# Boosting Parallel Parsing through Cyclic Operator Precedence Grammars

Michele Chiari
michele.chiari@tuwien.ac.at
TU Wien
Vienna, Austria

Michele Giornetta
Dino Mandrioli
Matteo Pradella*
michele.giornetta@mail.polimi.it
dino.mandrioli@polimi.it
matteo.pradella@polimi.it
DEIB, Politecnico di Milano
Milano, Italy

## Abstract

Deterministic parsing of tree-structured data is usually performed sequentially left-to-right. Recently however, also motivated by the need to process extremely large data sets, a parallel version thereof has been devised which, thanks to the theoretical features of *operator precedence languages* (OPL) particularly well-suited to split the input into separate chunks, provided high improvements w.r.t. traditional sequential parsing. Further investigation pointed out a restriction imposed on the OPL formalism that prevents from fully exploiting parallelism and proposed an improvement of the original algorithm which proved effective in many practical cases. Stimulated by the above contribution here we remove the mentioned restriction on OPL and build a new parallel parser generator based thereon. We conducted a comparative experimentation among the three parallel algorithms that showed a consistent further improvement w.r.t. both the previous ones (with an exception in the case of purely sequential execution). Based on these early results, we believe that the horizon of parallel parsing large tree-structured data promises dramatic gains of efficiency in the analysis of this fundamental data structure.

***CCS Concepts:*** • **Software and its engineering** → **Parsers**; **Syntax**; *Context specific languages.*

***Keywords:*** Parallel parsing, operator precedence, regular expressions

**ACM Reference Format:**
Michele Chiari, Michele Giornetta, Dino Mandrioli, and Matteo Pradella. 2025. Boosting Parallel Parsing through Cyclic Operator Precedence Grammars. In *Proceedings of 18th ACM SIGPLAN International Conference on Software Language Engineering (SLE'25).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Within the countless number of applications based on the tree data structure and the *context-free language (CFL)* formalism, parsing, i.e., the process of building tree structure(s) associated with a given input sentence, say $x$, usually and herewith named *syntax-tree (ST)* of $x$, represents a fundamental part of any automatic elaboration of the string representing data input and accounts for a major fraction of its computational effort [21]:

> It becomes clear that even for complex queries that involve joins and aggregations, the total cost of a query is dominated (> 80%) by parsing the raw JSON data.

In fact, parsing is also the necessary pre-elaboration for most of the semantic operations that can be applied to data input, such as, e.g., compiling, interpreting, data-filtering, data property verification, ...; a major approach to such a syntax-driven semantic elaboration is based on attribute-based semantics, where the semantics to be computed is defined as an attribute associated with the internal nodes of the ST.

The traditional techniques for deterministic parsing of data formalized as CFLs, such as those based on the LR or LL subclasses of CFLs, are intrinsically left-to-right, which makes them not amenable to exploit the features offered by modern HW and SW parallel architectures.

Important subclasses of CFLs, however, exhibit the so-called *local parsability property*, i.e., the fact that a fragment of any input sentence $x$ can be parsed, and the corresponding fragment of the ST whose frontier is $x$ can be built, independently of its context. In the literature there exists at least one such language family that possesses the locality property

[*] Also with IEIIT – Consiglio Nazionale delle Ricerche.

*SLE'25, Koblenz, Germany*

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

that has been exploited to build highly performant parallel and incremental parsers, namely *operator precedence languages (OPLs)* [2–4][1]. Furthermore, recent and less recent investigations [9, 10, 22] have shown that OPLs enjoy algebraic and logic properties that make them amenable for automatic semantic elaboration spanning form traditional compilation and interpretation to sophisticated *model checking (MC)* algorithms [6].

The key feature that makes OPLs well amenable for efficient parsing and compilation is that the ST of a sentence is determined exclusively by three binary precedence relations over the terminal alphabet that are easily pre-computed from the grammar productions. For example: the arithmetic sentence $a + b \times c$ does not make manifest the natural structure $(a + (b \times c))$, but the latter is implied by the fact that the plus operator yields precedence to the times.

The parallel parser generator PAPAGENO has proven quite effective in reducing time and space complexity w.r.t. traditional sequential parsing [2]. Recently, however, it has been observed [20] that in some cases the OPL-based parser misses important parallelization opportunities. As an example, consider the structure of non-parenthesized arithmetic expressions, say, a sequence of sums: normally OPLs associate to them STs of the type depicted in Figure 1 (left), which corresponds to a left-associative semantics of the sum; the same applies to all four basic arithmetic operations; notice, however that, whereas for subtraction and division the semantics is necessarily associative to the left, not so for addition and multiplication which, thanks to their commutative property can be associated both to the left and to the right indifferently.

As a consequence of such a structure, however, the parsing of a sequence of plus operators must necessarily proceed left-to-right, thus missing the opportunity to split a long sequence into chunks to be computed separately so that subsequently the partial results can be added together to obtain the complete sum. This can be obtained by giving the sequence of pluses a "flat structure" as suggested in Figure 1 (center). For reasons that will be explained later, however, such a structure cannot be defined by means of traditional OP grammars (OPGs); thus, in [20] an ad-hoc algorithm has been developed and implemented that allows to build STs such as the one given in Figure 1 (right) (despite the risk of introducing ambiguities in the grammar). As expected, the new algorithm overtook the original PAPAGENO, mainly when parsing long sequences of data with flat structure. Notice that such a circumstance occurs rarely in computer programs, certainly not in arithmetic expressions, but more frequently in sequences of switch-like statements; rather, it is much more frequent in the use of data description languages such as JSON.

We realized that the above inefficiency of parallel parsing based on traditional OPLs is due to a hypothesis on the operator precedences that has been often assumed to simplify the analysis of algebraic and logic properties of this language family. Such a hypothesis slightly affects the generative power of OPGs [11, 22] but, so far, the loss turned out to be limited to a few counterexamples of pure mathematical interest. The above recent experience, instead, proved the opposite, so that we investigated the consequences of removing the restrictive hypothesis on precedence relations.

As a first result, in [8] we generalized the definition of OPGs to give them the full power necessary to produce structures such as that of Figure 1 (center) (notice that STs describing such a structure are unranked). We also proved that the generalized version of OPGs maintains all the algebraic and logic properties already proved for the restricted version thereof.

Then, based on the theoretical foundation provided by [7, 8] we built a new parallel parser generator that overtakes the above weakness. In this paper, we report on the design and implementation of the new parallel parser generator and on the experimentation we carried over, showing that it overtakes, sometimes in a dramatic way, both the original PAPAGENO and the new tool by [20]. Section 2 provides the necessary background on formal language terminology and the "historical OPLs"; Section 3 summarizes the essential aspects of the theoretical revision of OPGs and their accepting automata reported in [7, 8]; Section 4 describes the architecture of the new parser generator and Section 5 offers some details on its implementation through the GO language, to foster reproducibility of the experimental results, which in turn are reported in Section 6. Section 7 concludes with suggestions to optimize the performance of the present tool and to build new applications based thereon, mainly in the field of semantic analysis and program property verification.

## 2 Background

We assume some familiarity with the classical literature on formal language and automata theory, e.g., [18, 23]. Here, we just list and explain our notations for the basic concepts we use from this theory. The terminal alphabet is usually denoted by $\Sigma$, and the empty string is $\varepsilon$. The character $\# \notin \Sigma$ is used as *delimiter*, and we define $\Sigma_\# = \Sigma \cup \{\#\}$.

A *context-free (CF) grammar* is a tuple $G = (\Sigma, V_N, P, S)$ where $\Sigma$ and $V_N$, with $\Sigma \cap V_N = \emptyset$, are resp. the terminal and the nonterminal alphabets, the total alphabet is $V = \Sigma \cup V_N$, $P \subseteq V_N \times V^*$ is the rule (or production) set, and $S \subseteq V_N, S \neq \emptyset$, is the axiom set. For a generic rule, denoted as $A \longrightarrow \alpha$, where $A$ and $\alpha$ are resp. called the left/right hand sides (lhs/rhs), the following forms are relevant: *axiomatic* $A \in S$; *terminal* $\alpha \in \Sigma^+$; *empty* $\alpha = \varepsilon$; *renaming* $\alpha \in V_N$; *operator* $\alpha \notin V^* V_N V_N V^*$, i.e., at least one terminal is interposed between any two nonterminals occurring in $\alpha$.

[1] Parallel and incremental parsing [4, 13] are two strictly connected ways to exploit the locality property.

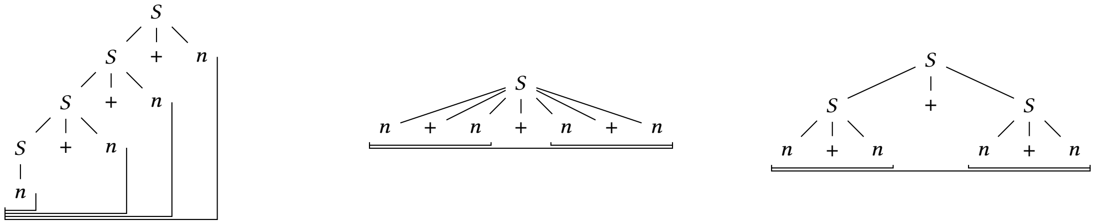


**Figure 1.** Left-associative syntax tree (left) vs equal-level (center) and symmetric (right) ones of the plus operator. The left syntax tree imposes a sequential left-to-right parsing and semantic processing whereas the center one can be –and the right one is– split onto several branches to be partially processed independently from each other and further aggregated.

A grammar is *backward deterministic (BD)* if $(B \longrightarrow \alpha, C \longrightarrow \alpha \in P)$ implies $B = C$. If all rules of a grammar are in operator form, it is called an *operator grammar* or O-grammar. The symbols $\underset{G}{\Longrightarrow}$, $\overset{*}{\underset{G}{\Longrightarrow}}$, $\overset{+}{\underset{G}{\Longrightarrow}}$ denote, respectively, an *immediate derivation*, its reflexive and transitive closure, its transitive closure. The subscript $G$ will be omitted whenever clear from the context. We give also for granted the notion of *syntax tree (ST)*. The *frontier* of a syntax tree is the ordered left-to-right sequence of the leaves of the tree.

The *language* defined by $G$, said $L(G)$, is $\{w \mid w \in \Sigma^*, A \overset{*}{\underset{G}{\Longrightarrow}} w \land A \in S\}$. Two grammars defining the same language are *equivalent*. Two grammars generating the same set of syntax trees, up to a renaming of internal nodes, are *structurally equivalent*.

*From now on, w.l.o.g., we exclusively deal with O-grammars without renaming and empty rules with the only exception that, if $\varepsilon$ is part of the language, there is a unique empty rule whose lhs is an axiom that does not appear in the rhs of any production. In fact, this is a well-known normal form for CF grammars [1, 18].*

We now define operator precedence grammars (OPGs). Intuitively, OPGs are O-grammars whose parsing is driven by three *precedence relations* (PR), called *equal*, *yield* and *take*, included in $\Sigma_\# \times \Sigma_\#$. They are defined in such a way that every rhs occurring within a ST is enclosed within a pair *yield–take*, and all terminals in between are separated by an *equal* (nonterminals are irrelevant for precedence relations) so that the rhs can be *reduced* to a corresponding lhs by a typical bottom-up parsing.

**Definition 2.1** ([12]). Let $G = (\Sigma, V_N, P, S)$ be an O-grammar. Let $a, b$ denote elements in $\Sigma$, $A, B$ in $V_N$, $C$ either an element of $V_N$ or the empty string $\varepsilon$, and $\alpha, \beta$ range over $V^*$. The *left and right terminal sets* of nonterminals are respectively:

$$\mathcal{L}_G(A) = \left\{ a \in \Sigma \mid \exists C : A \overset{*}{\underset{G}{\Longrightarrow}} Ca\alpha \right\} \text{ and}$$

$$\mathcal{R}_G(A) = \left\{ a \in \Sigma \mid \exists C : A \overset{*}{\underset{G}{\Longrightarrow}} \alpha aC \right\}.$$

The *operator precedence (OP) relations* are defined over $\Sigma_\# \times \Sigma_\#$ as follows:

**Equal in precedence** $a \doteq b \Leftrightarrow \exists A \longrightarrow \alpha aCb\beta \in P$.
**Takes precedence** $a \gtrdot b \Leftrightarrow \exists A \longrightarrow \alpha Bb\beta \in P, a \in \mathcal{R}(B)$; $a \gtrdot \# \Leftrightarrow a \in \mathcal{R}(B), B \in S$.
**Yields precedence** $a \lessdot b \Leftrightarrow \exists A \longrightarrow \alpha aB\beta \in P, b \in \mathcal{L}(B)$; $\# \lessdot b \Leftrightarrow b \in \mathcal{L}(B), B \in S$.

The OP relations are collected into a $|\Sigma_\#| \times |\Sigma_\#|$ array, called the *operator precedence matrix* of the grammar, $OPM(G)$: for each (ordered) pair $(a, b) \in \Sigma_\# \times \Sigma_\#$, $OPM_{a,b}(G)$ contains the OP relations holding between $a$ and $b$.

An OPM is *conflict-free* iff $\forall a, b \in \Sigma_\#$, $0 \leq |M_{a,b}| \leq 1$. A conflict-free OPM is *total* or *complete* iff $\forall a, b \in \Sigma_\#$, $|M_{a,b}| = 1$. If $M_{\#,\#}$ is not empty, $M_{\#,\#} = \{\doteq\}$. An OPM is $\doteq$*-acyclic* if the transitive closure of the $\doteq$ relation over $\Sigma \times \Sigma$ is irreflexive.

We extend the set inclusion relations and the Boolean operations in the obvious cell-by-cell way, to any two matrices having the same terminal alphabet. Two matrices are *compatible* iff their union is conflict-free.

**Definition 2.2** (Operator precedence grammar). A grammar $G$ is an *operator precedence grammar* (OPG) iff the matrix $OPM(G)$ is conflict-free. An OPG is $\doteq$*-acyclic* if $OPM(G)$ is so. An *operator precedence language (OPL)* is a language generated by an OPG.

Figure 2 (left) displays an OPG, $G_{AE}$, which generates simple, unparenthesized arithmetic expressions and its OPM (center). The left and right terminal sets of $G_{AE}$'s nonterminals $E$, $T$ and $F$ are, resp.: $\mathcal{L}(E) = \{+, \times, n\}$, $\mathcal{L}(T) = \{\times, n\}$, $\mathcal{L}(F) = \{n\}$, $\mathcal{R}(E) = \{+, \times, n\}$, $\mathcal{R}(T) = \{\times, n\}$, and $\mathcal{R}(F) = \{n\}$.
*Remark.* If the relation $\doteq$ is acyclic, then the length of the rhs of any rule of $G$ is bounded by the length of the longest $\doteq$-chain in $OPM(G)$.

We first illustrate through a simple example how a conflict-free OPM drives the deterministic parsing of a string to build its associated ST (if any); those strings whose parsing succeeds to produce a ST for them are said *compatible* with the given OPM and are a *universe* of sentences with their unique corresponding STs. Any OPG whose OPM is a subset of the given one define a language that is a subset of the universe. We refer the reader to previous literature for a thorough description of OP parsing.

$$G_{AE}: \quad \begin{array}{l} S = \{E\} \\ E \longrightarrow E + T \mid T \times F \mid n \\ T \longrightarrow T \times F \mid n \\ F \longrightarrow n \end{array}$$

|   | $+$ | $\times$ | $n$ | $\#$ |
|---|---|---|---|---|
| $+$ | $\gtrdot$ | $\lessdot$ | $\lessdot$ | $\gtrdot$ |
| $\times$ | $\gtrdot$ | $\gtrdot$ | $\lessdot$ | $\gtrdot$ |
| $n$ | $\gtrdot$ | $\gtrdot$ |  | $\gtrdot$ |
| $\#$ | $\lessdot$ | $\lessdot$ | $\lessdot$ |  |

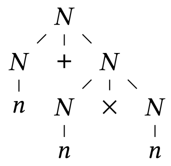


**Figure 2.** $G_{AE}$ (left), its OPM (center), and the syntax tree of $n + n \times n$ (right).

**Example 2.3.** Consider the $OPM(G_{AE})$ of Figure 2 and the string $n + n \times n$. Display all precedence relations holding between consecutive terminal characters, *including the relations with the delimiters* # as shown here:

$$\# \lessdot n \gtrdot + \lessdot n \gtrdot \times \lessdot n \gtrdot \#$$

Each pair $\lessdot, \gtrdot$ (with no further $\lessdot, \gtrdot$ in between) includes a *possible* rhs of a production of *any OPG* sharing the OPM with $G_{AE}$, not necessarily a $G_{AE}$ rhs. Thus, as it happens in typical bottom-up parsing, we replace —*possibly in parallel*— each string included within the pair $\lessdot, \gtrdot$ with a *dummy nonterminal* $N$; this is because nonterminals are irrelevant for OPMs. The result is the string $\#N + N \times N\#$. Next, we compute again the precedence relations between consecutive terminal characters by *ignoring nonterminals*: the result is $\# \lessdot N + \lessdot N \times N \gtrdot \#$.

This time, there is only one pair $\lessdot, \gtrdot$ including a potential rhs determined by the OPM. Again, we replace the pattern $N \times N$, with the dummy nonterminal $N$; notice that there is no doubt about associating the two $N$ to the $\times$ rather than to one of the adjacent symbols: if we replaced, say, just the $\times$ with an $N$ we would obtain the string $N + NNN$ which cannot be derived by an O-grammar. By iterating the above procedue we finally end up with the string $\#N\#$. The result of the whole bottom-up reduction procedure is synthetically represented by the *syntax tree* of Figure 2 (right) which shows the precedence of the multiplication operation over the additive one in traditional arithmetics. It also suggests a natural association to the left of both operations.

The tree of Figure 2 has been obtained —uniquely and deterministically— by using exclusively the OPM, not the grammar $G_{AE}$ although the string $n + n \times n \in L(G_{AE})$. The above procedure, however, could be easily adapted to become an algorithm that produces a new syntax tree whose internal nodes are labeled by $G_{AE}$'s nonterminals. Such an algorithm could be made deterministic by transforming $G_{AE}$ into a structurally equivalent BD grammar sharing the same OPM.

Obviously, all sentences of $L(G_{AE})$ can be given a syntax tree by $OPM(G_{AE})$, but there are also strings in the universe defined by its OPM, e.g. $+++$, that are not in $L(G_{AE})$. Notice also that, in general, not every string in $\Sigma^*$, e.g. $nn$, belongs to the universe of the OPM.

Thus, an OPG selects a set of STs within the universe defined by its OPM; a similar selection can be operated by an *operator precedence automaton (OPA)* [22].

**Definition 2.4** (Operator precedence automaton (OPA))**.** A nondeterministic *OPA* is given by a tuple: $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ where: $\Sigma$ is the terminal alphabet, $M$ an OPM thereon, $Q$ a set of states (disjoint from $\Sigma$), $I \subseteq Q$ a set of initial states, $F \subseteq Q$ a set of final states, $\delta$, the transition function, is a triple of functions $\delta_{\text{shift}} : Q \times \Sigma \rightarrow \wp(Q)$, $\delta_{\text{push}} : Q \times \Sigma \rightarrow \wp(Q)$, $\delta_{\text{pop}} : Q \times Q \rightarrow \wp(Q)$.

We represent a nondeterministic OPA by a graph with $Q$ as the set of vertices and $\Sigma \cup Q$ as the set of edge labelings. We write $p \xrightarrow{a} q$ iff $q \in \delta_{\text{push}}(p, a)$, $p \overset{a}{\dashrightarrow} q$ iff $q \in \delta_{\text{shift}}(p, a)$, and $q \overset{p}{\Longrightarrow} r$ iff $r \in \delta_{\text{pop}}(q, p)$.

To define the semantics of the automaton, we introduce some notations. We use letters $p, q, p_i, q_i, \dots$ to denote states in $Q$. Let $\Gamma$ be $\Sigma \times Q$ and let $\Gamma'$ be $\Gamma \cup \{\bot\}$; we denote symbols in $\Gamma'$ as $[a,\ q]$ or $\bot$. We set $symbol([a,\ q]) = a$, $symbol(\bot) = \#$, and $state([a,\ q]) = q$. Given a string $\Pi = \bot \pi_1 \pi_2 \dots \pi_n$, with $\pi_i \in \Gamma$ , $n \geq 0$, we set $symbol(\Pi) = symbol(\pi_n)$, including the particular case $symbol(\bot) = \#$.

A *configuration* of an OPA is a triple $C = \langle \Pi,\ q,\ w \rangle$, where $\Pi \in \bot\Gamma^*$, $q \in Q$ and $w \in \Sigma^*\#$. The first component represents the contents of the stack, the second component represents the current state of the automaton, while the third component is the part of input still to be read.

A *computation* or *run* of the automaton is a finite sequence of *moves* or *transitions* $C_1 \vdash C_2$; there are three kinds of moves, depending on the precedence relation between the symbol on top of the stack and the next symbol to read:

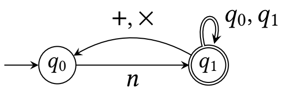
**push move:** if $symbol(\Pi) \lessdot a$ then $\langle \Pi,\ p,\ ax \rangle \vdash \langle \Pi[a,\ p],\ q,\ x \rangle$, with $q \in \delta_{\text{push}}(p, a)$;

**shift move:** if $a \doteq b$ then $\langle \Pi[a,\ p],\ q,\ bx \rangle \vdash \langle \Pi[b,\ p],\ r,\ x \rangle$, with $r \in \delta_{\text{shift}}(q, b)$;

**pop move:** if $a \gtrdot b$ then $\langle \Pi[a,\ p],\ q,\ bx \rangle \vdash \langle \Pi,\ r,\ bx \rangle$, with $r \in \delta_{\text{pop}}(q, p)$.

Shift and pop moves are never performed when the stack contains only $\bot$.

Push and shift moves update the current state of the automaton according to the transition functions $\delta_{\text{push}}$ and $\delta_{\text{shift}}$, respectively: push moves put a new element on the top of the stack consisting of the input symbol together with the current state of the automaton, whereas shift moves update the top element of the stack by changing its input symbol only. The pop move removes the symbol on the top of the stack, and the state of the automaton is updated by $\delta_{\text{pop}}$ on



| stack | state | current input |
|---|---|---|
| $\perp$ | $q_0$ | $n + n \times n\#$ |
| $\perp[n,\ q_0]$ | $q_1$ | $+n \times n\#$ |
| $\perp$ | $q_1$ | $+n \times n\#$ |
| $\perp[+,\ q_1]$ | $q_0$ | $n \times n\#$ |
| $\perp[+,\ q_1][n,\ q_0]$ | $q_1$ | $\times n\#$ |
| $\perp[+,\ q_1]$ | $q_1$ | $\times n\#$ |
| $\perp[+,\ q_1][\times,\ q_1]$ | $q_0$ | $n\#$ |
| $\perp[+,\ q_1][\times,\ q_1][n,\ q_0]$ | $q_1$ | $\#$ |
| $\perp[+,\ q_1][\times,\ q_1]$ | $q_1$ | $\#$ |
| $\perp[+,\ q_1]$ | $q_1$ | $\#$ |
| $\perp$ | $q_1$ | $\#$ |

**Figure 3.** An OPA (top) defined on the OPM of Figure 2 and an example of computation for $L(G_{AE})$ (bottom).

the basis of the pair of states consisting of the current state of the automaton and the state of the removed stack symbol; notice that in this move the input symbol is used only to establish the $\gtrdot$ relation and it remains available for the following move.

A configuration $\langle \perp,\ q_I,\ x\# \rangle$ is *initial* if $q_I \in I$; a configuration $\langle \perp,\ q_F,\ \# \rangle$ is *accepting* if $q_F \in F$. The language accepted by the automaton is:
$L(\mathcal{A}) = \left\{ x \mid \langle \perp,\ q_I,\ x\# \rangle \vdash^* \langle \perp,\ q_F,\ \# \rangle, q_I \in I, q_F \in F \right\}.$

**Example 2.5.** The OPA depicted in Figure 3 (top) based on the same OPM as that in Figure 2 accepts the language $L(G_{AE})$. The same figure (bottom) also shows an accepting computation on input $n + n \times n$.

## 3 Cyclic OPGs and their properties

Previous research [9, 10] has shown that *some, but not all*, of the algebraic properties of OPLs depend critically on the $\doteq$-acyclicity hypothesis. In particular, whereas it is not needed to prove closure w.r.t. boolean operations and concatenation, it is necessary for the closure w.r.t. Kleene *. This is due to the fact that without such a hypothesis the rhs of an OPG have an unbounded length but cannot be infinite: e.g., no OPG can generate the language $\{a, b\}^*$ if $a \doteq b$ and $b \doteq a$. In most cases cycles of this type can be "broken" as it has been done up to now, e.g., to avoid the $+ \doteq +$ relation in arithmetic expressions by associating the operator indifferently to the right or to the left. From a theoretical point of view, the $\doteq$-acyclicity hypothesis affects the expressive power of OPGs; thus, the OPL family as generated by OPGs is strictly included within that accepted by OPAs.[2] We assumed so far the $\doteq$-acyclicity hypothesis to keep the notation as simple as possible so that the two formalisms are equivalent.

Recently, however, it has been observed [20] that such a restriction may hamper the benefits achievable by the parallel compilation techniques that exploit the local parsability property of OPLs: e.g., with reference to Figure 1, the acyclicity hypothesis imposes an associative structure of type (left) whereas the flat structure of type (center) demands for a cyclic $+ \doteq +$ [3]. Thus, we finally augmented traditional OPGs by avoiding the $\doteq$-acyclicity hypothesis and introducing an extended version of the OPG's rhs; we proved that all the algebraic properties of previously proved for OPLs still hold and that the extended OPGs now reach the full power of OPAs [8]. Next we briefly recall the referred result which is the basis on top of which we built the present new parallel parser generator[4].

**Definition 3.1** (Cyclic OP Grammar (C-OPG))**.** A $^+$-O-expression on $V^*$ is an expression obtained from the elements of $V$ by iterative application of concatenation and the $^+$ operator[5], provided that any substring thereof has no two adjacent nonterminals; for convenience, and w.l.o.g., we assume that all subexpressions that are argument of the $^+$ operator are terminated by a terminal character.

A Cyclic O-grammar (C-OG) is an O-grammar whose production rhs are $^+$-O-expressions. For a rule $A \longrightarrow \alpha$ of a C-OG, the $\underset{G}{\Longrightarrow}$ (immediate derivation) relation is defined as $\beta A \gamma \Longrightarrow \beta \zeta \gamma$ iff $\zeta$ is a string belonging to the language defined by the $^+$-O-expression $\alpha$, $L(\alpha)$. The $\doteq$ relation is redefined as $a \doteq b$ iff $\exists A \longrightarrow \alpha \land \exists \zeta = \eta a C b \theta \mid (C \in V_N \cup \{\varepsilon\} \land \zeta \in L(\alpha))$. The other relations remain defined as for non-cyclic O-grammars. A C-OG is a C-OPG iff its OPM is conflict-free.

As a consequence of the definition of the immediate derivation relation for C-OPGs the STs derived therefrom can be unranked, i.e., their internal nodes may have an unbounded number of children. For instance, it is immediate to verify that, by replacing the rule $E \longrightarrow E + T$ of $G_{AE}$ with the rule $E \longrightarrow (T+)^+T$ we obtain, for a sequence of + operations, the structure of Figure 1(center) instead of that of part (left).

Two symmetric theorems [7, 8] prove that C-OPGs and OPAs are fully equivalent even without assuming the $\doteq$-acyclicity hypothesis. For our purposes, here it suffices to report only the following one which is the core on the basis of which our new parallel parser is built.

**Theorem 3.2** (From C-OPGs to OPAs)**.** *For any C-OPG defined on an alphabet $\Sigma$ and OPM $M$ an equivalent OPA can be effectively built.*

A nondeterministic OPA[6] $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ from a given C-OPG $G$ with the same precedence matrix $M$ as $G$ is

[2] The language $\{a^n(bc)^n\} \cup \{b^n(ca)^n\} \cup \{c^n(ab)^n\} \cup (abc)^+$ cannot be generated by an OPG because the $a \doteq b \doteq c \doteq a$ relations are necessary [11], but it is accepted by OPAs.

[3] [20] instead adopts a structure of type (right) which generates precedence conflicts to be managed through an *ad hoc* solution.

[4] More details and explanations can be found in [7].

[5] For our purposes $^+$ is more convenient than $^*$ without affecting the generality.

[6] Any nondeterministic OPA can be transformed into a deterministic one at the cost of quadratic exponential increase in the size of the state space [22].

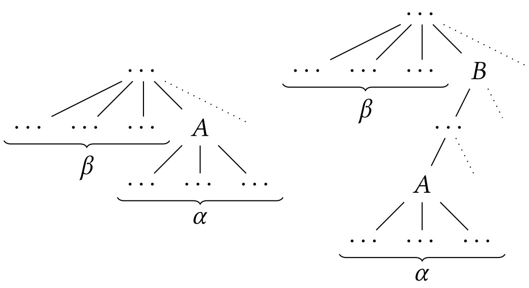


**Figure 4.** When parsing $\alpha$, the prefix previously under construction is $\beta$.

built in such a way that a successful computation thereof corresponds to building bottom-up a syntax tree of $G$: the automaton performs a push transition when it reads the first terminal of a new rhs; it performs a shift transition when it reads a terminal symbol inside a rhs, i.e. a leaf with some left sibling leaf. It performs a pop transition when it completes the recognition of a rhs, then it guesses (nondeterministically) the nonterminal at the lhs. Each state contains two pieces of information: the first component is the prefix of the rhs under construction, whereas the second component is used to recover the rhs *previously under construction* whenever all rhs nested below have been completed (see Figure 4).

Let $\hat{P}$ be the set of rhs $\gamma$ where all $^+$ and related parentheses have been erased. Let $\tilde{P}$ be the set of strings $\tilde{\gamma} \in V^+$ belonging to the language of some rhs $\gamma$ of $P$ that is inductively defined as follows: if $(\eta)^+$ is a subexpression of $\gamma$ such that $\eta$ is a single string $\in V^+$ then $\tilde{\eta} = \{\eta, \eta\eta\}$; if $\eta = \alpha_1(\beta_1)^+\alpha_2(\beta_2)^+ \dots \alpha_n$ where $\alpha_i \in V^*$, then $\tilde{\eta} = \{\eta_1, \eta_1\eta_1\}$ where $\eta_1 = \alpha_1\tilde{\beta}_1\alpha_2\tilde{\beta}_2 \dots \alpha_n$.

E.g., let $\eta$ be $(Ba(bc)^+)^+$; then $\hat{\eta} = \{Babc\}$ and $\tilde{\eta} = \{Babc, BBabcbc, BBabcBabc, BabcbcBabc, BabcBabcbc, BabcbcBabcbc\}$.

Let $\mathbb{P} = \{\alpha \in V^*\Sigma \mid \exists A \longrightarrow \eta \in P \wedge \exists\beta(\alpha\beta \in \tilde{\eta})\}$ be the set of prefixes, ending with a terminal symbol, of strings $\in \tilde{P}$; define $\mathbb{Q} = \{\varepsilon\} \cup \mathbb{P} \cup V_N$, $Q = \mathbb{Q} \times (\{\varepsilon\} \cup \mathbb{P})$, $I = \{\langle\varepsilon, \varepsilon\rangle\}$, and $F = S \times \{\varepsilon\} \cup \{\langle\varepsilon, \varepsilon\rangle \text{ if } \varepsilon \in L(G)\}$. Note that $|\mathbb{Q}| = 1 + |\mathbb{P}| + |V_N|$ is $O(m^h)$ where $m$ is the maximum length of the rhs in $P$, and $h$ is the maximum nesting level of $^+$ operators in rhs; therefore $|Q|$ is $O(m^{2h})$.

The transition functions are defined by the following formulas, for $a \in \Sigma$ and $\alpha, \alpha_1, \alpha_2 \in \mathbb{Q}$, $\beta, \beta_1, \beta_2 \in \{\varepsilon\} \cup \mathbb{P}$, and where for any expression $\xi$, $\bar{\xi}$ is obtained from $\xi$ by erasing parentheses and $^+$ operators:

- $\delta_{\text{shift}}(\langle\alpha, \beta\rangle, a) \ni$
$$\begin{cases} \text{if } \alpha \notin V_N : & \begin{cases} \text{if} \begin{pmatrix} \exists A \longrightarrow \gamma \mid \gamma = \eta(\zeta)^+\theta \wedge \\ \alpha a = \bar{\eta}\bar{\bar{\zeta}}\bar{\zeta} \wedge \alpha a\bar{\theta} \in L(\gamma) \cap \tilde{P} \end{pmatrix} \\ \quad \text{then} \quad \langle\bar{\eta}\bar{\bar{\zeta}}, \beta\rangle \text{ else } \langle\alpha a, \beta\rangle \end{cases} \\ \text{if } \alpha \in V_N : & \begin{cases} \text{if} \begin{pmatrix} \exists A \longrightarrow \gamma \mid \gamma = \eta(\zeta)^+\theta \wedge \\ \beta\alpha a = \bar{\eta}\bar{\bar{\zeta}}\bar{\zeta} \wedge \beta\alpha a\bar{\theta} \in L(\gamma) \cap \tilde{P} \end{pmatrix} \\ \quad \text{then} \quad \langle\bar{\eta}\bar{\bar{\zeta}}, \beta\rangle \text{ else } \langle\beta\alpha a, \beta\rangle \end{cases} \end{cases}$$
- $\delta_{\text{push}}(\langle\alpha, \beta\rangle, a) \ni \begin{cases} \langle a, \alpha\rangle & \text{if } \alpha \notin V_N \\ \langle\alpha a, \beta\rangle & \text{if } \alpha \in V_N \end{cases}$
- $\delta_{\text{pop}}(\langle\alpha_1, \beta_1\rangle, \langle\alpha_2, \beta_2\rangle) \ni \langle A, \gamma\rangle$
$$\forall A : \begin{cases} \text{if } \alpha_1 \notin V_N : A \longrightarrow \alpha \in P \wedge \alpha_1 \in L(\alpha) \cap \hat{P} \\ \text{if } \alpha_1 \in V_N : A \longrightarrow \delta \in P \wedge \beta_1\alpha_1 \in L(\delta) \cap \hat{P} \end{cases}$$
$$\text{and } \gamma = \begin{cases} \alpha_2 & \text{if } \alpha_2 \notin V_N \\ \beta_2 & \text{if } \alpha_2 \in V_N. \end{cases}$$

The states reached by push and shift transitions have the first component in $\mathbb{P}$. If state $\langle\alpha, \beta\rangle$ is reached after a push transition, then $\alpha$ is the prefix of the rhs (deprived of the $^+$ operators) that is currently under construction and $\beta$ is the prefix previously under construction; in this case $\alpha$ is either a terminal or a nonterminal followed by a terminal.

If the state is reached after a shift transition, and the $\alpha$ component of the previous state was not a single nonterminal, then the new $\alpha$ is the concatenation of the first component of the previous state with the read character. If, instead, the $\alpha$ component of the previous state was a single nonterminal —which was produced by a pop transition— then the new $\alpha$ also includes the previous $\beta$ and $\beta$ is not changed from the previous state. However, if the new $\alpha$ becomes such that a suffix thereof is a double occurrence of a string $\zeta \in L((\zeta)^+)$ —hence $\alpha \in \mathbb{P}$— then the second occurrence of $\zeta$ is cut from the new $\alpha$, which therefore becomes a prefix of an element of $\hat{P}$.

The states reached by a pop transition have the first component in $V_N$: if $\langle A, \gamma\rangle$ is such a state, then $A$ is the corresponding lhs, and $\gamma$ is the prefix previously under construction.

For instance, imagine that a C-OPG contains the rules $A \longrightarrow (Ba(bc)^+)^+a$ and $B \longrightarrow h$ and that the corresponding OPA $\mathcal{A}$ parses the string $habcbchabca$: after scanning the prefix $habcb$ $\mathcal{A}$ has reduced $h$ to $B$ and has $Babcb$ as the first component of its state; after reading the new $c$ it recognizes that the suffix of the first state component would become a second instance of $bc$ belonging to $(bc)^+$; thus, it goes back to $Babc$. Then, it proceeds with a new reduction of $h$ to $B$ and, when reading with a shift the second $a$ appends $Ba$ to its current $\beta$ which was produced by the previous pop so that the new $\alpha$ becomes $BabcBa$; after shifting $b$ it reads $c$ and realizes that its new $\alpha$ would become $BabcBabc$, i.e., an element of $(Ba(bc)^+)^+$ and therefore "cuts" it to the single instance thereof, i.e., $Babc$. Finally, after having shifted the last $a$ it is ready for the last pop.

The result of $\delta_{\text{shift}}$ and $\delta_{\text{push}}$ is a singleton, whereas $\delta_{\text{pop}}$ may produce several states, in case of repeated rhs. Thus, if $G$ is BD, the corresponding $\mathcal{A}$ is deterministic. Notice that, unlike the case of acyclic OPGs, a BD C-OPG may have rules that are a prefix of other rules, but nevertheless this fact does not preclude the determinism of parsing.

**Example 3.3.** Figure 5 (top right) displays a run of the (nondeterministic) OPA obtained from the (not BD) C-OPG of Figure 5 (top left) accepting the sentence $n + n + n - n - n + n$.

| stack | state | current input |
|---|---|---|
| $\bot$ | $\langle \varepsilon, \varepsilon \rangle$ | $n+n+n+n-n-n+n+n-n+n+n\#$ |
| $\bot[n,\ \langle \varepsilon, \varepsilon \rangle]$ | $\langle n, \varepsilon \rangle$ | $+n+n+n-n-n+n+n-n+n+n\#$ |
| $\bot$ | $\langle T, \varepsilon \rangle$ | $+n+n+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T+, \varepsilon \rangle$ | $n+n+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, \varepsilon \rangle]$ | $\langle n, T+ \rangle$ | $+n+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $+n+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle \mathbf{T+}, \mathbf{T+} \rangle$ | $n+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, T+ \rangle]$ | $\langle n, T+ \rangle$ | $+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $+n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle \mathbf{T+}, \mathbf{T+} \rangle$ | $n-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, T+ \rangle]$ | $\langle n, T+ \rangle$ | $-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle M, T+ \rangle$ | $-n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle]$ | $\langle M-, T+ \rangle$ | $n-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle][n,\ \langle M-, T+ \rangle]$ | $\langle n, M- \rangle$ | $-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle]$ | $\langle N, M- \rangle$ | $-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle M, T+ \rangle$ | $-n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle]$ | $\langle M-, T+ \rangle$ | $n+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle][n,\ \langle M-, T+ \rangle]$ | $\langle n, M- \rangle$ | $+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle]$ | $\langle N, M- \rangle$ | $+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $+n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle \mathbf{T+}, \mathbf{T+} \rangle$ | $n+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, T+ \rangle]$ | $\langle n, T+ \rangle$ | $+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $+n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle \mathbf{T+}, \mathbf{T+} \rangle$ | $n-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, T+ \rangle]$ | $\langle n, T+ \rangle$ | $-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle M, T+ \rangle$ | $-n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle]$ | $\langle M-, T+ \rangle$ | $n+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle][n,\ \langle M-, T+ \rangle]$ | $\langle n, M- \rangle$ | $+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][-,\ \langle M, T+ \rangle]$ | $\langle N, M- \rangle$ | $+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $+n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle \mathbf{T+}, \mathbf{T+} \rangle$ | $n+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, T+ \rangle]$ | $\langle n, T+ \rangle$ | $+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $+n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle \mathbf{T+}, \mathbf{T+} \rangle$ | $n\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle][n,\ \langle T+, T+ \rangle]$ | $\langle n, T+ \rangle$ | $\#$ |
| $\bot[+,\ \langle T, \varepsilon \rangle]$ | $\langle T, T+ \rangle$ | $\#$ |
| $\bot$ | $\langle P, \varepsilon \rangle$ | $\#$ |

$G_{\text{C-AE}}$ :

$$
\begin{aligned}
&S = \{P, T, M, N\}\\
&P \to (T+)^{+}T \mid n\\
&T \to M - N \mid n\\
&M \to M - N \mid n\\
&N \to n
\end{aligned}
$$

|  | $+$ | $-$ | $n$ | $\#$ |
|---|---|---|---|---|
| $+$ | $\doteq$ | $\lessdot$ | $\lessdot$ | $\gtrdot$ |
| $-$ | $\gtrdot$ | $\gtrdot$ | $\lessdot$ | $\gtrdot$ |
| $n$ | $\gtrdot$ | $\gtrdot$ |  | $\gtrdot$ |
| $\#$ | $\lessdot$ | $\lessdot$ | $\lessdot$ | $\doteq$ |

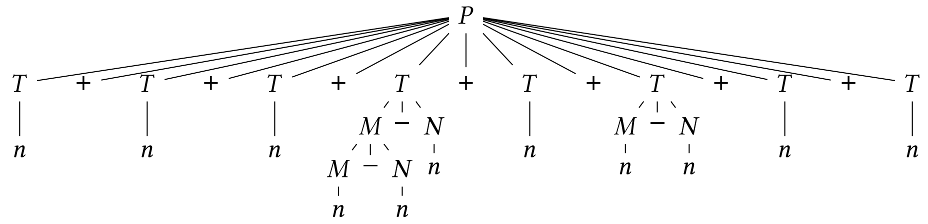


**Figure 5.** A C-OPG (top left), its OPM (middle left), a ST generated by them (bottom), and a run of the OPA built from the C-OPG accepting the sentence $n+n+n+n-n-n+n+n-n+n+n$ (top right). The states truncated by erasing a repeated suffix $\zeta$ occurring under the scope of a $^{+}$ operator are emphasized. Notice that the adopted C-OPG is not BD; thus, the corresponding OPA is nondeterministic and the accepting computation given here is just one among other failing ones.

## 4 Parallel parsing based on cyclic OPGs

The OPA built by Theorem 3.2 is a pure acceptor of all OPLs as defined by C-OPGs. To build a *parallel parser generator* based thereon, we need two major constructions:

1. As well as a generic *pushdown acceptor* for CFLs must be augmented to a *pushdown transducer* to become a real *parser* suitable to build the ST of a CF sentence, the OPA must be augmented with operations to build elements of the ST during its analysis of the input. Notice that in this case the ST to be built must be unranked.
2. As it was done in [2], mechanisms must be built to:
   a. split the input into chunks to be partially parsed by independent *workers*;
   b. recompose their partial output (ST fragments) into a new input to be resubmitted to the parser to produce a complete ST of the original string (possibly by iterating several passes of parallelization in case of extremely long input).

As it happened for [2] too, this requires an important enrichment of the original sequential parser since, after the first pass, parsing must work on partially processed input that is not anymore a simple string on the input alphabet. We will see, however, that this requires a major departure from algorithms given in [2].

In the rest of this section, we describe the global structure of a parallel parser based on C-OPGs with the help of a running example consisting of the parallelization of Example 3.3. Subsequently, we provide an enriched parsing algorithm suitable to work on fragments of input strings, to produce *partial STs* thereof and to recombine them to obtain a full ST of the original input.

### 4.1 Splitting the input string into chunks

Depending on the length of the input string and on the available parallelism, the input string is split into a number of *chunks* to be assigned to the available independent *workers*. Whereas in normal parsing of OPLs the input string is always delimited by the marker #, in this case only the left-(resp. right-)most chunk is delimited at its left (resp. right) by character #; thus, a generic chunk may be delimited at its left and/or right by any character in $\Sigma$. Two consecutive chunks must share the same character that acts as delimiter, resp. right and left, to allow for driving the parsing by means of the precedence relations w.r.t. the preceding and following chunks.

For instance, with reference to the grammar of Example 3.3, the input $\#n + n + n + n - n - n + n + n - n + n + n\#$ could be split into the three chunks: $\#n{+}n{+}n{+}n{-}$, ${-}n{-}n{+}n{+}$, $+n - n + n + n\#$, where the character + plays, respectively, the role of right and left delimiter for the second and third chunks, and − acts as the right delimiter for first chunk).

### 4.2 Partial, parallel parsing of every chunk

Thanks to the *local parsability property* [2] every chunk can be parsed independently on the other ones but, in general, the parsing can proceed only as far as in the chunk there will be substrings enclosed within a pair of matching $\lessdot$ and $\gtrdot$. Thus, the OPA driving the parsing will stop after producing a fragment of ST and will remain with a nonempty stack consisting of two adjacient parts (in some cases one of them can be empty) containing, respectively, the left side, elements such that no $\lessdot$ relation holds between two consecutive ones of them, and the right side elements with no $\gtrdot$ PR in between. To clarify, let us consider how the OPA built from the C-OPG of Figure 5 performs on the three above chunks. For the first chunk it produces the transition sequence:

$\langle \bot,\ \langle \varepsilon, \varepsilon \rangle,\ n + n + n + n - \rangle \vdash^* \langle \bot[+, \langle T, \varepsilon \rangle],\ \langle M, T+ \rangle,\ - \rangle$ (remember that the symbol − to the right acts as a delimiter to compute the PR with the preceding character but cannot be read).

Notice that during this computation the OPA does not push onto the stack all produced pairs $T+$: only the firstly pushed one remains therein —possibly, but not in this case, with a different terminal symbol— until no further progress is possible; they all must be stored in the ST under construction, however. Thus, the parser at every shift action must also generate the corresponding nodes to store them and bind them to their siblings, as shown in Figure 6(a). The last node labeled $M$ is not a sibling of previous ones labeled $T$ since the + that precedes it $\lessdot$ to the − following it.

Consider now the third chunk: it is easy to verify that the computation $\langle +,\ \langle \varepsilon, \varepsilon \rangle,\ n{-}n{+}n{+}n\# \rangle \vdash^* \langle +[T, \langle T, \varepsilon \rangle],\ \langle T, T+ \rangle,\ \# \rangle$ will stop after producing the partial ST given in Figure 6(c). Notice that, whereas the first computation ends with a pending $+ \lessdot -$, the third one remains with a pending $+ \gtrdot \#$.

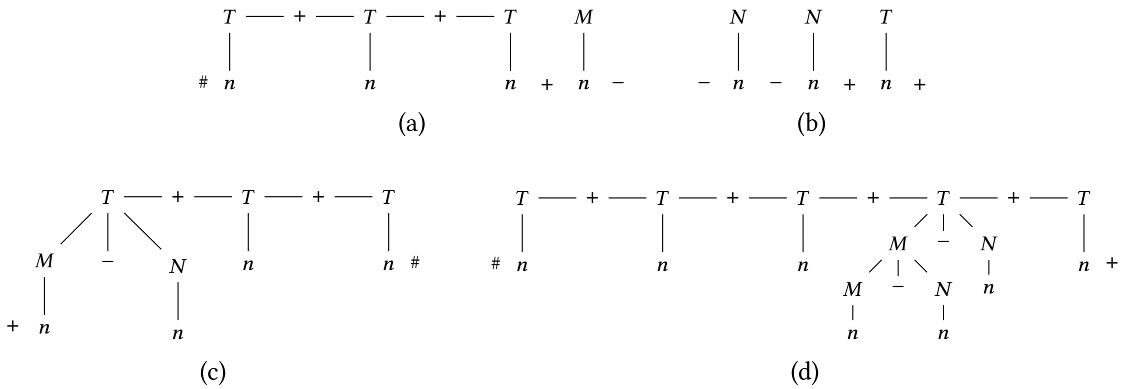
Finally, consider the chunk in between. The OPA's initial configuration is $\langle -,\ \langle \varepsilon, \varepsilon \rangle,\ n - n - n + n+ \rangle$; after pushing $n$ and immediately reducing it to $N$, it enters configuration $\langle -,\ \langle N, \varepsilon \rangle,\ -n - n + n+ \rangle$ and produces the portion of ST consisting of the node $N$ with child $n$; the stack consists only of its bottom storing a − (instead of the usual #) which $\gtrdot$ over the − at the beginning of the remaining input; since the $\gtrdot$ is not matched by any $\lessdot$ below it, the OPA cannot proceed with any further reduction. This is not an error situation, however: thus, we execute a *dummy push* (as in the previous [2]), i.e. we read the next character and push it onto the stack, paired with the current state; then the state is re-initialized to $\langle \varepsilon, \varepsilon \rangle$ to emphasize that the computation now restarts as if it were the beginning of the input with the present stack contents acting as its bottom since it will never be affected. After having pushed and reduced another $n$ we find again a $\gtrdot$ between the − on top of the stack and the + to be read; thus, we proceed with a new dummy push and re-initialization of the state and eventually produce the configuration $\langle -[-, \langle N, \varepsilon \rangle][+, \langle N, \varepsilon \rangle],\ \langle T, \varepsilon \rangle,\ + \rangle$ and the partial ST given in Figure 6(b). This time the PRs between consecutive terminals of the stack contents, including the two delimiters are $- \gtrdot - \gtrdot + \doteq +$. The corresponding partial ST is given in Figure 6(b).

Notice that the order between terminals and nonterminals in any element of the stack is inverted w.r.t. the order of the same symbols in the partial STs: this is because the terminal component is the read character whereas the state component is the state of the OPA *before* the reading.

### 4.3 Combining the partial outputs and building the complete ST

We now have to build a new input and, while parsing it, connect the three ST fragments to obtain the complete one.

First, observe that the parser can be set into an initial configuration that is the final one of the parse of the first chunk. Then, the key observation is that, thanks to the local parsability property, the partial STs built so far by the three parallel workers will not be affected by further processing: thus, *we use them as the input of the new pass* (more precisely, the new input is the sequence of terminals and nonterminals yet to be reduced by further processing: e.g., for the second chunk the new input is $-N - N + T+$ where, as usual the two extreme terminals act as delimiters). This will require augmenting the present OPA to allow it to read characters in $V_N$ too. This is quite simple since nonterminals to be read are the root of subtrees built in the previous pass, i.e.,



(a) (b) (c) (d)

**Figure 6.** (a) The partial ST built by the parser while processing $\#n + n + n + n-$. (b) The partial ST built by the parser while processing $-n - n + n+$. (c) The partial ST built by the parser while processing $+n - n + n + n\#$. (d) The partial ST obtained by recombining (a) and (b).

the $\alpha$ component of the state reached after the pop move that created them, which, in turn, is the lhs of a grammar's production —if the OPA is the one built on the basis of the C-OPG—. On the other hand, the $\beta$ component, i.e. the rhs *previously* under construction, is the same as it was when the parsing of the subtree rooted in the read nonterminal began; therefore, it remains unaffcted in the new state (see Figure 4).

To illustrate, let us consider the transition from the first to the second chunk in our running example.

The first step to join the parsing of the two chunks is to "read" the character − that acted as right and left delimiter, resp., of the two chunks: since the terminal $symbol(\Pi) = + \lessdot -$, the transition function to apply is a $\delta_{\text{push}}$ (it could also be a $\delta_{\text{shift}}$ in case of $\doteq$ PR but not a $\delta_{\text{pop}}$). Thus, the next configuration of the OPA is

$\langle \bot[+, \langle T, \varepsilon\rangle][-, \langle M, T+\rangle],\ \langle M-, T+\rangle,\ N - N + T+\rangle$

At this point the next move consists in *reading the nonterminal* $N$: this leads the OPA into the configuration

$\langle \bot[+, \langle T, \varepsilon\rangle][-, \langle M, T+\rangle],\ \langle M - N, T+\rangle,\ -N+T+\rangle$ (a configuration that is instead "implicit" in the transition sequence of Figure 5) and immediately performs the pop move that leads it to $\langle \bot[+, \langle T, \varepsilon\rangle],\ \langle M, T+\rangle,\ -N + T+\rangle$. In essence, the reading of $N$ has led the OPA from the 17th row of Figure 5 to the 20th one, a little gain in this example, but remember that the gain is proportional to the size of the subtree rooted in the read nonterminal.

The computation now proceeds in the same way, finally reaching the configuration $\langle \bot[+, \langle T, \varepsilon\rangle],\ \langle T, T+\rangle,\ +\rangle$ and producing the partial tree of Figure 6(d), ready to be completed by processing the partial tree of Figure 6(c) in the same way as for the transition from the first to the second chunk, and therefore producing the ST of Figure 5 (where the tree representation has been changed from the one typical of ranked trees to the one for unranked trees).

Notice that after attaching the fragment of Figure 6(c) to that of Figure 6(d) through the shared character +, the parser should repeat the sequence of shift moves to scan the sequence $T + T + T\#$, skipping the subtrees rooted in the nonterminals. This sequence of moves is obviously an useless replay, but can be naturally and efficiently avoided by remembering that a shift move of the OPA does not affect the stack, apart from the change of the $\Sigma$-component of its top. Thus, when the parser enters the first $T$ of the sequence, the stack left by the worker of the previous pass already refers to the last + of the sequence (which $\gtrdot\#$) and closes the sequence of shift moves. The parser can therefore be repositioned at that point of the partial ST with a potential dramatic gain when the input consists of very long sequences of elements in the $\doteq$ PR.

### 4.4 Parsing algorithm revised for parallelization

We are now ready for a complete parsing algorithm that is suitable to:

- Partially parse chunks of the global input.
- Receive and deliver fragments of STs.
- Accept initial —and deliver final— configurations other than the standard OPA's ones (with empty stack and initial and accepting states).
- Receive as input strings $\in V^*$. Such strings consist of sequences of terminals and nonterminals of partial STs that have not yet been reduced (due to the lack of pairs $(\lessdot, \gtrdot)$ enclosing portions thereof) as exemplified in Figure 6.

An abstract description of this algorithm is given in Algorithm 1. For simplicity and consistency with previous examples, we assume that a state is formalized as the pair $(\alpha, \beta)$ described in Section 3, and that the fragments of the ST are kept as a global variable, built incrementally and in parallel by the various workers, and implemented in the usual way of unranked trees. The reader can verify that such an abstract algorithm formulation can be easily adapted to different conventions of state and tree representations. The fragments of ST's are stored as an ordered sequence of subtrees —whose roots are possibly linked through one or more terminal characters that are in the PR $\doteq$ between each other as, e.g., in Figure 6(a)— built so far; initialized to the input string, i.e., the sequence of ST's leaves.

During the first parsing pass, $\xi \in \Sigma^*$ is a chunk, i.e., a sequence of contiguous terminals of the global input string stored in nodes that are leaves of the ST to be built. Later, it will contain also nonterminals, obtained from the concatenation of several contiguous partial STs sharing the characters playing the role of respective right and left delimiters.

Thus, after the first pass every character Y of $\xi$ refers to, and is the label of, a node $\tau_Y$ of the ST: if Y is a terminal $\tau_Y$ is a leaf; if Y is a nonterminal, $\tau_Y$ is the root of one of the subtrees belonging to the ST fragment (remember that two consecutive nonterminals are separated by at least one terminal character).

For simplicity in Algorithm 1 we do not mention explicitly the creation of leaf nodes in the particular case of the first pass. We also assume that the state that was entered by the OPA when the node was lastly visited is always immediately available —e.g. by storing it as an attribute of the node.

## 5 Implementation Notes

To support an experimental proof of concept of the effectiveness of parallel parsing based on C-OPGs, we developed GoPAPAGENO [14, 15]. GoPAPAGENO is the C-OPG evolution of a homonymous tool [17], which in turn is the re-implementation of the original PAPAGENO [2] in GO, a programming language supporting efficient concurrency and parallel programming.

GoPAPAGENO builds an OPA from a given C-OPG along the lines illustrated in Theorem 3.2, and performs parallel parsing as follows: first, it splits the input string into chunks of a configurable size, then it runs Alg. 1 in parallel on each chunk, and finally it combines the resulting partial STs, possibly further exploiting parallelism, in a hierarchical way similar to previous work [2, 20].

GoPAPAGENO and its predecessor [17] follow a workflow analogous to that of Flex and Bison [19]. In particular, the present [15] fully imports from [17] a parallel lexer based on a standard regular expression matching algorithm.

Furthermore GoPAPAGENO naturally supports the classic semantic elaboration of tree-structured data based on (synthesized) attributes associated to the nodes of the ST. It also exploits the fact that STs are unranked in that some *semantic action* can be performed even during the construction of a rhs —i.e., in association with shift parsing actions— without waiting for its reduction to the corresponding lhs: this feature may be useful, e.g., when the attribute value of the father node depends on an *associative operation* applied to the children's values, which can be performed online and even in parallel by the various workers.

To simplify the construction of the OPA from the given C-OPG we introduced a restriction on the regular expressions to define grammar rhs: we limited their *star-height* to 1. From a theoretical point of view this affects the generative power of the grammar since it is well-known that regular expressions are a hierarchy w.r.t. their star height but we deem that such a limitation has normally no practical effect. In fact, a hierarchical structure as implied by nested $^+$ operators would better be represented by making it apparent on the structure of the ST —which, however, would in turn be made explicit by a non-$\doteq$-cyclic PR—. Such a restriction could be easily removed at the cost of producing OPAs with a size exponentially depending on the star-height.

## 6 Experimental Evaluation

We compare our C-OPG parsing algorithm (C-OPP) with the original OP parallel parsing algorithm (OPP) [2] and Associative OP Parsing (AOPP) [20]. To exclude performance differences due to the use of different programming languages, and because the tool by [20] is not publicly available, we implemented both baselines within GoPAPAGENO. We made an artifact available to replicate the experiments [5].

We evaluate the approaches on the same three JSON datasets used in [20]:

- *Emojis* [16] (180 kB, replicated to 180 MB) contains a flat list of emoji names associated to URLs pointing to image files.
- *Citylots* [25] (180 MB) contains geographic data of San Francisco's Subdivision parcels, represented as a list of records containing several fields;
- *Wikidata* [24] (180 MB) is a dump of the Wikidata database, consisting of a list of very complex objects.

We chose the JSON format because it is widely used to represent large monolithic datasets, that justify the need for efficient parallel parsing. On the other hand, it features a hierarchical structure that requires a context-free parser, unlike other popular but simpler formats such as CSV. Of the three datasets, Emojis features the flattest structure, and Wikidata the most deeply nested.

We conducted the experiments on a server equipped with an AMD EPYC 7551 (32 cores, 2 GHz) CPU and 503 GiB of RAM, running Debian 6 and Go version 1.23.0. The results are reported in Figure 7, which shows the time required for parsing by the three techniques while varying the degree of parallelism.

When parsing sequentially (1 thread), C-OPP presents a slowdown up to three-fold with respect to OPP and AOPP. This is caused by the additional overhead incurred by C-OPP during initialization, which requires the allocation of more data structures due to the greater complexity of the algorithm and to the fact that a few optimization techniques,

**Algorithm 1** Sequential C-OPP Parsing

```
 1: Input: $\xi \in V^*$; a stack $\Pi$; $q = (\alpha, \beta) \in Q$.    Output: $\Pi'$, updated stack; $q' = (\alpha', \beta') \in Q$.
 2: head := 0
 3: X := $\xi$[head]; Y := symbol(top($\Pi$))
 4: if $X \in V_N$ then                // if $X \in V_N$ it cannot be $\alpha \in V_N$; either $\alpha \in V^*\Sigma_\#$ or $\alpha = \varepsilon$
 5:     $\alpha := \alpha X$; head++     // (and $\Pi = \perp$ only if the chunk being processed is the first one).
 6: else if $Y = \perp$ or $Y \lessdot X$ then                                   // Push move
 7:     Push $(X, q^{\lessdot})$ where the symbol $^{\lessdot}$ is a special marker attached to the state to remember that the state has been pushed onto the stack
        as the consequence of a "real push" as opposed to the "dummy push" used in the case of unmatched $\gtrdot$;
 8:     if $\alpha = Z \in V_N$ then
 9:             // Thus, $Z$ is the first element of the rhs under construction and the node $\tau_Z$ storing it already exists in the partial ST
10:         Append $\tau_X$ to $\tau_Z$ as its right sibling;
        else
                // $\tau_X$ is going to become the leftmost child of a future internal node
11:     $q' := \delta_{\text{push}}(q, X)$; head++
12: else if $Y \doteq X$ then                                                      // Shift move
13:     if $\alpha = \eta Y$ then                          // $\alpha$ is necessarily not $\varepsilon$ because $Y \doteq X$
14:         Append $\tau_X$ to $\tau_Y$ as its right sibling;
15:     else                                      // $\alpha = \eta Y Z, Z \in V_N$ and referring to node $\tau_Z$
16:         Append $\tau_Z$ to node $\tau_Y$, and $\tau_X$ to $\tau_Z$ as their respective right siblings (if not already so);
17:     symbol(top($\Pi$)) := X;
18:     Update the reference of the $X$ item on top of the stack to refer to $\tau_X$;
19:     $q' := \delta_{\text{shift}}(q, X)$;
20:     Update $\xi$[head] to the element referred to by the (updated) element on top of the stack;
21:     Reset the state to the state that was entered last time that the node of the ST was visited
            // In this way, the whole sequence of shift actions already performed in possible previous pass(es) is skipped.
22: else if $Y \gtrdot X$ then
23:     if top($\Pi$) = [Y, $q_t^{\lessdot}$] then        // The item was pushed as a "real push"; thus, this is a pop move.
24:         Create a new (internal) node, say $M$, and make it the father of the leftmost node of the rhs just completed;
25:         $q' := \delta_{\text{pop}}(q, q_t)$
26:         Label $M$ as the $\alpha$ component of $q'$ and let it refer to the newly created node $M$;
27:         Pop
28:     else                                          // The $Y \gtrdot X$ is not matched by any $\lessdot$ in the stack
29:         Push $(X, q)$;
30:         The leaf $\tau_X$ remains with with no left sibling;
31:         let the item just pushed onto the stack refer to $\tau_X$;
32:         $q' := \langle \varepsilon, \varepsilon \rangle$; head++
33: Repeat from Line 3 until end of the input is reached.
```

mainly in the construction of the OPA from the C-OPG have not yet been implemented.

When the number of threads is greater than 1, however, C-OPP consistently outperforms both OPP and AOPP in all benchmarks, due to its better scaling capabilities.

Interestingly, as the thread number increases above 1, performance gets worse before improving again. This effect is more marked for OPP and AOPP than C-OPP, and likely occurs because the additional time spent allocating memory is not yet fully compensated by the parallelism.

For all approaches, the benefit of increasing the number of threads is steeper between 2 and 16, and becomes less marked afterwards. A possible cause is the lack of guarantees on actual parallel execution of *goroutines* by the Go runtime: after a certain threshold, the benefits of adding more goroutines may diminish due to scheduling overhead.

In general, however, the effect of increasing parallelism is positive, except for OPP on the Emojis dataset. In this case, the input string structure is essentially flat, but the OPG employed by OPP is linearly recursive, and generates an extremely deep syntax tree, that can only be parsed strictly sequentially, left-to-right. Indeed, according to Table 1, OPP produces the highest syntax trees of all approaches, while those produced by C-OPP are smaller by several orders of magnitude, thus, fully confirming the effectiveness of C-OPGs.

## 7 Concluding remarks

We have designed and implemented a first prototype of the parallel parser generator GoPAPAGENO which exploits the peculiar features of cyclic OPGs. Cyclic OPGs, in turn, augment the generative power of traditional OPGs just with

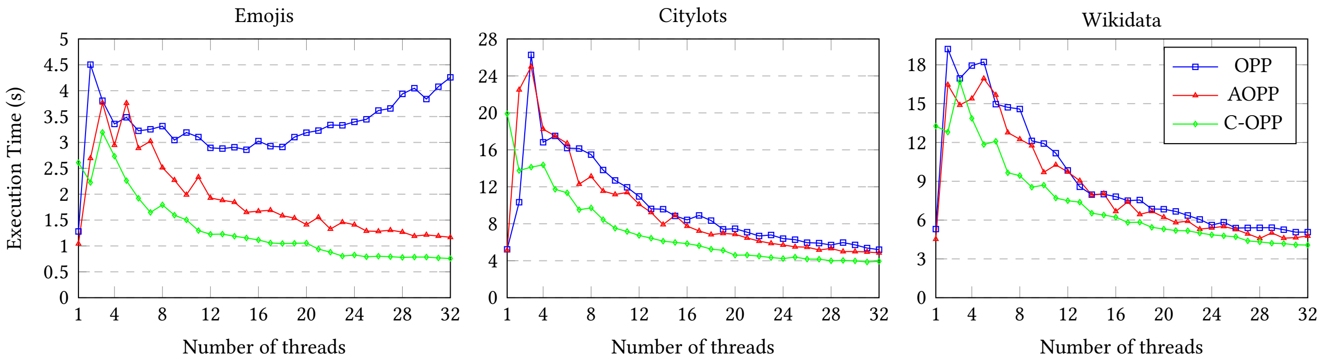


**Figure 7.** Scalability of parallel parsing times for OPP, AOPP, and C-OPP w.r.t. no. of parallel threads (*goroutines*).

the purpose of further enriching the benefits of parallelism. In fact, the experimentation we performed on a fairly typical benchmark already confirmed significant improvements w.r.t. previous instances of parallel parser, which in turn dramatically overtook traditional sequential parsers. We expect, however, much further benefit from the ongoing work, which is planned along the following lines:

- We will generalize the present benchmark to make it better represent all applications devoted to the analysis of tree-shaped data: the present benchmark, in fact, is deliberately focused on JSON-structured data but other fields may exhibit much more complex structures.
  We will also enlarge the comparison with other widely known parsers, whether sequential or parallel.
- We are confident that the performances of GoPAPAGENO can be further significantly improved by introducing a few optimization techniques that are not yet implemented: e.g., the present version is strictly bound to the translation schema from C-OPG to OPA defined in Section 3 —which we believe to be the main reason of the poor performances of the present tool in strictly sequential parsing— but the OPA could be built and optimized once and forever thus avoiding much work during the parsing.
  It is also worth investigating the potential benefits, and relative overheads, obtained by adopting multi-pass policies as opposed to single pass ones (which seem preferable with the present benchmark but not necessarily in other cases [2]). Similarly, some help could be obtained by heuristics guiding the choice for splitting the input into chunks.
- We also look for specific applications, to join parsing with a semantic analysis based on traditional attribute computation: this could go from classical compilation to more recent automatic verification techniques.

**Table 1.** Height of generated syntax trees. The AOPP column contains the minimum value (obtained with 32 threads), while values for OPP and C-OPP do not depend on the number of threads.

| Input | OPP | AOPP | C-OPP |
|---|---:|---:|---:|
| Emojis | 1,935,003 | 61,280 | **5** |
| Citylots | 206,604 | 10,443 | **21** |
| Wikidata | 20,045 | 1,769 | **37** |

## Acknowledgments

The authors are thankful to the anonymous reviewers for their helpful comments and suggestions. This work was partially funded by the European Commission within the Horizon Europe programme under Grant No. 101107303 "CORPORA".

## References

[1] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. Context-Free Languages and Pushdown Automata. In *Handbook of Formal Languages (1)*. 111–174. doi:10.1007/978-3-642-59136-5_3

[2] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Parallel Parsing made practical. *Sci. Comput. Program.* 112, 3 (2015), 195–226. doi:10.1016/j.scico.2015.09.002.

[3] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, and Matteo Pradella. 2013. Parallel parsing of operator precedence grammars. *Inf. Process. Lett.* 113, 7 (2013), 245–249. doi:10.1016/j.ipl.2013.01.008

[4] Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, and Dino Mandrioli. 2015. Syntactic-semantic incrementality for agile verification. *Sci. Comput. Program.* 97 (2015), 47–54.

[5] Michele Chiari and Michele Giornetta. 2025. Boosting Parallel Parsing through Cyclic Operator Precedence Grammars: Artifact. doi:10.5281/zenodo.15262020 Zenodo.

[6] Michele Chiari, Dino Mandrioli, Francesco Pontiggia, and Matteo Pradella. 2023. A Model Checker for Operator Precedence Languages. *ACM Trans. Program. Lang. Syst.* 45, 3 (2023), 19:1–19:66. doi:10.1145/3608443

[7] Michele Chiari, Dino Mandrioli, and Matteo Pradella. 2023. Cyclic Operator Precedence Grammars for Parallel Parsing. *CoRR* abs/2309.04200 (2023). arXiv:2309.04200 [cs.FL] https://arxiv.org/abs/2309.04200

[8] Michele Chiari, Dino Mandrioli, and Matteo Pradella. 2024. Cyclic Operator Precedence Grammars for Improved Parallel Parsing. In

*DLT'24 (LNCS, Vol. 14791)*. Springer, 98–113. doi:10.1007/978-3-031-66159-4_8

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

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

[11] Stefano Crespi Reghizzi and Matteo Pradella. 2020. Beyond operator-precedence grammars and languages. *J. Comput. System Sci.* 113 (2020), 18–41. doi:10.1016/j.jcss.2020.04.006

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

[13] Carlo Ghezzi and Dino Mandrioli. 1979. Incremental Parsing. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 58–70. doi:10.1145/357062.357066

[14] Michele Giornetta. 2024. GoPAPAGENO. https://github.com/giornetta/gopapageno.

[15] Michele Giornetta. 2024. *Parallel parsing of cyclic operator precedence grammars*. Master's thesis. Politecnico di Milano. https://www.politesi.polimi.it/handle/10589/227581

[16] GitHub, Inc. 2022. REST API endpoints for emojis. https://api.github.com/emojis.

[17] Simone Guidi. 2017. GoPAPAGENO. https://github.com/simoneguidi94/gopapageno.

[18] Michael A. Harrison. 1978. *Introduction to Formal Language Theory*. Addison Wesley.

[19] John Levine. 2009. *flex & bison*. O'Reilly Media.

[20] Le Li and Kenjiro Taura. 2023. Associative Operator Precedence Parsing: A Method To Increase Data Parsing Parallelism. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2023, Singapore, 27 February 2023 - 2 March 2023*. ACM, 75–87. doi:10.1145/3578178.3578233

[21] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. doi:10.14778/3115404.3115416

[22] Violetta Lonati, Dino Mandrioli, Federica Panella, and Matteo Pradella. 2015. Operator Precedence Languages: Their Automata-Theoretic and Logic Characterization. *SIAM J. Comput.* 44, 4 (2015), 1026–1088.

[23] Arto K. Salomaa. 1973. *Formal Languages*. Academic Press, New York, NY.

[24] Wikimedia. 2024. Wikidata JSON dump. https://dumps.wikimedia.org/wikidatawiki/entities/.

[25] Mirco Zeiss. 2012. City Lots in San Francisco in .json. https://github.com/zemirco/sf-city-lots-json/.