Locally Chain-Parsable Languages*

Stefano Crespi Reghizzi[1], Violetta Lonati[2], Dino Mandrioli[1], and Matteo Pradella[1]

[1] DEIB - Politecnico di Milano
{stefano.crespi, dino.mandrioli, matteo.pradella}@polimi.it
[2] DI - Università degli Studi di Milano
lonati@di.unimi.it

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 (LCPL), included in deterministic context-free languages, is here defined by means of the chain-driven automaton and characterized by decidable properties of grammar derivations. Such automaton decides to reduce or not a factor in a way purely driven by the terminal characters, thus extending the well-known concept of Input-Driven (ID) (visibly) pushdown machines. LCPL extend and improve the practically relevant operator-precedence languages (Floyd), which are known to strictly include the ID languages, and for which a parallel-parser generator exists. Consistently with the classical results for ID, chain-compatible LCPL are closed under reversal and Boolean operations, and language inclusion is decidable.

1 Introduction

Syntax analysis or parsing of context-free (CF) languages is a mature research area, and good parsing algorithms are available for the whole CF family and for the deterministic subfamily (DCFL) 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 [2,3] based on Floyd's [7] operator-precedence grammars (OPG) and their languages (OPL), which are included in DCFL. This is a data-parallel algorithm that is based on a theoretical property of OPG, called *local parsability*: any arbitrary factor of a sentence can be deterministically parsed, returning the unique partial syntax-tree whose frontier is the input string.

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 production right-hand side (RHS) and reduces it. The local parsability property ensures the correctness of the syntax tree thus obtained, no matter the position of reduction applications.

The informal idea of local parsability is occasionally mentioned in old research on parallel parsing, and has been formalized for OPG in [2]. Our first contribution is to propose the definition of a new and more general class of locally parsable languages: the

* Partially supported by PRIN 2010LYA9RH-006, and CNR-IEIIT.

language family to be called *Locally Chain-Parsable* (LCPL), which gains in generative capacity and bypasses some inconveniences of OPG.

The other contribution is towards a generalization of the well-known family of input-driven (alias visibly push-down) languages (IDL) [11,1], which are characterized by push-down 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. Since the attention IDL have recently attracted is due to their rich closure and decidability properties, we hope that the introduction of a larger family of languages with similar properties may be also of interest.

To understand in what sense our LCPL are input-driven, we first recall that IDL 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 IDL have the local parsability property, a fact also stemming from the fact that IDL are included in OPL [6]. 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 IDL, we obtain the OPL family, which has essentially the same closure and decidability properties [6,9]. Loosely speaking, we may say that the input that drives the automaton for OPL 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 factor (which may contain nonterminals) on the purely terminal string orderly containing: the preceding terminal, the terminals of the factor, and the following terminal. Such triplet will be called a *chain* and the machine a chain-driven automaton. For a given CF grammar, the length of chains has an upper bound, which bounds the input portion that drives the choice of a move by the recognizer.

The paper is organized as follows. After the Preliminaries, Sect. 3 introduces the chain-driven machine as a recognizer for CF. Sect. 4 defines local chain parsability for chain-driven automata and for grammars, and proves the two notions to be equivalent; Sect. 4.1 extends the definition of chains and formulates a decidability condition for local chain parsability based on the absence of conflicts; Sect. 4.2 proves, among others, the Boolean closure and decidability properties of LCPL. Sect. 5 establishes the strict inclusion of OPL (and hence also IDL) within LCPL, and claims through a practical example that LCPG are more suitable than OPG for specifying real programming languages. Sect. 6 is on related work and draw some conclusions.

2 Preliminaries

For terms not defined here, we refer to any textbook on formal languages, e.g. [8]. 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 $\beta \in (\Sigma \cup \Delta)^*$ is in *operator form* if it contains one or more terminals and does not contain a factor from Δ^2, i.e., no adjacent symbols from Δ; OF(Δ) denotes the set of all operator form strings over $\Sigma \cup \Delta$.

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. The *total* alphabet is

$V = V_N \cup \Sigma$. The *stencil* of a rule $A \to \alpha$ is the rule $N \to \sigma(\alpha)$, where $\sigma : V_N \to \{N\}$ maps every nonterminal to the new symbol $N \notin V$.

The *derivation relation* for a grammar G is denoted as usual by $\Rightarrow_G$ and its reflexive and transitive closure by $\overset{*}{\Rightarrow}_G$. The set of *sentential forms* (s.f.) generated by G is $\mathrm{SF}_G = \{\alpha \in V^* \mid T \overset{*}{\Rightarrow}_G \alpha, T \in S\}$ and the language generated is $L(G) = \mathrm{SF}_G \cap \Sigma^*$. A grammar is *invertible* if no two rules have identical r.h.s. A grammar is an *operator grammar* (OG) if all r.h.s.'s are in $\mathrm{OF}(V_N)$. Any CF grammar that does not generate ε admits an equivalent OG, which can be assumed to be invertible [8]. Clearly, every s.f. of an OP grammar is in $\mathrm{OF}(V_N)$. In this paper we deal only with reduced OG.
The following naming convention is adopted, unless otherwise specified: lowercase Latin letters $a, b, \ldots$ denote terminal characters; uppercase Latin letters $A, B, \ldots$ denote nonterminal characters; lowercase Latin letters $x, y, z \ldots$ denote terminal strings; and Greek lowercase letters $\alpha, \ldots, \omega$ denote strings over $\Sigma \cup V_N$.
We use bold symbols to denote strings over an alphabet that includes the square brackets, e.g. $\boldsymbol{x} \in (\Sigma \cup \{[,]\})^*, \boldsymbol{\alpha} \in (\Sigma \cup V_N \cup \{[,]\})^*$. We introduce the following short notation for frequently used operations based on 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 $\widetilde{\boldsymbol{\alpha}} = \widetilde{\boldsymbol{\beta}}$. Also, we use $\boldsymbol{\alpha}_{\text{first}}$ and $\boldsymbol{\alpha}_{\text{last}}$ for taking the first or last symbol in Σ from a string $\boldsymbol{\alpha}$. The same notation is applied when V_N is replaced by the state set of a machine.
For a CF 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 $\boldsymbol{x_1} \mathrel{\tilde{=}} \boldsymbol{x_2}$. 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, that can be seen as an abstract parser for CF languages. As stated in the introduction such type of abstract parser is particularly well-suited to exploit parallel implementation. First we give and illustrate by example the formal definition of chain-driven automaton, then we prove the equivalence between chain-driven automata and CF grammars.

The key driver in the search for a string to be reduced is the concept of chain. According with the general philosophy of input-driven languages and other similar families, such as, e.g., OPL, 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

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 chain-driven automaton 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 factor to be reduced is detected depending on input symbols only; enriching states are used then to (nondeterministically) determine which state will replace the detected factor.

Definition 2. *A* chain-driven automaton *is a tuple* $(\Sigma, Q, C, \delta, F)$ *where*

- Σ *is the input alphabet;*
- Q *is a finite set of states;*
- C *is a finite set of chains;*
- $\delta : \Sigma \times \mathrm{OF}(Q) \times \Sigma \to \mathcal{P}(Q)$ *is the* reduce *function, where* $\delta(a, \gamma, b) \neq \emptyset$ *implies* $a\langle\widehat{\gamma}\rangle b \in C$.
- $F \subseteq Q$ *is the set of final states.*

A *configuration* of the automaton is a string $\gamma \in \mathrm{OF}(Q)$. 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 *reduction move* is defined as follows: the automaton may perform the move

$$\alpha\ a\gamma b\ \beta \overset{a\langle y\rangle b}{\longmapsto} \alpha\ aqb\ \beta$$

where $\widehat{\gamma} = y$, and $\delta(a, \gamma, b) \ni q$. Hence, a move of the automaton deletes a factor in $\mathrm{OF}(Q)$ (corresponding to the body of the chain in C, possibly enriched with states in Q) and replaces it with a state.

A *computation* of the automaton is a sequence $K_0 \overset{c_1}{\longmapsto} K_1 \overset{c_2}{\longmapsto} K_2 \overset{c_3}{\longmapsto} \cdots \overset{c_n}{\longmapsto} K_n$ where K_i are configurations, c_i are chains. When not relevant we omit the chain and write simply $K_1 \longmapsto K_2$. We also use $\overset{*}{\longmapsto}$ to denote the reflexive and transitive closure of $\longmapsto$. The language accepted by the automaton is defined as $L(\mathcal{A}) = \{x \in \Sigma^* \mid \#x\# \overset{*}{\longmapsto} \#q\# \text{ with } q \in F\}$.

Example 1. Consider the language of arithmetic expressions on $\{e, +, *\}$ with the obvious meaning of symbols. A chain-driven automaton recognizing such expressions can be defined as follows: C 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.

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

Here are two accepting computations for $e + e * e$:

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

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

In general, the syntactic structure is not uniquely determined, since different computations may associate different structures with the same accepted string.

As for grammars, we can formalize a notion of structural ambiguity by using parenthesis automata.

Definition 3. *For a chain-driven automaton $\mathcal{A} = \langle \Sigma, Q, C, \delta, F \rangle$, the associated* parenthesis automaton *is the chain-driven automaton $[\mathcal{A}] = \langle \Sigma \cup \{[,]\}, Q, [C], \delta', F \rangle$ where $[C]$ is the set of chains $a\langle [y] \rangle b$ such that $a\langle y \rangle b \in C$, and δ' is defined by setting $\delta'(a, [\gamma], b) = \delta(a, \gamma, b)$ whenever $\delta(a, \gamma, b) \neq \emptyset$.*

A chain-driven automaton $\mathcal{A}$ is structurally ambiguous *if $L([\mathcal{A}])$ contains two strings $\boldsymbol{x_1} \neq \boldsymbol{x_2}$ such that $\boldsymbol{x_1} \doteq \boldsymbol{x_2}$.*

For instance, consider a variant of the automaton of Example 1, where the context $(+, \#)$ is moved from the second row to the first, and a new body $q_e + q_+$ is added to the second column. This automaton can perform two structurally different computations for the input string $e + e + e$, namely, starting from configuration $\#q_e + q_e + q_e\#$:
$\#q_e + q_e + q_e\# \overset{\#\langle + \rangle +}{\longmapsto} \#q_+ + q_e\# \overset{\#\langle + \rangle \#}{\longmapsto} \#q_+\#$, i.e. where the string is assigned the structure $[[[e] + [e]] + [e]]$; and $\#q_e + q_e + q_e\# \overset{+\langle + \rangle \#}{\longmapsto} \#q_e + q_+\# \overset{\#\langle + \rangle \#}{\longmapsto} \#q_+\#$, where the structure is $[[e] + [[e] + [e]]]$.

Definition 4. *A chain-driven automaton $\mathcal{A} = \langle \Sigma, Q, C, \delta, F \rangle$ is* reduced *if every chain in C is used in some accepting computation.*

W.l.o.g. in what follows we consider only reduced automata.

We are going to see that chain-driven automata recognize CF languages, and can be seen as parsers for CF 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 rule of the grammar, and any state computed by the reduction function corresponds to the nonterminal at the l.h.s of the same rule.

Definition 5. *The chain $a\langle y \rangle b$ is a* grammatical chain *associated with G if there exists a derivation*

$$\#T\# \overset{*}{\underset{G}{\Longrightarrow}} \alpha\, aAb\, \beta \underset{G}{\Longrightarrow} \alpha\, a\gamma b\, \beta \tag{1}$$

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

Theorem 1. *Chain-driven automata recognize the class of CF languages.*

Proof. We prove that the language recognized by any chain-driven automaton can be generated by a grammar, and vice versa. We first need the concept of 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 labelled 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 $(V_N \cup \mathrm{OF}(V_N), C, \Longleftarrow)$ where the LTS states are all strings in operator forms, the labels are all 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 production of G and $\widehat{\gamma} = y$. A chain-driven automaton 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 $G = (V_N, \Sigma, P, S)$. Define the chain-driven automaton $\mathcal{A}_G = \langle \Sigma, Q, C_G, \delta, F \rangle$ where: $Q = V_N$; $F = S$ is the set of axioms; C_G is the set of grammatical chains associated with G; δ is defined by setting $B \in \delta(a, \gamma, b)$ for each production $B \to \gamma$ such that $a[\hat{\gamma}]b \in C_G$. Both G and $\mathcal{A}_G$ define the same LTS, except that for G the set of LTS states is $\mathrm{SF}(V_N)$ whereas for $\mathcal{A}$ the LTS states are the configurations of the automaton, i.e., strings $\#\gamma\# \in \mathrm{SF}(V_N)$. In particular, this means that the derivations $T \overset{*}{\Rightarrow} x$ of G with $T \in S$ are in bijection with the computations $\#x\# \overset{*}{\vdash\!\!\!-} \#T\#$ and this implies that $L(\mathcal{A}_G) = L(G)$.

Conversely, let $\mathcal{A} = \langle \Sigma, Q, C, \delta, F \rangle$. Define the grammar $G_{\mathcal{A}} = (V_N, \Sigma, P, S)$ where: $V_N = \Sigma \times Q \times \Sigma \; S = \{(\#, q, \#) \mid q \in F\}$ P is the set of productions $(a_0, q, a_{n+1}) \to \gamma$ where $q \in \delta(a_0, \gamma, a_{n+1})$. Both $\mathcal{A}$ and $G_{\mathcal{A}}$ define the same LTS, except that for $\mathcal{A}$ the LTS states are configurations $\#q_0 a_1 q_1 a_2 \ldots a_n q_n\#$ (any q_i may be missing), whereas for $G_{\mathcal{A}}$ the LTS states are written in the form $(\#, q_0, a_1)a_1(a_1, q_1, a_2)a_2 \ldots a_{n-1}(a_n, q_n, \#)$. In particular, this means that computations $\#x\# \overset{*}{\vdash\!\!\!-} \#q\#$ of $\mathcal{A}$ with $q \in F$ are in bijection with the derivations $(\#, q, \#) \overset{*}{\Rightarrow} x$ of G and this implies that $L(G_{\mathcal{A}}) = L(\mathcal{A})$. Notice that $C_G = C$. □

Traditional general CF parsers proceed always left to right and produce a unique representation of the syntax trees associated with the input string; our chain-driven automata, instead, may nondeterministically produce any bottom-up possible traversal of the grammar's trees, as it is illustrated by the parser of Example 1 and by its structurally ambiguous modification. Clearly, $\mathcal{A}$ is structurally unambiguous iff the equivalent grammar $G_{\mathcal{A}}$ defined in the proof of Theorem 1 is structurally unambiguous, and vice versa.

4 Locally chain-parsable languages

The following definitions formalize our intuitive idea of local parsability.

Definition 6. *A* local chain parser *(LCPA) is a chain-driven automaton such that, for every chain $a\langle y \rangle b$, the following condition holds: if $\widehat{\gamma} = y$, then every computation $\alpha\ a\gamma b\ \beta \overset{*}{\vdash\!\!\!-} \#q_F\#$ with $q_F \in F$ can be decomposed as $\alpha\ a\gamma b\ \beta \overset{*}{\vdash\!\!\!-} \alpha'\ a\gamma b\ \beta' \vdash\!\!\!- \alpha'\ aqb\ \beta' \overset{*}{\vdash\!\!\!-} \#q_F\#$ with suitable α', β', and q.*

Definition 7. *A grammar is* locally chain-parsable *(LCPG) if, for every grammatical chain $a\langle y \rangle b$, the following condition holds: if $\gamma \mathrel{\hat{=}} y$, then each derivation $\#T\# \overset{*}{\Longrightarrow} \alpha a\gamma b\beta$ with $T \in S$ can be decomposed as $\#T\# \overset{*}{\Longrightarrow} \alpha' aAb\ \beta' \Longrightarrow \alpha' a\gamma b\ \beta' \overset{*}{\Longrightarrow} \alpha\ a\gamma b\ \beta$. A language L is locally chain-parsable (LCPL) if it is generated by a LCPG.*

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

Theorem 2. *A language is LCPL if and only if it is recognized by a LCPA. An LCPA recognizes any string in linear time.*

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

Concerning time complexity, it is well-known that every grammar can be automatically transformed into a structurally equivalent invertible one [8]; thus, if we apply such a procedure to a locally parsable grammar, the corresponding local parser defined by Theorem 1 has a deterministic reduction function δ. It is therefore a simple exercise to derive a traditional deterministic pushdown automaton from a deterministic local parser: the former one simply restricts the set of computations of the latter one to the reverse of the rightmost visit of syntax trees. Thus, LCPL are (strictly) included in DCFL. □

Example 2. The following grammar G_1, which generates the same arithmetic expressions recognized by the chain-driven automaton of Example 1, is locally parsable.

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

In fact, consider a generic derivation such as $\#E\# \Rightarrow \#E + T\# \Rightarrow \#E + T + T\# \Rightarrow \#E + T * F + T\# \stackrel{*}{\Rightarrow} \#e + e * F + e\# \Rightarrow \#e + e * e + e\#$. 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 $N \to e$; the only $*$ in the context $(+, +)$ can only be produced through a rule with stencil $N \to N * N$; 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 $N \to N + N$.

Thus, a possible 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 could be fully deterministic if G were invertible.

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

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

The grammatical chains associated with G_2 are $\#\langle + \rangle\#$, $\#\langle + \rangle +$, $\#\langle e \rangle +$, $+\langle e \rangle +$, $+\langle e \rangle\#$, and $+\langle + \rangle\#$. For instance, chain $+\langle + \rangle\#$ is obtained by applying rule $X \to E + E$ in the last step of the following derivation: $\#X\# \stackrel{*}{\Rightarrow} \#E + E + X\# \Rightarrow \#E + E + E + E\#$

Now consider the following derivation: $\#Y\# \Rightarrow \#Y + E\# \Rightarrow \#Y + E + E\# \Rightarrow \#E + E + E + E\#$. The factor $\gamma = E + E$ occurs in context $(+, \#)$ but it is not reduced in any step of the derivation. Hence, G_2 is not locally parsable.

Informally, a possible parser, after having reduced all es to E, would be confronted with the sentential form $\#E + E + E + E\#$ and would not have any indication to decide whether to apply $X \to E + E$ reducing the last $+$, or $Y \to E + E$ reducing the first $+$.

4.1 Extended chains, conflicts and decidability of the LCP property

Both LCPA and LCPG give a unique structure to each string of their respective languages. To formalize this point, we first introduce the notion of extended chain, that generalizes Definition 1.

Definition 8. Structured strings *are special well-parenthesized strings over* $\Sigma \cup \{[,]\}$*, defined recursively as follows:*

- $y \in \Sigma^+$ *are* atomic *structured strings;*
- *if* $a_i \in \Sigma$ *and* $\boldsymbol{y_i} = \varepsilon$ *or* $\boldsymbol{y_i} = [\boldsymbol{v_i}]$ *for some structured strings* $\boldsymbol{v_i}$*, then* $\boldsymbol{y_0} a_1 \boldsymbol{y_1} a_2 \dots a_n \boldsymbol{y_n}$ *is a* composed *structured string if at least one* $\boldsymbol{y}_i$ *is different from* ε*.*

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

Any grammar or chain-driven automaton determines a set of xchains which have an important role w.r.t the local parsability property.

Definition 9. *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*, respectively, if there exist* $\boldsymbol{\gamma}$ *such that* $\widehat{\boldsymbol{\gamma}} = \boldsymbol{y}$ *and*

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

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

Remark 1. If $G_{\mathcal{A}}$ is the grammar equivalent to the chain-driven automaton $\mathcal{A}$, as defined in the proof of Theorem 1, then $\mathcal{X}_{G_{\mathcal{A}}} = \mathcal{X}_{\mathcal{A}}$; vice versa the chain-driven automaton $\mathcal{A}_G$ equivalent to a grammar G is such that $\mathcal{X}_{\mathcal{A}_G} = \mathcal{X}_G$.

Example 3. Consider grammar G_1 of Example 2. The sentential form $[E + [[e] * [e]]]$ is derived by the associated 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 the chain-driven automaton of Example 1; 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 string y they can generate/recognize, there is only one $G_1/\mathcal{A}$-xchain $\boldsymbol{y}$ such that $\widetilde{\boldsymbol{y}} = y$.

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" of s.

Definition 10. *An xchain* conflicts *with a chain* $a\langle y \rangle b$ *iff it can be decomposed as* $\boldsymbol{xaybz}$ *where* $\widetilde{\boldsymbol{y}} = y$ *and* $\boldsymbol{y} \notin [^+y]^+$*. A set* $\mathcal{X}$ *of xchains and a set* C *of chains are* conflictual *iff there is an xchain in* $\mathcal{X}$ *that conflicts with some chain in* C*.*

Example 4. 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 $\boldsymbol{x} = \varepsilon$, $\boldsymbol{y} = [+[$, and $\boldsymbol{z} = [+[+]]]]\#$. Otherwise, #[[[+]+]+]+]# does not conflict with #⟨+⟩+ since when #+ + occurs in the xchain (once, as a prefix), the first occurrence of symbol + is bracketed.

Example 5. By referring again to Example 2, with a little patience it can be verified that the set of G_1-xchains does not exhibit any conflict with $\mathcal{C}_{G_1}$, whereas $\mathcal{X}_{G_2}$ and $\mathcal{C}_{G_2}$ are conflictual. We next show that the property of having nonconflictual $\mathcal{X}_G$ and $\mathcal{C}_G$ is decidable for any grammar G (and is supported by an automatic tool.[3]) Also, G_1 is locally parsable, whereas G_2 is not. These remarks leads to the main property stated in Theorem 4.

Theorem 3. *The fact that $\mathcal{X}_G$ and $\mathcal{C}_G$ are nonconflictual is decidable for every grammar G; the fact that $\mathcal{X}_{\mathcal{A}}$ and C are nonconflictual is decidable for every automaton $\mathcal{A}$ driven by the set of chains C.*

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

$$P' = \{A \to [\alpha'] \mid A \to \alpha \in P, \text{ where } \alpha' = \alpha \text{ or } \alpha' \text{ is obtained from } \alpha \text{ by erasing some (or every) nonterminals}\} \cup \{T' \to \#[T]\# \mid T \in S\}.$$

It is easy to see that G' defines the language of all the G-xchains. For a grammatical chain c, we can define a regular language $R(c) = (\Sigma \cup \{[,]\})^* \cdot a \cdot \neg([^+ \cdot y \cdot]^+) \cdot b \cdot (\Sigma \cup \{[,]\})^*$ that is the language of all the possible xchains that conflict with c. Clearly, $R' := \bigcup_{c \in \mathcal{C}_G} R(c)$ is also a regular language. G is conflictual iff $L(G') \cap R' \neq \emptyset$; since $L(G') \cap R'$ is context-free, its emptiness problem is decidable.

The statement for the automaton follows from Theorem 1 and Remark 1. □

Theorem 4. *A chain-driven automaton $\mathcal{A}$ is a local parser if and only if $\mathcal{X}_{\mathcal{A}}$ and the set C of chains driving $\mathcal{A}$ are not conflictual. A grammar G is locally parsable if and only if $\mathcal{X}_G$ and $\mathcal{C}_G$ are not conflictual.*

Theorems 4 and 3 imply the following result.

Corollary 1. *The fact that a grammar is LCPG is decidable; the fact that a chain-driven automaton is LCPA is decidable.*

4.2 Basic properties of local chain-parsable languages

LCP grammars and parsers associate a unique structure with each generated/accepted string x; such a structure is represented by an xchain #[$\boldsymbol{x}$]# with $\boldsymbol{x} \doteq x$.

Theorem 5. *LCP grammars and parsers are structurally unambiguous.*

[3] https://github.com/bzoto/chainsaw.

Proof. Both properties can be proved similarly reasoning by contradiction. Assume that $\mathcal{A}$ is structurally ambiguous; then one can show that $\mathcal{X}_{\mathcal{A}}$ and the set of chains that drive $\mathcal{A}$ are conflictual. By Theorem 4 this means that $\mathcal{A}$ is not LCPA. For grammars the same results can be proved by using Corollary 1.

Definition 11. *Two LCPA $\mathcal{A}_1 = (\Sigma, Q_1, C_1, \delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, C_2, \delta_2, F_2)$ are* compatible *if $\mathcal{X}_{\mathcal{A}_1} \cup \mathcal{X}_{\mathcal{A}_2}$ and $C_1 \cup C_2$ are not conflictual. Two LCPL L_1 and L_2 are* compatible *if they are recognized by compatible LCPA.*

Theorem 6. *Let L be LCPL. Then its reversal L^R is LCPL.*

Theorem 7. *Let $\mathcal{A}_1 = (\Sigma, Q_1, C_1, \delta_1, F_1)$ and $\mathcal{A}_2 = (\Sigma, Q_2, C_2, \delta_2, F_2)$ be compatible chain-driven automata recognizing respectively L_1 and L_2. Then $L_1 \cup L_2$, $L_1 \cap L_2$, and $L_1 \setminus L_2$ are LCPL.*

Proof. W.l.o.g. we may assume that the set of states Q_1 and Q_2 are disjoint. Let $C = C_1 \cup C_2$, and $Q = (Q_1 \cup \{\bot, q_{\text{err}}\}) \times (Q_2 \cup \{\bot, q_{\text{err}}\})$, with $\bot, q_{\text{err}} \notin Q_1 \cup Q_2$. For each strings $\gamma \in \text{OF}(Q)$, 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.

We remark that independent moves of a LCPA (i.e., moves that reduce non-overlapping factors of the input string) can be applied in any order. For $\diamond \in \{\cap, \cup, \setminus\}$, the automaton for $L_1 \diamond L_2$ is given by $(\Sigma, Q, C, \delta, F_\diamond)$, where: $F_\cap = F_1 \times F_2$, $F_\cup = F_1 \times (Q_2 \cup \{q_{err}\}) \cup (Q_1 \cup \{q_{err}\}) \times F_2)$, $F_\setminus = F_1 \times (Q \setminus F_2 \cup \{q_{err}\})$. □

Corollary 2. *The inclusion problem for compatible LCPL is decidable.*

5 LCPL versus Operator-Precedence and Input-Driven languages

It is worthwhile to examine the LCPL as an outgrowth of the classical OPL [7], whose knowledge, both theoretically ([6,9] and for application to parallel parsing [2]), has much progressed in recent years. OPL is a subfamily of DCFL having the local parsability property and characterized by a bottom-up parser that is driven by three binary *precedence relations* between terminals of the grammar, defined as follows. Let α be a r.h.s.:

- a is *equal in precedence* to b ($a \doteq b$), if ab is a factor of $\widehat{\alpha}$;
- a *yields precedence* to b ($a \lessdot b$), if $b = \alpha_{\text{first}}$ and some s.f. contains $a\alpha$ as factor;
- a *takes precedence* over b ($a \gtrdot b$), if $a = \alpha_{\text{last}}$ and some s.f. contains αb as factor.

For a grammar to be an OPG, at most one relation may hold between a pair of terminals. For instance, the relations for G_1 in Example 2 are: $+ \gtrdot +, + \lessdot *, + \lessdot e, * \gtrdot +, * \gtrdot *, * \lessdot e, e \gtrdot +, e \gtrdot *$. The language $L_3 = \{c^n d^n \mid n > 0\}$, generated e.g. by $A \to cAd \mid cd$, necessarily has the relations $c \lessdot c, c \doteq d, d \gtrdot d$. Its reversal ${L_3}^R$ (generated by the mirror

grammar) has the symmetric relations $c \gtrdot c, d \doteq c, d \lessdot d$; therefore the language $L_4 = L_3 \cup L_3{}^R$ has precedence conflicts such as $c \lessdot c$ and $c \gtrdot c$ and is easily proved not to be an OPL.
Next, after proving that OPL is strictly contained within LCPL, we argue that the extra generative capacity of LCPG has practical value.

Theorem 8. *The OPL family is strictly contained within the LCPL family. Moreover every OPG is locally chain-parsable.*

Proof. To prove that every OPG G is LCP: the grammatical chains C_G are determined by the precedence relations of G, as follows: $a\langle c_1 \cdots c_k \rangle b \in C_G$ iff $a \lessdot c_1$, $c_i \doteq c_{i+1}$ for every $1 \leq i < k$, and $c_k \gtrdot b$.

Consider now any G's derivation of type $\#T\# \Rightarrow^* \alpha a \gamma b \beta$ with $\widehat{\gamma} = y$, $a\langle y \rangle b \in C_G$; since for each pair of terminals at most one precedence relation holds, it is necessarily $a \lessdot \gamma_{\text{first}}$, $\gamma_{\text{last}} \gtrdot b$ 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 $\gamma_{1\text{last}} \gtrdot \gamma_{2\text{first}}$).
The strict inclusion $OPL \subset LCPL$ is witnessed by the previous language, $L_4 = \{c^n d^m \mid n \geq 1\} \cup \{d^n c^n \mid n \geq 1\}$, which is recognized by the obviously local automaton driven by the chains: $\#\langle cd \rangle \#$, $\#\langle dc \rangle \#$, $c\langle cd \rangle d$, $d\langle dc \rangle c$. □

Useful generative capacity of LCPG. LCPG permit to define relevant syntactic constructs beyond the capacity of OPG. As an argument we present a well-known practical construct: arithmetic expressions containing both unary and binary minus signs, which notoriously introduce precedence conflicts. To keep the example small, the next grammar features only subtraction and multiplication and e as operand, but other operators and parentheses would be straightforward to add. E is the only axiom:

$$E \to e \mid -e \mid eX \mid -eX \mid E - e \mid E - -e \mid E - eX \mid E - -eX$$
$$X \to *e \mid *-e \mid *eX \mid *-eX$$

To prove that this grammar has the LCP property, we have used the previously mentioned tool. While traditional precedence parsers are forced to use some trick (like diversifying the two types of minus in the preceding phase of lexical analysis), our LCPG permits a pure syntactic approach.

6 Related work and conclusions

In addition to the mentioned relations of our work to the IDL, other classical lines of research present conceptual analogies to be briefly presented, and have somewhat inspired our effort. Brevity forces us to limit explanations and citations.

The NTS languages [5] are defined by the nonterminal separation property. They enjoy the local parsability property in the following sense: if a factor (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 factor can be reduced to the same symbol. NTS languages, however, are not input-driven, because they rely on the presence of nonterminals for localizing the position of a reduction.

Moving to another research area, the local parsability property is remindful of the confluence property of Church-Rosser languages (also called McNaughton languages): they are defined by string rewriting rules [10,4]. 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.

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

This class properly extends the known input-driven classes, preserving important closure properties, and maintains the decidability of the containment problem. This increased generative power can be exploited to define practical languages and to obtain efficient parallel parsers, thus extending the algorithm presented in [3] for OPL and replicating the obtained benefits in terms of time complexity and speed up. Further closure properties, in particular concatenation and Kleene's star, are still to be investigated.

The present paper represents a new step in the long term path towards a general theory of local deterministic parsing.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56(3) (2009)
2. Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: Parallel parsing of operator precedence grammars. IPL (2013)
3. Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: Parallel parsing made practical. SCP (2015), under revision
4. Beaudry, M., Holzer, M., Niemann, G., Otto, F.: McNaughton families of languages. TCS 290(3), 1581–1628 (2003)
5. Boasson, L., Sénizergues, G.: NTS languages are deterministic and congruential. JCSS 31(3), 332–342 (1985)
6. Crespi Reghizzi, S., Mandrioli, D.: Operator Precedence and the Visibly Pushdown Property. JCSS 78(6), 1837–1867 (2012)
7. Floyd, R.W.: Syntactic Analysis and Operator Precedence. JACM 10(3), 316–333 (1963)
8. Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley (1978)
9. Lonati, V., Mandrioli, D., Panella, F., Pradella, M.: Operator precedence languages: Their automata-theoretic and logic characterization. SICOMP (2015), to appear
10. McNaughton, R., Narendran, P., Otto, F.: Church-Rosser Thue systems and formal languages. JACM 35(2), 324–344 (1988)
11. Mehlhorn, K.: Pebbling mountain ranges and its application of DCFL-recognition. In: Automata, languages and programming (ICALP-80). LNCS, vol. 85, pp. 422–435 (1980)