mid n \geq 0\right\}c(cc)^*$$

is inherently ambiguous, hence not deterministic: the classical proof (e.g., see Theorem 5.31 in [15]) that language $a^+b^nc^n \cup a^nb^nc^+$ is inherently ambiguous carries over to this case. Indeed, let N be the pumping number for the grammar as in Ogden's lemma. If N is odd, the proof of Theorem 5.31 in [15] is based on the two strings $a^Nb^Nc^{N+N!}$, $a^{N+N!}b^Nc^N$ and still holds. If N is even, the only difference is that the considered strings must be $a^{N+1}b^{N+1}c^{N+1+N!}$ and $a^{N+1+N!}b^{N+1}c^{N+1}$.
Since deterministic languages are closed under intersection with regular sets, it follows that also $L_1 \cdot L_2$ is not deterministic. □

Theorem 10. *The LCPL family is not closed under Kleene star and under letter-to-letter homomorphism.*

Proof. The non-closure under Kleene star is a corollary of the preceding argumentation. Consider the language $(L_1 \cup L_2)^*$; its intersection with R is exactly L_R. Notice that $(L_1 \cup L_2)$ is an LCPL because of Theorem 7. Since deterministic languages are closed under intersection with regular ones, if $(L_1 \cup L_2)^*$ were deterministic, so would be L_R, that has just been proven to be inherently ambiguous.

For homomorphism, as proved in Theorem 5, language $L = \{0a^n b^n \mid n \geq 1\} \cup \left\{1a^n b^{2n} \mid n \geq 1\right\}$ is not an LCPL, yet it is the image of the LCPL $\{0a^n b^n \mid n \geq 1\} \cup \left\{1a^n c^{2n} \mid n \geq 1\right\}$ under the homomorphism that maps c to b and leaves all the other characters unchanged. □

7. LCPL versus Operator-Precedence and Input-Driven languages

It is worthwhile to examine the LCPL family as an outgrowth of the classical OPL family [13], whose knowledge, both theoretically ([9, 18] and for application to parallel parsing [4]), has much progressed in recent years.

R. Floyd took inspiration from the traditional notion of precedence between arithmetic operators in order to define a broad class of languages, such that the shape of the parse tree is solely determined by a binary relation between terminals that are consecutive, or become consecutive after a bottom-up reduction step. Thus, the parsing of such languages is driven by the terminal alphabet only, as it happens for IDLs and our LCPLs. Recent and much less recent work has subsequently proved interesting algebraic properties of OPLs, e.g., qualifying the OPL family as the largest known (to us) language family structurally closed under all basic language operations (the Boolean ones, concatenation, Kleene *, ...) and characterized in terms of monadic second order logic [9, 18]; we also showed that OPLs enjoy the local parsability property and exploited it to build an efficient parallel parser therefor.

In this section we show that LCPLs are a further generalization of OPLs and preserve several but not all of the above properties: precisely, the LCPL family strictly contains the OPL family; on the other hand we have seen in Section 6 that, e.g., with the structural compatibility hypothesis they are closed under Boolean operations, but not under concatenation and Kleene *. After resuming the basic definitions and properties of OPLs for the sake of self-completeness, we formally prove that OPLs are LCPLs and provide examples of LCPLs that are not OPLs; we also show that the increased generative power of LCPLs allows us to capture intricate syntactic features of some programming languages that are not expressible in terms of OPLs.

The following definitions for operator precedence grammars [13], are from [9].

Definition 11. *For an OG G and a nonterminal A, the left and right terminal sets are*

$$\mathcal{L}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} Ba\alpha\} \qquad \mathcal{R}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} \alpha aB\}$$

where $B \in V_N \cup \{\varepsilon\}$, D is a new nonterminal, and G' is the same as G except for the addition of the rule $D \to \beta$. Notice that $\mathcal{L}_G(\varepsilon) = \emptyset$.

Three binary operator precedence (OP) relations are defined:

$$\begin{aligned} \textit{equal in precedence: } a \doteq b &\iff \exists A \to \alpha a B b \beta, B \in V_N \cup \{\varepsilon\} \\ \textit{takes precedence: } a \gtrdot b &\iff \exists A \to \alpha D b \beta, D \in V_N \textit{ and } a \in \mathcal{R}_G(D) \\ \textit{yields precedence: } a \lessdot b &\iff \exists A \to \alpha a D \beta, D \in V_N \textit{ and } b \in \mathcal{L}_G(D) \end{aligned}$$

The operator precedence matrix *(OPM) $M = OPM(G)$ is a $|\Sigma| \times |\Sigma|$ array that to each ordered pair (a, b) associates the set M_{ab} of OP relations holding between a and b. For two OPMs M_1 and M_2, we define set union $M_1 \cup M_2$ if $\forall a, b : M_{ab} = M_{1,ab} \cup M_{2,ab}$.*

Definition 12. *G is an operator precedence grammar (OPG) if $M = OPM(G)$ is a conflict-free matrix, i.e., $\forall a, b : |M_{ab}| \leq 1$. L is an operator precedence language (OPL) if it is generated by an OPG. Two OPMs are compatible if their union is conflict-free.*

Intuitively, the OPM of an OPG drives the parsing algorithm in that the terminal part of a grammar r.h.s. is enclosed, in a unique way, within a pair *yields precedence*, *takes precedence* and *equal in precedence* holds between the consecutive terminals in between. This property is exploited in the following Theorem 11 which asserts the strict containment of the OPL family within the LCPL family.

OPLs having compatible OPM are closed with respect to Boolean operations, concatenation, Kleene *, reversal, prefix, suffix, homomorphism preserving precedence relations, intersection with regular languages[2] [10, 9]. Every IDL is an OPL having very restricted precedence relations induced by the partition of the terminal alphabet into opening, closing, and internal symbols; e.g., $\{a^n b^n \mid n \geq 1\}$ is an IDL where a and b, are, respectively, opening and closing symbols [9]. Therefore, IDL closure properties can be derived as a special case of the closure properties of OPL.

Theorem 11. *Every OPG is locally chain-parsable. The OPL family is strictly contained within the LCPL family.*

Proof. The grammatical chains C_G are determined by the precedence relations of G, as follows: $a\langle c_1 \cdots c_k \rangle b \in C_G$ if, and only if, $a \lessdot c_1$, $c_i \doteq c_{i+1}$ for every $1 \leq i < k$, and $c_k \gtrdot b$. Consider now any derivation $\#T\# \overset{*}{\Rightarrow} \alpha a \gamma b \beta$ with $\widehat{\gamma} = y$, $a\langle y \rangle b \in C_G$; since for each pair of terminals at most one precedence relation holds, necessarily a yields precedence to the first terminal symbol of γ, the last terminal symbol of γ takes precedence over b, and $\doteq$ holds between any pair of consecutive terminals in y. Thus, the above derivation must be decomposed into $\#T\# \Rightarrow^* \alpha' a A b \beta' \Rightarrow^* \alpha' a \gamma b \beta' \Rightarrow^* \alpha a \gamma b \beta$: in fact deriving γ in separate steps (e.g., $\#T\# \Rightarrow^* \alpha' a \gamma_1 \delta \Rightarrow^* \alpha' a \gamma_1 \gamma_2 b \beta' \Rightarrow^* \alpha a \gamma b \beta$, with $\gamma_1 \gamma_2 = \gamma$, γ_1 and $\gamma_2 \neq \varepsilon$) would imply the existence of a $\lessdot$ or of a $\gtrdot$ relation within γ in conflict with the $\doteq$ relation (in this example the last terminal symbol of γ_1 takes precedence over the first terminal symbol of γ_2).

[2] To be precise, this last closure is an immediate consequence of Corollary 17 in [9], though not explicitly listed in the paper. Also notice that closure under reversal is obtained by applying a "symmetric" OPM which reverses yield and take precedence relations in exactly the same way as it is obtained in Theorem 8 by reversing the chains.

The strict inclusion is witnessed by the language, $L = \{a^n b^n \mid n \geq 1\} \cup \{b^n a^n \mid n \geq 1\}$, which is recognized by the obviously local automaton driven by the chains: $\#\langle ab\rangle\#$, $\#\langle ba\rangle\#$, $a\langle ab\rangle b$, $b\langle ba\rangle a$. However, any grammar generating $\{a^n b^n \mid n \geq 1\}$ (respectively $\{b^n a^n \mid n \geq 1\}$) necessarily exhibits the $a \lessdot a$ relation (respectively the $a \gtrdot a$ relation); this, in turn, is an immediate consequence of the fact pointed out in the proof of Theorem 5, that, to generate strings like $a^n b^n$, it is necessary to have derivations of the type $A \stackrel{*}{\Rightarrow} a^k A b^k$ for some $k > 0$. □

The generative capacity of LCPGs in the field of programming languages

The increased generative power of LCPGs with respect to OPGs is potentially exploitable beyond the mathematical realm of formal languages. For instance, OPLs have been proved effective in the definition of programming language syntax and in compiler construction, but they notoriously suffer from a few minor weaknesses which required some ad-hoc trick in their parsers. A first "historical" example is the case of operator "unary minus" already pointed out and treated ad-hoc in the original Floyd's paper; in the more recent work the unary minus has been disambiguated from the binary one by means of a preprocessing during the lexical analysis [3] .

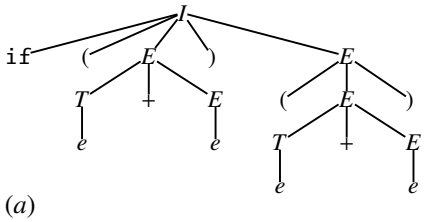
In this section we offer two examples that show how some limitations of OPGs can be overtaken by LCPGs. Precisely, Example 10 deals with a typical case of OPM conflict of several C-like languages produced by conditional expressions; Example 11 describes a grammar fragment suitable to express the syntax of arithmetic expressions that include the unary minus operator.

Example 10. *The following grammars describe a typical syntactic feature of several C-like languages where expressions can be used as statements; since expressions can be embraced by parentheses, the closed parenthesis of the expression defining, e.g., the condition of an if-statement can be immediately followed by the open parenthesis of the expression defining the "then-branch" of the conditional statement. For simplicity we limit the grammars to formalize parenthetic expressions containing only one additive and one multiplicative operator, then* $\Sigma = \{\texttt{if}, (,), +, *, e\}$.

The first grammar is in Figure 1 (a), where a syntax tree is shown for illustration; the axiom set is the singleton $\{I\}$. *It is easy to verify that this grammar is neither an OPG nor a LCPG: for instance the chain* $)\langle +\rangle\#$, *which requires to reduce a r.h.s.* $T + E$ *before reducing the r.h.s. that includes the*), *conflicts with the xchain* $\#[\texttt{if}\,()[+[[([()])]+]]]\#$ *which imposes to reduce the last*) *before the last* +. *However, by applying a technique (not an algorithm!) that, roughly speaking, consists in "raising the level" of some r.h.s. without altering the semantic precedence between the two operators, we obtain the equivalent grammar in Figure 1 (b), which is an LCPG (proven through our tool) but still not an OPG.*

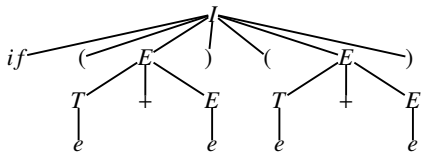
In this case the adopted technique simply consisted in replacing some nonterminals that were the source of conflicts with the corresponding r.h.s.; not surprisingly the procedure had to be iterated but "fortunately" without producing non-terminating self loops. The example also offers an intuitive explanation of why the same procedure did not transform the original grammar into an OPG: in that case the effect would have been to replace a conflict between $\gtrdot$ *and* $\lessdot$ *into one between* $\lessdot$ *and* $\doteq$. *The apparent*

$I \rightarrow$ `if`$(E)E$
$E \rightarrow T + E \mid e \mid (E)$
$T \rightarrow e * T \mid e \mid (E) \mid (E) * T$



(*a*)

$I \rightarrow$ `if`$(E)(E) \mid$ `if`$(E)e + E \mid$ `if`$(E)e + T \mid$
`if`$(E)e * T \mid$ `if`$(E)e \mid$ `if`$(E)(E) + T \mid$
`if`$(E)(E) + E \mid$ `if`$(E)(E) * T$
$E \rightarrow T + E \mid e \mid (E)$
$T \rightarrow e * T \mid e \mid (E) \mid (E) * T$



(*b*)

Figure 1: Part (*a*): the original non-locally chain parsable grammar for the "if" construct and syntax tree of `if`$(e + e)(e + e)$. Part (*b*): the equivalent LCPG and the corresponding syntax tree.

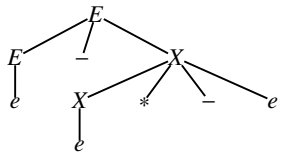
drawback of the applied technique is the considerable increase in the number and length of the grammar rules; we will shortly comment thereon in the conclusions.

Example 11. *Consider now the grammar in Figure 2 (a) which defines a simplified syntax for arithmetic expressions that include binary and unary operators. To keep the example small, the grammar features only one subtraction and one multiplication operator and e as operand, but other operators and parentheses would be straightforward to add. E is the only axiom.*

It is easy to verify that the above grammar too is not an OPG: there is an obvious conflict $- \doteq -$ *and* $- \lessdot -$*. Indeed the grammar is not even an LCPG: for instance, the xchain* $\#[[[[- - e] - [-e]]*]*]\#$ *conflicts with the chain* $-\langle e\rangle -$*. In this case too we can transform the original grammar into the equivalent one in Figure 2 (b) (proven free from conflicts and therefore LCPG by means of our automatic tool.)*

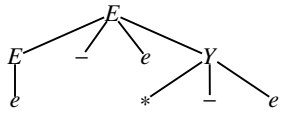
The adopted technique is similar to the previous one though, at a first glance the transformation may appear unnatural and rather tricky: in fact, in this case rather than raising a whole r.h.s. to an upper level we have "split" it by replacing its original l.h.s. with a new nonterminal (Y replaced X) and we attached it where its original r.h.s. could occur; as a consequence we see new r.h.s. that begin with the $*$ *operator; observe, however, that the transformation keeps the key syntactic feature of giving the multiplication operator its usual semantic precedence over the subtraction one as it is illustrated in Figure 2 which shows that the "essential structure" of the grammar's syntax trees is not affected. On the other hand, that the unary minus remains, as it was*

$$E \rightarrow \quad e \mid -e \mid E * X \mid E - X \mid E - e \mid E - -e$$
$$X \rightarrow \quad e \mid -e \mid X * -e \mid X * e \mid$$



(*a*)

$$E \rightarrow \quad e \mid -e \mid eY \mid -eY \mid E - e \mid$$
$$\qquad\quad E - -e \mid E - eY \mid E - -eY$$
$$Y \rightarrow \quad *e \mid * - e \mid *eY \mid * - eY$$



(*b*)

Figure 2: Part (*a*): original non-locally chain parsable grammar for the expressions with unary and binary minus operators, and syntax tree of $e - e * -e$. Part (*b*): the equivalent LCPG and the corresponding syntax tree.

originally, at the same level of both $*$ and the binary $-$; since we do not admit nested unary minuses, however, this does not prevent the semantic analysis from processing the unary operator before the preceding binary one.

The above examples can be further generalized: for instance we have built an LCPG that describe conditional expressions involving both the above parenthesization typical of C-like languages and all arithmetic and assignment operators, including the unary minus; we have also been able to deal with the fairly tricky Javascript's syntax which allows to write conditional expressions containing an unbounded sequence of parenthesized sub-expressions such as `if`$(E)(E)(E)...(E)$. As we already admitted, however, the systematic application of the techniques exemplified in the above examples to remove conflicts from the original BNF considerably increases the grammar size; for this reason we do not include in this paper such more complex examples, which are however reported in `https://github.com/bzoto/chainsaw`.

8. Related work

Having previously clarified the close relationship of our work with the input-driven languages, we first enlarge the comparison to some other language families that, similarly to ours, strictly include the input-driven family. Then we briefly mention some interesting analogies with other language families, which have somewhat influenced our ideas.

The language family recognized by one-way real-time cellular automata (also known as *trellis* automata) coincides with the family of *linear conjunctive languages* denoted *LinConj* (see [24] where other relevant references are available). Such family strictly contains the input-driven family and is incomparable with the CF family and also with the family defined by LL(k) CF grammars and denoted *CFLL*. The same paper shows,

by means of witness languages, that CFLL is incomparable with the input-driven family. Clearly LinConj $\not\subseteq$ LCPL. The question whether LCPL is strictly included into LinConj remains open. With respect to the parsing algorithms, trellis automata are a sort of parallel machine which analyzes the substrings, starting from the single characters, and then gradually combines the partial analysis of adjacent substrings until the entire analysis is obtained; the time complexity is quadratic. In contrast, an LCPL chain-driven parser is a linear-time algorithm that can be executed either serially or in parallel, as explained for the similar OPL case in [3].

We mention another family strictly including the input-driven languages, the so called *tinput-driven* family [17], where letter "t" indicates that a deterministic finite-state *transducer* is used to translate the input string to another string, which is then parsed by an input-driven pushdown machine. Such finite-state preprocessing is, for instance, able to translate a string from language $\{a^n ba^n \mid n \geq 1\}$, which is not input-driven (and is not LCPL either), to a string of the form $a^n bc^n$, which is clearly accepted by an input-driven machine. From this example it follows that OPL $\not\supseteq$ *tinput-driven*. On the other hand, since [17] the *tinput-driven* family is a strict subset of the real-time deterministic CF family, and the OPL family includes also non-real-time languages [9], it follows that OPL $\not\subseteq$ *tinput-driven*, and by Theorem 11, the same holds for family LCPL. Actually, the use of finite-state preprocessing for taming a recalcitrant language is a time-honored technique used by some compilers that preprocess the source text at lexical analysis time, and is extensively described for the case of OPG in [3].

It would be too long to examine all language families that have extended in recent years the idea of input driven languages, such as, e.g., the languages recognized by the synchronized pushdown automata of [7], which have similar Boolean closure properties but, unlike the LCPL, are included into the real-time deterministic context-free languages.

Next we focus on other approaches. The NTS grammars and languages [6] are defined by the so called *non-terminal separation* property. They enjoy a sort of local parsability property in the following sense: if a substring (with terminals and nonterminals) occurs in a sentence as a constituent, i.e., is generated by a nonterminal symbol, then, for every sentence, the same substring can be reduced to the same nonterminal symbol. NTS languages, however, are not input-driven, because they rely on the presence of nonterminals for localizing the position of a reduction. To increase the practically insufficient generative capacity of NTS grammars, researchers working on grammatical inference [19] have recently incorporated into the model the idea of checking the terminal context (of length one or greater than one but still bounded) that surrounds the substring to be reduced; this is similar to our notion of context in a chain.

Actually, the idea of making grammatical reductions more selective by checking a bounded context is much older: we already mentioned Floyd's *bounded-context* contex-free grammars [14], theoretically studied in [20]. They are not input-driven (unlike Floyd's OPG model) because they admit nonterminal symbols in the context that surrounds a string, also possibly containing nonterminal symbols, which is candidate to reduction. In this way bounded-context grammars are able to generate also languages which are not in the LCPL family: an example is $\{a^n ba^n \mid n \geq 1\}$ of Example 5.

Moving to another research area, the conditions for local parsability, as expressed

in Lemma 1, are remindful of the confluence property of Church-Rosser languages (also called McNaughton languages), which are defined by string-rewriting rules [22, 5]. Such systems, under the length-reducing hypothesis that ensures that the length of reduction chains is not infinite, bear some similarity to our approach. But they are more powerful than ours, because they define also deterministic context-sensitive languages. Moreover, they are not input-driven in any sense, since the rules contain also nonterminal symbols.

On the other hand, nonterminal symbols are not used at all in the string-rewriting rules of the Church-Rosser *congruential* languages [12], a restricted family where each rewriting rule is specified by a pair of terminal strings; in particular, the already mentioned NTS languages are congruential. Congruential languages include all regular languages, are incomparable with the DCFLs (they may be context-sensitive), and, in our opinion, are too weak in generative capacity, to be useful for defining practically relevant languages.

To sum up, to our knowledge, the input-driven locally chain parsable automata and grammars differ from all existing, somewhat related, models, either, or both, with respect to the local parsability property and to the input-driven aspects.

9. Conclusion

The LCPL family properly extends the known input-driven families, and, under suitable, decidable hypotheses, maintains the decidability of the containment problem. This represents a new step in the long term research effort towards a general theory of local deterministic parsing. Much remains to be done to better understand the properties of LCPLs, and we just mention two questions left for investigation: closure under intersection with regular languages and iterative pumping properties.

Another research direction is to examine whether some lattice-theoretical properties of OP grammars and languages [10] can be extended to the LCPL family; such algebraic properties motivated the early use of OPGs for grammatical inference [11].

Concerning closure properties, it is not difficult to find compatible LCPLs such that, say, their concatenation preserves the local parsing property: for instance $\{a^n b^n \mid n \geq 1\}$ and $\{b^n c^n \mid n \geq 1\}$. We observe that the known closure under concatenation result of operator-precedence languages does not apply to this case because the two languages have conflicting precedence relations. It remains for investigation to discover a sufficient, yet not overly restrictive, condition for the closure of LCPLs under concatenation and star.

It would also be interesting to study the possible gain in generative capacity that may be obtained by extending the width of the terminal context of chains, from one to larger integers, in a way similar to Floyd bounded-context grammars, but purely input-driven.

From the application point of view we envisage two major objectives. The most natural application of LCPLs is for parallel deterministic language parsing. As said, the serial deterministic parsers (LL(1) and LR(1)) are not suitable for parallelization because they cannot exploit a local parsing property. In our opinion, the best existing grammar model for parallel deterministic parsing is the OPG, which has been used in

a practical parallel parser generator [3] named PAPAGENO. The increased generative power of LCPGs versus OPGs can be exploited to define realistic language constructs, such as those exemplified in Section 7, which exceed the capacity of OPGs.

We are confident that the noticeable results obtained by PAPAGENO by parallelizing the parsing of OPLs can be extended to LCPLs; we must point out, however, that in this paper we deliberately defined our CDAs and LCPAs by abstracting away from the search for chain and the r.h.s. to be reduced; converting such a nondeterministic choice into an efficient deterministic algorithm certainly appears as a more cumbersome job than the simple search for $\lessdot$ and $\gtrdot$ exploited in OPG parsing; furthermore, the examples given in Section 7 show that the increased generative power with respect to OPGs is obtained in general at the price of longer and more numerous r.h.s.'s; this will probably affect the efficiency of pattern matching algorithms looking for chains to be reduced. Thus, converting the abstract model of LCPA into an efficient parallel parser raises new challenges of practical flavor.

Finally, a totally different application of our new family of languages may be found in the field of automatic verification. In fact Corollary 2 states the decidability of the inclusion problem for compatible LCPLs, a key property to apply model checking techniques which is not enjoyed by most deterministic language families (by looking at previous similar results it seems that being input-driven plays a major role to obtain this property). Of course, fully exploiting this basic property to obtain practical automatic verification techniques is a long intriguing research path which possibly involves providing suitable logical characterization(s) of the language family as it has been done historically in the pioneering case of the finite state formalism, and subsequently extended to a few infinite state ones [2, 1, 18].

Acknowledgment. We thank the anonymous reviewers for their valuable suggestions; in particular Theorem 6 was inspired by them.

References

[1] R. Alur, M. Arenas, P. Barceló, K. Etessami, N. Immerman, and L. Libkin. First-order and temporal logics for nested words. *Logical Methods in Computer Science*, 4(4), 2008.

[2] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[3] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, F. Panella, and M. Pradella. Parallel parsing made practical. *Sci. Comput. Program.*, 112(3):195–226, 2015. DOI: 10.1016/j.scico.2015.09.002.

[4] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. Parallel parsing of operator precedence grammars. *Inf. Process. Lett.*, 113(7):245–249, 2013.

[5] M. Beaudry, M. Holzer, G. Niemann, and F. Otto. McNaughton families of languages. *Theor. Comput. Sci.*, 290(3):1581–1628, 2003.

[6] L. Boasson and G. Sénizergues. NTS languages are deterministic and congruential. *J. Comput. Syst. Sci.*, 31(3):332–342, 1985.

[7] D. Caucal. Synchronization of Pushdown Automata. In O. H. Ibarra and Z. Dang, editors, *Developments in Language Theory*, volume 4036 of *LNCS*, pages 120–132. Springer, 2006.

[8] S. Crespi Reghizzi, V. Lonati, D. Mandrioli, and M. Pradella. Locally chain-parsable languages. In *Mathematical Foundations of Computer Science MFCS*, volume 9234 of *LNCS*, pages 154–166. Springer, 2015.

[9] S. Crespi Reghizzi and D. Mandrioli. Operator Precedence and the Visibly Pushdown Property. *J. Comput. Syst. Sci.*, 78(6):1837–1867, 2012.

[10] S. Crespi Reghizzi, D. Mandrioli, and D. F. Martin. Algebraic Properties of Operator Precedence Languages. *Information and Control*, 37(2):115–133, May 1978.

[11] S. Crespi Reghizzi, M. A. Melkanoff, and L. Lichten. The Use of Grammatical Inference for Designing Programming Languages. *Commun. ACM*, 16(2):83–90, 1973.

[12] V. Diekert, M. Kufleitner, K. Reinhardt, and T. Walter. Regular languages are Church-Rosser congruential. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II*, ICALP'12, pages 177–188, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] R. W. Floyd. Syntactic Analysis and Operator Precedence. *J. ACM*, 10(3):316–333, 1963.

[14] R. W. Floyd. Bounded context syntactic analysis. *Commun. ACM*, 7(2):62–67, 1964.

[15] R. W. Floyd and R. Beigel. *The language of machines. An introduction to computability and formal languages*. Computer Science Press, New York, 1994.

[16] M. A. Harrison. *Introduction to Formal Language Theory*. Addison Wesley, 1978.

[17] M. Kutrib, A. Malcher, and M. Wendlandt. Tinput-driven pushdown automata. In *Machines, Computations, and Universality - 7th International Conference, MCU Proceedings*, pages 94–112, 2015.

[18] V. Lonati, D. Mandrioli, F. Panella, and M. Pradella. Operator precedence languages: Their automata-theoretic and logic characterization. *SIAM J. Comput.*, 44(4):1026–1088, 2015.

[19] F. M. Luque and G. G. I. López. PAC-learning unambiguous k, l-NTS $^{<=}$ languages. In *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI Proceedings*, pages 122–134, 2010.

[20] R. W. McCloskey. *Bounded context grammars and languages*. PhD thesis, Rensselaer Polytechnic, 1993.

[21] R. McNaughton. Parenthesis Grammars. *J. ACM*, 14(3):490–500, 1967.

[22] R. McNaughton, P. Narendran, and F. Otto. Church-Rosser Thue systems and formal languages. *J. ACM*, 35(2):324–344, 1988.

[23] K. Mehlhorn. Pebbling mountain ranges and its application of DCFL-recognition. In *Automata, languages and programming (ICALP-80)*, volume 85 of *LNCS*, pages 422–435, 1980.

[24] A. Okhotin. Comparing linear conjunctive languages to subfamilies of the context-free languages. In *SOFSEM 2011: Theory and Practice of Computer Science - 37th Conference on Current Trends in Theory and Practice of Computer Science Proceedings*, pages 431–443, 2011.