Generalizing input-driven languages: theoretical and practical benefits

Dino Mandrioli[a], Matteo Pradella[a,b]

[a]DEIB, Politecnico di Milano, P.zza L. da Vinci 32, 20133 Milano, Italy.
Email: {dino.mandrioli, matteo.pradella}@polimi.it
[b]IEIIT, Consiglio Nazionale delle Ricerche, via Ponzio 34/5, 20133 Milano, Italy

Abstract

Regular languages (RL) are the simplest family in Chomsky's hierarchy. Thanks to their simplicity they enjoy various nice algebraic and logic properties that have been successfully exploited in many application fields. Practically all of their related problems are decidable, so that they support automatic verification algorithms. Also, they can be recognized in real-time.

Context-free languages (CFL) are another major family well-suited to formalize programming, natural, and many other classes of languages; their increased generative power w.r.t. RL, however, causes the loss of several closure properties and of the decidability of important problems; furthermore they need complex parsing algorithms. Thus, various subclasses thereof have been defined with different goals, spanning from efficient, deterministic parsing to closure properties, logic characterization and automatic verification techniques.

Among CFL subclasses, so-called *structured* ones, i.e., those where the typical tree-structure is visible in the sentences, exhibit many of the algebraic and logic properties of RL, whereas deterministic CFL have been thoroughly exploited in compiler construction and other application fields.

After surveying and comparing the main properties of those various language families, we go back to *operator precedence languages* (OPL), an old family through which R. Floyd pioneered deterministic parsing, and we show that they offer unexpected properties in two fields so far investigated in totally independent ways: they enable parsing parallelization in a more effective way than traditional sequential parsers, and exhibit the same algebraic and logic properties so far obtained only for less expressive language families.

Keywords: regular languages, context-free languages, input-driven languages, visibly pushdown languages, operator-precedence languages, monadic second order logic, closure properties, decidability and automatic verification

Contents

1. Introduction

Regular (RL) and context-free languages (CFL) are by far the most widely studied families of formal languages in the rich literature of the field. In Chomsky's hierarchy, they are, respectively, in positions 2 and 3, 0 and 1 being recursively enumerable and context-sensitive languages.

Thanks to their simplicity, RL enjoy practically all positive properties that have been defined and studied for formal language families: they are closed under most algebraic operations, and most of their properties of interest (emptiness, finiteness, containment) are decidable. Thus, they found fundamental applications in many fields of computer and system science: HW circuit design and minimization, specification and design languages (equipped with powerful supporting tools), automatic verification of SW properties, etc. One of their most relevant applications is now model-checking which exploits the decidability of the containment problem and important characterizations in terms of mathematical logics [1, 2].

On the other hand, the typical linear structure of RL sentences makes them unsuitable or only partially suitable for application in fields where the data structure is more complex, e.g., is tree-like. For instance, in the field of compilation they are well-suited to drive lexical analysis but not to manage the typical nesting of programming and natural language features. The classical language family adopted for this type of modeling and analysis is the context-free one. The increased expressive power of CFL allows to formalize many syntactic aspects of programming, natural, and various other categories of languages. Suitable algorithms have been developed on their basis to parse their sentences, i.e., to build the structure of sentences as syntax-trees.

General CFL, however, lose various of the nice mathematical properties of RL: they are closed only under some of the algebraic operations, and several decision problems, typically the inclusion problem, are undecidable; thus, the automatic analysis and synthesis techniques enabled for RL are hardly generalized to CFL. Furthermore, parsing CFL may become considerably less efficient than recognizing RL: the present most efficient parsing algorithms of practical use for general CFL have an $O(n^3)$ time complexity.

The fundamental subclass of deterministic CFL (DCFL) has been introduced, and applied to the formalization of programming language syntax, to exploit the fact that in this case parsing is in $O(n)$. DCFL, however, do not enjoy enough algebraic and logic properties to extend to this class the successful applications developed for RL: e.g., although their equivalence is decidable, containment is not; they are closed under complement but not under union, intersection, concatenation and Kleene *.

From this point of view, *structured CFL* are somewhat in between RL and general CFL. Intuitively, by structured CFL we mean languages where the structure of the syntax-tree associated with a given sentence is immediately apparent in the sentence. *Parenthesis languages (PL)* introduced in a pioneering paper by McNaughton [3] are the first historical example of such languages. McNaughton showed that they enjoy closure under Boolean operations (which, together with the decidability of the emptiness problem, implies decidability of the containment problem) and their generating grammars can be minimized in a similar way as *finite state automata* (FSA) are minimized (in fact an equivalent formalism for parenthesis languages are *tree automata* [4, 5]).

Starting from PL various extensions of this family have been proposed in the literature, with the main goal of preserving most of the nice properties of RL and PL, yet increasing their generative power; among them *input-driven languages (IDL)* [6, 7], later renamed *visibly pushdown languages (VPL)* [8] have been quite successful: the attribute *Input-driven* is explained by the property that their recognizing pushdown automata can decide whether to apply a push operation, or a pop one to their pushdown store or leaving it unaffected exclusively on the basis of the current input symbol; the attribute *visible*, instead, refers to the fact that their tree-like structure is immediately visible in their sentences.

IDL, alias VPL, are closed under all traditional language operations (and therefore enjoy the consequent decidability properties). Also, they are characterized in terms of a *monadic second order* (MSO) logic by means of a natural extension of the classic characterization for RL originally and independently developed by Büchi, Elgot, and Trakhtenbrot [9, 10, 11]. For these reasons they are a natural candidate for extending model checking techniques from RL. To achieve such a goal in practice, however, MSO logic is not yet tractable due to the complexity of its decidability problems; thus, some research is going on to "pair" IDL with specification languages inspired by temporal logic as it has been done for RL [12].

Structured languages do not need a real *parsing*, since the syntax-tree associated with their sentences is already ”embedded” therein; thus, their recognizing automata only have to decide whether an input string is accepted or not, whereas full parsers for general CFL must build the structure(s) associated with any input string which naturally supports its semantics (think, e.g., to the parsing of unparenthesized arithmetic expressions where the traditional precedence of multiplicative operators over the additive ones is ”hidden” in the syntax of the language.) This property, however, severely restricts their application field as the above example of arithmetic expressions immediately shows.

Rather recently, we resumed the study of an old class of languages which was interrupted a long time ago, namely *operator precedence languages (OPL)*. OPL and their generating grammars (OPG) have been introduced by Floyd [13] to build efficient deterministic parsers; indeed they generate a large and meaningful subclass of DCFL. We can intuitively describe OPL as "*input driven but not visible*": they can be claimed as *input-driven* since the parsing actions on their words –whether to push or pop– depend exclusively on the input alphabet and on the relation defined thereon, but their structure is *not visible* in their words: e.g, they can include unparenthesized expressions.

In the past their algebraic properties, typically closure under Boolean operations [14], have been investigated with the main goal of designing inference algorithms for their languages [15]. After that, their theoretical investigation has been abandoned because of the advent of more powerful grammars, mainly LR ones [16, 17], that generate all DCFL (although some deterministic parsers based on OPL's simple syntax have been continuously implemented at least for suitable subsets of programming languages [18]).

The renewed interest in OPG and OPL has been ignited by two seemingly unrelated remarks: on the one hand we realized that they are a proper superclass of IDL and that all results that have been obtained for them (closures, decidability, logical characterization) extend naturally, but not trivially, to OPL; on the other hand new motivation for their investigation comes from their distinguishing property of *local parsability*: with this term we mean that their deterministic parsing can be started and led to completion from any position of the input string unlike what happens with general deterministic pushdown automata, which must necessarily operate strictly left-to-right from the beginning of the input. This property has a strong practical impact since it allows for exploiting modern parallel architectures to obtain a natural speed up in the processing of large tree-structured data. An automatic tool that generates parallel parsers for these grammars has already been produced and is freely available. The same local parsability property can also be exploited to incrementally analyze large structures without being compelled to reparse them from scratch after any modification thereof.

This renewed interest in OPL has also led to extend their study to *ω-languages*, i.e., those consisting of infinite strings: in this case too the investigation produced results that perfectly parallel the extension of other families, noticeably RL and IDL, from the finite string versions to the infinite ones.

In this paper we follow the above "story" since its beginning to these days and, for the first time, we join within the study of one single language family two different application domains, namely parallel parsing on the one side, and algebraic and logic characterization finalized to automatic verification on the other side. To the best of our knowledge, OPL is the largest family that enjoys all of such properties.

The paper is structured as follows: initially we resume the two main families of formal languages. Section 2 briefly summarizes the main features of RL and focuses more deeply on their logic characterization which is probably less known to the wider computer science audience and is crucial for some fundamental results of this paper. Similarly, Section 3 introduces CFL by putting the accent on the problems of their parsing, whether deterministic or not, and their logic characterization. Then, two sections are devoted to different subclasses of CFL that allowed to obtain important properties otherwise lacking in the larger original family: precisely, Section 4 deals with *structured CFL*, i.e., those languages whose tree-shaped structure is in some way immediately apparent from their sentences with no need to parse them; it shows that for such subclasses of CFL important properties of RL, otherwise lacking in the larger original family, still hold. Finally, Section 5 "puts everything together" by showing that OPL on the one hand are significantly more expressive than traditional structured languages, but enjoy the same important properties as regular and structured CFL and, on the other hand, enable exploiting parallelism in parsing much more naturally and efficiently than for general deterministic CFL.

Since all results presented in this paper have already appeared in the literature, we based our presentation more on intuitive explanation and simple examples than on detailed technical constructions and proofs, to which appropriate references are supplied for the interested reader.

We assume some familiarity with the basics of formal language theory, i.e., regular and CFL, their generating grammars and recognizing automata and their main algebraic properties, which are available in many textbooks such as [17, 19]. However, since the adopted mathematical notation is not always standard in the literature, we report in Table 1 the notation adopted in the paper. An earlier version of this paper, with more details and introductory material about basic formal language theory is available in [20].

The following *naming conventions* are adopted for letters and strings, unless otherwise specified: lowercase Latin letters at the beginning of the alphabet $a, b, \ldots$ denote terminal characters; uppercase Latin letters $A, B, \ldots$ denote nonterminal characters; lowercase Latin letters at the end of the alphabet $x, y, z \ldots$ denote terminal strings; lowercase Greek letters $\alpha, \ldots, \omega$ denote strings over V.

For convenience we do not add a final 's' to acronyms when used as plurals so that, e.g., CFL denotes indifferently a single language, the language family and all languages in the family.

2. Regular Languages

The family of regular languages (RL) is one of the most important families in computer science. In the traditional Chomsky's hierarchy it is the least powerful language family. Its importance stems from both its simplicity and its rich set of properties.

RL is a very robust family: it enjoys many closure properties and practically all interesting decision problems are decidable for RL. It is defined through several different devices, both operational and descriptive. Among them we mention *Finite State Automata* (FSA), which are used for various applications, not only in computer science, *Regular Grammars*, *Regular Expressions*, often used in computing for describing the lexical elements of programming languages and in many programming environments for managing program sources, and various logic classifications that support automatic verification of their properties.

Next, we summarize the main and well-known algebraic and decidability properties of RL; subsequently, in Subsection 2.1, we introduce the characterization of RL in terms of an MSO logic, which will be later extended to larger and larger language families.

- RL are a Boolean algebra with top element Σ^* and bottom element the empty language $\emptyset$,
- RL are also closed w.r.t. *concatenation*, *Kleene* *, *string homomorphism*, and *inverse string homomorphism*.

Table 1: Adopted notation for typical formal language theory symbols.

Entity	Notation
Empty string	ε
Access to characters	$x = x(1)x(2)\dots x(\lvert x\rvert)$
Grammar	(V_n, Σ, P, S), where V_N is the nonterminal alphabet, Σ the terminal alphabet, P the productions, and S the axiom. V denotes $V_n \cup \Sigma$
Finite State Automaton (FSA)	$(Q, \delta, \Sigma, I, F)$, where Q is the set of states, δ the transition relation (or function in case of deterministic automata), I and F the sets of initial and final states, resp.
Pushdown Automaton (PDA)	$(Q, \delta, \Sigma, I, F, \Gamma, Z_0)$, where Γ is the stack alphabet, and Z_0 is the initial stack symbol; the other symbols are the same as for FSA
Stack	When writing stack contents, we assume that the stack grows leftwards, e.g. in $A\alpha$, the symbol A is at the top.
Grammar production or grammar rule	$\alpha \to \beta \mid \gamma$; α is the left hand side (lhs) of the production, β and γ two alternative right hand sides (rhs), the ’|’ symbol separates alternative rhs to shorten the notation. In CFG α is a single nonterminal character
Grammar derivation and logic implication	$\Rightarrow$, overloaded symbol
Automaton transition relation between configurations c_1 and c_2	$c_1 \vdash\!\!\!- c_2$

- Nondeterminism does not affect the recognition power of FSA, i.e., given a nondeterministic FSA an equivalent deterministic one can effectively be built.

- FSA are *minimizable*, i.e. given a FSA, there is an algorithm to build an equivalent automaton with the minimum possible number of states.

- Thanks to the *Pumping Lemma* many properties of RL are decidable; noticeably, *emptiness, infiniteness*, and, thanks to the Boolean closures, *containment*.

- The same Pumping Lemma can be used to prove the limits of the power of FSA, e.g., that the CFL $\{a^n b^n\}$ is not regular.

2.1. Logic characterization

From the very beginning of formal language and automata theory the investigation of the relations between defining a language through some kind of abstract machine and through a logic formalism has produced challenging theoretical problems and important applications in system design and verification. A well-known example of such an application is the classical Hoare's method to prove the correctness of a Pascal-like program w.r.t. a specification stated as a pair of pre- and post-conditions expressed through a first-order theory [21].

Such a verification problem is undecidable if the involved formalisms have the computational power of Turing machines but may become decidable for less powerful formalisms as in the important case of RL. Originally, Büchi, Elgot, and Trakhtenbrot [9, 10, 11] independently developed an MSO logic defining exactly the RL family, so that the decidability properties of this class of languages could be exploited to achieve automatic verification; later on, in fact a major breakthrough in this field has been obtained thanks to advent of model checking.

Let us define a countable infinite set of first-order variables $\boldsymbol{x}, \boldsymbol{y}, \dots$ and a countable infinite set of monadic second-order (set) variables $\boldsymbol{X}, \boldsymbol{Y}, \dots$. In the following we adopt the convention to denote first and second-order variables in boldface italic font.

The basic elements of the logic defined by Büchi and the others are summarized here:

- First-order variables, denoted as lowercase letters at the end of the alphabet, $\boldsymbol{x}$, $\boldsymbol{y}$, ... are interpreted over the natural numbers $\mathbb{N}$ (these variables are written in boldface to avoid confusion with strings);

- Second-order variables, denoted as uppercase letters at the end of the alphabet, written in boldface, $\boldsymbol{X}$, $\boldsymbol{Y}$, ... are interpreted over sets of natural numbers;

- For a given input alphabet Σ, the monadic predicate $a(\cdot)$ is defined for each $a \in \Sigma$: $a(\boldsymbol{x})$ evaluates to true in a string iff the character at position $\boldsymbol{x}$ is a;

- The *successor* predicate is denoted by succ, i.e. $\mathrm{succ}(\boldsymbol{x}, \boldsymbol{y})$ means that $\boldsymbol{y} = \boldsymbol{x} + 1$.

- Let $\mathcal{V}_1$ be a set of first-order variables, and $\mathcal{V}_2$ be a set of second-order (or set) variables. The MSO (*monadic second-order logic*) is defined by the following syntax:

  $$\varphi := a(\boldsymbol{x}) \mid \boldsymbol{x} \in \boldsymbol{X} \mid \mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \mid \neg\varphi \mid \varphi\vee\varphi \mid \exists\boldsymbol{x}(\varphi) \mid \exists\boldsymbol{X}(\varphi).$$

  where $a \in \Sigma$, $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}_1$, and $\boldsymbol{X} \in \mathcal{V}_2$.

- The usual predefined abbreviations are introduced to denote the remaining propositional connectives, universal quantifiers, arithmetic relations ($=, \neq, <, >$), sums and subtractions between first order variables and numeric constants. E.g. $\boldsymbol{x} = \boldsymbol{y}$ and $\boldsymbol{x} < \boldsymbol{y}$ are abbreviations for $\forall\boldsymbol{X}(\boldsymbol{x} \in \boldsymbol{X} \iff \boldsymbol{y} \in \boldsymbol{X})$ and $\forall\boldsymbol{X}\left(\begin{array}{c} \exists\boldsymbol{w}(\mathrm{succ}(\boldsymbol{x}, \boldsymbol{w}) \wedge \boldsymbol{w} \in \boldsymbol{X})\wedge \\ \forall\boldsymbol{z}(\boldsymbol{z} \in \boldsymbol{X} \Rightarrow \exists\boldsymbol{v}(\mathrm{succ}(\boldsymbol{z}, \boldsymbol{v}) \wedge \boldsymbol{v} \in \boldsymbol{X})) \end{array} \Rightarrow \boldsymbol{y} \in \boldsymbol{X}\right)$, respectively;
  $\boldsymbol{x} = \boldsymbol{z} - 2$ stands for $\exists\boldsymbol{y}(\mathrm{succ}(\boldsymbol{z}, \boldsymbol{y}) \wedge \mathrm{succ}(\boldsymbol{y}, \boldsymbol{x}))$;
  the symbol $\nexists$ abbreviates $\neg\exists$.

- An MSO formula is interpreted over a string $w \in \Sigma^+$,[1] with respect to assignments $v_1 : \mathcal{V}_1 \to \{1, \dots |w|\}$ and $v_2 : \mathcal{V}_2 \to \mathcal{P}(\{1, \dots |w|\})$, in the following way.

  - $w, v_1, v_2 \models c(\boldsymbol{x})$ iff $w = w_1cw_2$ and $|w_1| + 1 = v_1(\boldsymbol{x})$.
  - $w, v_1, v_2 \models \boldsymbol{x} \in \boldsymbol{X}$ iff $v_1(\boldsymbol{x}) \in v_2(\boldsymbol{X})$.
  - $w, v_1, v_2 \models \boldsymbol{x} \le \boldsymbol{y}$ iff $v_1(\boldsymbol{x}) \le v_1(\boldsymbol{y})$.
  - $w, v_1, v_2 \models \neg\varphi$ iff $w, v_1, v_2 \not\models \varphi$.
  - $w, v_1, v_2 \models \varphi_1 \vee \varphi_2$ iff $w, v_1, v_2 \models \varphi_1$ or $w, v_1, v_2 \models \varphi_2$.
  - $w, v_1, v_2 \models \exists\boldsymbol{x}.\varphi$ iff $w, v_1', v_2 \models \varphi$, for some v_1' with $v_1'(\boldsymbol{y}) = v_1(\boldsymbol{y})$ for all $\boldsymbol{y} \in \mathcal{V}_1 \setminus \{\boldsymbol{x}\}$.
  - $w, v_1, v_2 \models \exists\boldsymbol{X}.\varphi$ iff $w, v_1, v_2' \models \varphi$, for some v_2' with $v_2'(\boldsymbol{Y}) = v_2(\boldsymbol{Y})$ for all $\boldsymbol{Y} \in \mathcal{V}_2 \setminus \{\boldsymbol{X}\}$.

  To improve readability, we will drop v_1, v_2 from the notation whenever there is no risk of ambiguity.

- A sentence is a closed formula of the MSO logic. For a given sentence φ, the language $L(\varphi)$ is defined as

  $$L(\varphi) = \{w \in \Sigma^+ \mid w \models \varphi\}.$$

For instance formula $\forall\boldsymbol{x}, \boldsymbol{y}(a(\boldsymbol{x}) \wedge \mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow b(\boldsymbol{y}))$ defines the language of strings where every occurrence of character a is immediately followed by an occurrence of b.

The original seminal result by Büchi and the others is synthesized by the following theorem.

Theorem 2.1. *A language L is regular iff there exists a sentence φ in the above MSO logic such that $L = L(\varphi)$.*

The proof of the theorem is constructive, i.e., it provides an algorithmic procedure that, for a given FSA builds an equivalent sentence in the logic, and conversely; next we offer an intuitive explanation of the construction, referring the reader to, e.g., [22] for a complete and detailed proof.

From FSA to MSO logic

The key idea of the construction consists in representing each state q of the automaton as a second order variable $\boldsymbol{X}_q$, which is the set of all string's positions where the machine is in state q. Without loss of generality we assume the automaton to be deterministic, and that $Q = \{0, 1, \dots, m\}$, with 0 initial, for some m. Then we encode the definition of the FSA recognizing L as the conjunction of several clauses each one representing a part of the FSA definition:

- The transition $\delta(q_i, a) = q_j$ is formalized by $\forall\boldsymbol{x}, \boldsymbol{y}(\boldsymbol{x} \in \boldsymbol{X}_i \wedge a(\boldsymbol{x}) \wedge \mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \boldsymbol{y} \in \boldsymbol{X}_j)$.
- The fact that the machine starts in state 0 is represented as $\exists\boldsymbol{z}(\nexists\boldsymbol{x}(\mathrm{succ}(\boldsymbol{x}, \boldsymbol{z})) \wedge \boldsymbol{z} \in \boldsymbol{X}_0)$.
- Since the automaton is deterministic, for each pair of distinct second order variables $\boldsymbol{X}_i$ and $\boldsymbol{X}_j$ we need the subformula $\nexists\boldsymbol{y}(\boldsymbol{y} \in \boldsymbol{X}_i \wedge \boldsymbol{y} \in \boldsymbol{X}_j)$.
- Acceptance by the automanton, i.e. $\delta(q_i, a) \in F$, is formalized by: $\exists\boldsymbol{y}(\nexists\boldsymbol{x}(\mathrm{succ}(\boldsymbol{y}, \boldsymbol{x})) \wedge \boldsymbol{y} \in \boldsymbol{X}_i \wedge a(\boldsymbol{y}))$.
- Finally the whole language L is the set of strings that satisfy the global sentence $\exists\boldsymbol{X}_0, \boldsymbol{X}_1, \dots \boldsymbol{X}_m(\varphi)$, where φ is the conjunction of all the above clauses.

At this point it is not difficult to show that the set of strings satisfying the above global formula is exactly L.

From MSO logic to FSA

The construction in the opposite sense has been proposed in various versions in the literature. Here we summarize its main steps along the lines of [22]. First, the MSO sentence is translated into a standard form using only second-order variables, the $\subseteq$ predicate, and variables $\boldsymbol{W}_a$, for each $a \in \Sigma$, denoting the set of all the positions of the word containing the character a. Moreover, we use Succ, which has the same meaning of succ, but, syntactically, has second order variable arguments that are singletons. This simpler, equivalent logic, is defined by the following syntax:

$$\varphi := \boldsymbol{X} \subseteq \boldsymbol{W}_a \mid \boldsymbol{X} \subseteq \boldsymbol{Y} \mid \mathrm{Succ}(\boldsymbol{X}, \boldsymbol{Y}) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists\boldsymbol{X}(\varphi).$$

As before, we also use the standard abbreviations for, e.g. $\wedge$, $\forall$, $=$. To translate first order variables to second order variables we need to state that a (second order) variable is a singleton. Hence we introduce the abbreviation: $\mathrm{Sing}(\boldsymbol{X})$ for $\exists\boldsymbol{Y}(\boldsymbol{Y} \subseteq \boldsymbol{X} \wedge \boldsymbol{Y} \neq \boldsymbol{X} \wedge \nexists\boldsymbol{Z}(\boldsymbol{Z} \subseteq \boldsymbol{X} \wedge \boldsymbol{Z} \neq \boldsymbol{Y} \wedge \boldsymbol{Z} \neq \boldsymbol{X}))$. Then, $\mathrm{Succ}(\boldsymbol{X}, \boldsymbol{Y})$ is implicitly conjuncted with $\mathrm{Sing}(\boldsymbol{X}) \wedge \mathrm{Sing}(\boldsymbol{Y})$ and is therefore false whenever $\boldsymbol{X}$ or $\boldsymbol{Y}$ are not singletons.

The following step entails the inductive construction of the equivalent automaton. This is built by associating a single automaton to each elementary subformula and by composing them according to the structure of the global formula. This inductive approach requires to use open formulas. Hence, we are going to consider words on the alphabet $\Sigma \times \{0, 1\}^k$, so that $\boldsymbol{X}_1$, $\boldsymbol{X}_2$, $\dots \boldsymbol{X}_k$ are the free variables used in the formula; 1 in the, say, j-th component means that the considered position belongs to $\boldsymbol{X}_j$, 0 vice versa. For instance, if $w = (a, 0, 1)(a, 0, 0)(b, 1, 0)$, then $w \models \boldsymbol{X}_2 \subseteq \boldsymbol{W}_a$, $w \models \boldsymbol{X}_1 \subseteq \boldsymbol{W}_b$, with $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ singletons.

[1] When specifying languages by means of logic formulas, the empty string must be excluded because formulas refer to string positions.

Formula transformation

1. First order variables are translated in the following way: $\exists \boldsymbol{x}(\varphi(\boldsymbol{x}))$ becomes $\exists \boldsymbol{X}(\mathrm{Sing}(\boldsymbol{X}) \wedge \varphi'(\boldsymbol{X}))$, where φ' is the translation of φ, and $\boldsymbol{X}$ is a fresh variable.
2. Subformulas having the form $a(\boldsymbol{x})$, $\mathrm{succ}(\boldsymbol{x}, \boldsymbol{y})$ are translated into $\boldsymbol{X} \subseteq \boldsymbol{W}_a$, $\mathrm{Succ}(\boldsymbol{X}, \boldsymbol{Y})$, respectively.
3. The other parts remain the same.

Inductive construction of the automaton

We assume for simplicity that $\Sigma = \{a, b\}$, and that $k = 2$, i.e. two variables are used in the formula. Moreover we use the shortcut symbol $\circ$ to mean all possible values.

- The formula $\boldsymbol{X}_1 \subseteq \boldsymbol{X}_2$ is translated into an automaton that checks that there are 1's for the $\boldsymbol{X}_1$ component only in positions where there are also 1's for the $\boldsymbol{X}_2$ component (Figure 1 (a)).
- The formula $\boldsymbol{X}_1 \subseteq \boldsymbol{W}_a$ is analogous: the automaton checks that positions marked by 1 in the $\boldsymbol{X}_1$ component must have symbol a (Figure 1 (b)).
- The formula $\mathrm{Succ}(\boldsymbol{X}_1, \boldsymbol{X}_2)$ considers two singletons, and checks that the 1 for component $\boldsymbol{X}_1$ is immediately followed by a 1 for component $\boldsymbol{X}_2$ (Figure 1 (c)).
- Formulas inductively built with $\neg$ and $\vee$ are covered by the closure of regular languages w.r.t. complement and union, respectively.
- For $\exists$, we use the closure under alphabet projection: we start with an automaton with input alphabet, say, $\Sigma \times \{0, 1\}^2$, for the formula $\varphi(\boldsymbol{X}_1, \boldsymbol{X}_2)$; we need to define an automaton for the formula $\exists \boldsymbol{X}_1(\varphi(\boldsymbol{X}_1, \boldsymbol{X}_2))$. But in this case the alphabet is $\Sigma \times \{0, 1\}$, where the last component represents the only free remaining variable, i.e. $\boldsymbol{X}_2$.
The automaton $\mathcal{A}_\exists$ is built by starting from the one for $\varphi(\boldsymbol{X}_1, \boldsymbol{X}_2)$, and changing the transition labels from $(a, 0, 0)$ and $(a, 1, 0)$ to $(a, 0)$; $(a, 0, 1)$ and $(a, 1, 1)$ to $(a, 1)$, and those with b analogously. The main idea is that this last automaton nondeterministically "guesses" the quantified component (i.e. $\boldsymbol{X}_1$) when reading its input, and the resulting word $w \in (\Sigma\times\{0, 1\}^2)^*$ is such that $w \models \varphi(\boldsymbol{X}_1, \boldsymbol{X}_2)$. Thus, $\mathcal{A}_\exists$ recognizes $\exists \boldsymbol{X}_1(\varphi(\boldsymbol{X}_1, \boldsymbol{X}_2))$.

We refer the reader to the available literature for a full proof of equivalence between the logic formula and the constructed automaton. Here we illustrate the rationale of the above construction through the following example.

Example 2.2. *Consider the language $L = \{a, b\}^* aa \{a, b\}^*$: it consists of the strings satisfying the formula:*
$\varphi_L = \exists \boldsymbol{x} \exists \boldsymbol{y}(\mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \wedge a(\boldsymbol{x}) \wedge a(\boldsymbol{y}))$.

As seen before, first we translate this formula into a version using only second order variables: $\varphi'_L = \exists \boldsymbol{X}, \boldsymbol{Y}(\mathrm{Sing}(\boldsymbol{X}) \wedge \mathrm{Sing}(\boldsymbol{Y}) \wedge \mathrm{Succ}(\boldsymbol{X}, \boldsymbol{Y}) \wedge \boldsymbol{X} \subseteq \boldsymbol{W}_a \wedge \boldsymbol{Y} \subseteq \boldsymbol{W}_a)$.

The automata for $\mathrm{Sing}(\boldsymbol{X})$ *and* $\mathrm{Sing}(\boldsymbol{Y})$ *are depicted in Figure 2; they could also be obtained by expanding the definition of* Sing *and then projecting the quantified variables.*

By intersecting the automata for $\mathrm{Sing}(\boldsymbol{X})$, $\mathrm{Sing}(\boldsymbol{Y})$, $\mathrm{Succ}(\boldsymbol{X}, \boldsymbol{Y})$ *we obtain an automaton which is identical to the one we defined for translating formula* $\mathrm{Succ}(\boldsymbol{X}_1, \boldsymbol{X}_2)$*, where here $\boldsymbol{X}$ takes the role of $\boldsymbol{X}_1$ and $\boldsymbol{Y}$ of $\boldsymbol{X}_2$. Combining it with those for $\boldsymbol{X} \subseteq W_a$ and $\boldsymbol{Y} \subseteq W_a$ produces the automaton of Figure 3.*

Finally, by projecting on the quantified variables $\boldsymbol{X}$ and $\boldsymbol{Y}$ we obtain the automaton for L, given in Figure 4.

The logical characterization of a class of languages, together with the decidability of the containment problem, is the main door towards automatic verification techniques. Suppose that a logic formalism $\mathfrak{L}$ is recursively equivalent to an automaton family $\mathfrak{A}$; then, one can use a formula $\varphi_{\mathfrak{L}}$ of $\mathfrak{L}$ to specify the requirements of a given system and an abstract machine $\mathcal{A}$ in $\mathfrak{A}$ to implement the desired system: the correctness of the design defined by $\mathcal{A}$ w.r.t. to the requirements stated by $\varphi_{\mathfrak{L}}$ is therefore formalized as $L(\mathcal{A}) \subseteq L(\varphi_{\mathfrak{L}})$, i.e., all behaviors realized by the machine are also satisfying the requirements. This is just the case with FSA and MSO logic for RL.

Unfortunately, known theoretical lower-bounds state that the decision of the above containment problem is PSPACE complete and therefore intractable in general. The recent striking success of model-checking [2], however, has produced many refined results that explain how and when practical tools can produce results of "acceptable complexity" (the term "acceptable" is context-dependent since in some cases even running times of the order of hours or weeks can be accepted). In a nutshell, normally –and roughly– it is accepted a lower expressive power of the adopted logic, typically linear temporal logic, to achieve a complexity that is "only exponential" in the size of the logic formulas, whereas the worst case complexity for MSO logic can be even a non-elementary function [23] [2]. In any case, our interest in this paper is not on the complexity issues but is focused on the equivalence between automata recognizers and MSO logics that leads to the decidability of the above fundamental containment problem.

3. Context-free Languages

Context-free languages (CFL), with their generating context-free grammars (CFG) and recognizing pushdown automata (PDA), are, together with RL, the most important chapter in the literature of formal languages. CFG have been introduced by Noam Chomsky in the 1950s as a formalism to capture the syntax of natural languages. Independently, essentially the same formalism has been developed to formalize the syntax of the first high level programming language, FORTRAN; in fact it is also referred to as Backus-Naur form (BNF) honoring the chief scientists of the team that developed FORTRAN and its first compiler. It is certainly no surprise that the same formalism has

[2] There are, however, a few noticeable cases of tools that run satisfactorily at least in some particular cases of properties expressed in MSO logic [24]

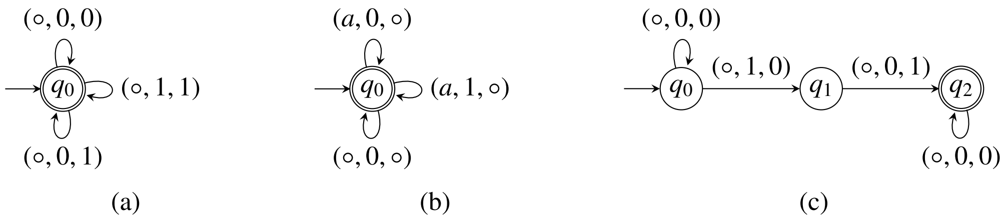


Figure 1: Automata for the construction from MSO logic to FSA.

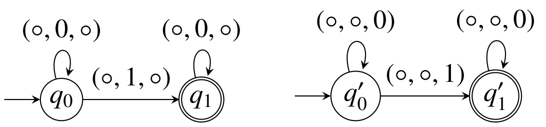


Figure 2: Automata for Sing(X) and Sing(Y).

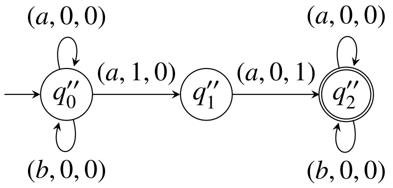


Figure 3: Automaton for the conjunction of Sing(X), Sing(Y), Succ(X, Y), $X \subseteq W_a$, $Y \subseteq W_a$.

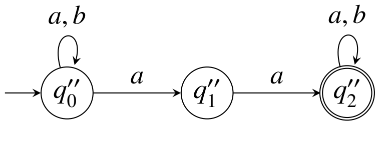


Figure 4: Automaton for $L = \{a, b\}^* aa\{a, b\}^*$.

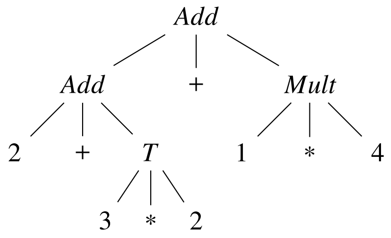


Figure 5: A tree structure that shows the precedence of multiplication over addition in arithmetic expressions.

been exploited to describe the syntactic aspects of both natural and high level programming languages, since the latter ones have exactly the purpose to make algorithm specification not only machine executable but also similar to human description.

The distinguishing feature of both natural and programming languages is that complex sentences can be built by combining simpler ones in an a priori unlimited hierarchy: for instance a conditional sentence is the composition of a clause specifying a condition with one or two sub-sentences specifying what to do if the condition holds, and possibly what else to do if it does not hold. Such a typical nesting of sentences suggests a natural representation of their *structure* in the form of a tree shape. The possibility of giving a sentence a tree structure which hints at its semantics is a sharp departure from the rigid linear structure of regular languages. As an example, consider a simple arithmetic expression consisting of a sequence of operands with either a + or a $*$ within any pair of them, as in $2+3*2+1*4$. Sentences of this type can be easily generated by, e.g., the following regular grammar:

$$\begin{aligned} S &\rightarrow 1 \mid 2 \mid \ldots 0 \mid 1A \mid 2A \mid \ldots 0A \\ A &\rightarrow +S \mid *S \end{aligned}$$

However, if we compute the value of the above expression by following the linear structure given to it by the grammar either by associating the sum and the multiply operations to the left or to the right we would obtain, respectively, 44 or 20 which is not the way we learned to compute the value of the expression at school. On the contrary, we first compute the multiply operations and then the sum of the three partial results, thus obtaining 12; this, again, suggests to associate the semantics of the sentence –in this case the value of the expression– with a syntactic structure that is more appropriately represented by a tree, as suggested in Figure 5, than by a flat sequence of symbols.

The following example illustrates how CFG associate to the

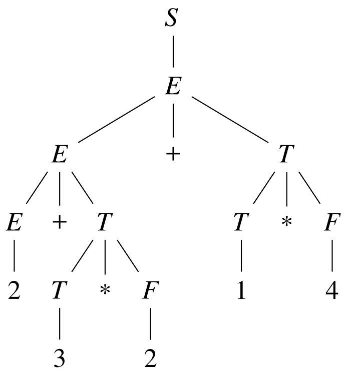


Figure 6: A syntax-tree produced by grammar GAE_1.

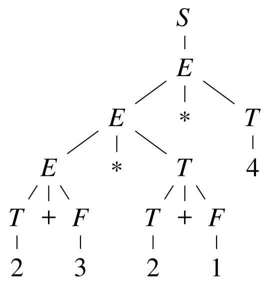


Figure 7: A tree that reverts the traditional precedence between arithmetic operators.

strings they generate a tree-like structure, which, for this reason, is named *syntax-tree* of the sentence.

Example 3.1. *The following CF grammar GAE_1 generates the same numerical arithmetic expressions as those generated by the above regular grammar but assigns them the appropriate structure exemplified in Figure 5* [3].

Notice that in the following, for the sake of simplicity, in arithmetic expressions we will make use of a unique terminal symbol e to denote any numerical value.

$$\begin{aligned} GAE_1: \quad & S \rightarrow E \mid T \mid F \\ & E \rightarrow E + T \mid T * F \mid e \\ & T \rightarrow T * F \mid e \\ & F \rightarrow e \end{aligned}$$

It is an easy exercise augmenting the above grammar to let it generate more general arithmetic expressions including more arithmetic operators, parenthesized sub-expressions, symbolic operands besides numerical ones, etc.

Consider now the following slight modification GAE_2 of GAE_1:

$$\begin{aligned} GAE_2: \quad & S \rightarrow E \mid T \mid F \\ & E \rightarrow E * T \mid T + F \mid e \\ & T \rightarrow T + F \mid e \\ & F \rightarrow e \end{aligned}$$

GAE_1 and GAE_2 are equivalent in that they generate the same language; however, GAE_2 assigns to the string $2 + 3 * 2 + 1 * 4$ the tree represented in Figure 7: if we executed the operations of the string in the order suggested by the tree –first the lower ones so that their result is used as an operand for the higher ones– then we would obtain the value 60 *which is not the "right one"* 12.

Last, the grammar:

$$GAE_{amb}: \quad S \rightarrow S * S \mid S + S \mid e$$

is equivalent to GAE_2 and GAE_1 as well, but it can generate the same string with different syntax-trees including, in the case of string $2+3*2+1*4$, those of Figures 6 and 7, up to a renaming of the non-terminals. In such a case we say that the grammar is ambiguous.

The above examples, inspired by the intuition of arithmetic expressions, further emphasize the strong connection between the structure given to a sentence by the syntax-tree and the semantics of the sentence: as it is the case with natural languages too, an ambiguous sentence may exhibit different meanings.

The examples also show that, whereas a sentence of a regular language has a fixed –either right or left-linear– structure, CFL sentences have a tree structure which in general, is not immediately "visible" by looking at the sentence, which is the frontier of its associated tree. As a consequence, in the case of CFL, analyzing a sentence is often not only a matter of deciding whether it belongs to a given language, but in many cases it is also necessary to build the syntax-tree(s) associated with it: its structure often drives the evaluation of its semantics as intuitively shown in the examples of arithmetic expressions and systematically applied in all major applications of CFL, including designing their compilers. The activity of jointly accepting or rejecting a sentence and, in case of acceptance, building its associated syntax-tree(s) is usually called *parsing*.

Next, we summarize the major and well-known properties of CFL and compare them with the RL ones, when appropriate; subsequently, in two separate subsections, we face two major problems for this class of languages, namely, parsing and logic characterization.

- All major families of formal languages are associated with corresponding families of generating grammars and recognizing automata. CFL are generated by CFG and recognized by pushdown automata (PDA) for which we adopt the notation given in Table 1. Furthermore the equivalence between CFG and PDA is recursive.

- CFL strictly include RL and enjoy a *pumping lemma* which naturally extends the homonymous lemma for RL.

- Unlike RL, nondeterministic PDA have a greater recognition power than their deterministic counterpart (DPDA); thus, since CFL are the class of languages recognized by general PDA, the class of deterministic CFL (DCFL), i.e., that recognized by DPDA, is strictly included in CFL (and strictly includes RL).

[3] More precisely, the complete syntax tree associated with the sentence of Figure 5 is the one depicted in Figure 6.

- CFL are closed under union, concatenation, Kleene*, homomorphism, inverse homomorphism, but not under complement and intersection. DCFL are closed under complement but not under union, intersection, concatenation, Kleene*, homomorphism, inverse homomorphism.
- Emptiness and infiniteness are decidable for CFL (thanks to the pumping lemma). Containment is undecidable for DCFL, but equivalence is decidable for DCFL and undecidable for CFL.

3.1. Parsing context-free languages

PDA are recursively equivalent to CFG; they, however, simply decide whether or not a string belongs to their language: if instead we want to build the syntax-tree(s) associated with input string –if accepted– we need an abstract machine provided with an output mechanism. The traditional way to obtain this goal is augmenting the recognizing automata with an output alphabet and extending the domain of the transition relation accordingly: formally, whereas $\delta \subseteq_F Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^*$ for a PDA, in a *pushdown transducer* (PDT): $\delta \subseteq_F Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^* \times O^*$ where O is the new output alphabet. Thus, at every move the PDT outputs a string in O^* which is concatenated to the one produced so far. The rest of PDT behavior, including acceptance, is the same as the underlying PDA.

We illustrate how PDT can produce a (suitable representation of) CFG syntax-trees through the following example.

Example 3.2. *Let's go back to grammar* GAE_1 *and number its productions consecutively; e.g.*
$S \rightarrow E \mid T \mid F$ *are* $\#1, \#2, \#3,$
$E \rightarrow E + T \mid T * F \mid e$ *are* $\#4, \#5, \#6$
respectively, etc.

Consider the following PDA $\mathcal{A}$*:*
$Q = \{q_0, q_1, q_F\}$, $\Gamma = \{E, T, F, +, *, e, Z_0\}$,
$\delta = \{(q_0, Z_0, \varepsilon, q_1, EZ_0), (q_0, Z_0, \varepsilon, q_1, TZ_0), (q_0, Z_0, \varepsilon, q_1, FZ_0)\} \cup$
$\{(q_1, E, \varepsilon, q_1, E + T), (q_1, E, \varepsilon, q_1, T * F), (q_1, E, \varepsilon, q_1, e)\} \cup$
$\{(q_1, T, \varepsilon, q_1, T * F), (q_1, T, \varepsilon, q_1, e)\} \cup \{(q_1, F, \varepsilon, q_1, e)\} \cup$
$\{(q_1, +, +, q_1, \varepsilon), (q_1, *, *, q_1, \varepsilon), (q_1, e, e, q_1, \varepsilon)\} \cup$
$\{(q_1, Z_0, \varepsilon, q_F, \varepsilon)\}$.

It is easy to infer how it has been derived from GAE_1*: at any step of its behavior, if the top of the stack stores a nonterminal it is* nondeterministically *replaced by the corresponding rhs of a production of which it is the lhs, in such a way that the leftmost character of the rhs is put on top of the stack; if the top of the stack stores a terminal it is compared with the current input symbol and, if they match the stack is popped and the reading head advances.* q_0 *is used only for the initial move and* q_F *only for final acceptance.*

Consider now the PDT $\mathcal{T}$ *obtained from* $\mathcal{A}$ *by adding the output alphabet* $\{1, 2, ..., 9\}$*, i.e., the set of labels of* GAE_1 *productions and by enriching each element of* δ *whose symbol on top of the stack is a nonterminal with the label of the corresponding* GAE_1*'s production;* δ*'s elements whose symbol on top of the stack is a terminal do not output anything: e.g.,* $(q_1, T, \varepsilon, q_1, T*F)$ *becomes* $(q_1, T, \varepsilon, q_1, T*F, 7)$*,* $(q_1, T, \varepsilon, q_1, T * F)$ *becomes* $(q_1, T, \varepsilon, q_1, T * F, 6)$*,* $(q_1, *, *, q_1, \varepsilon)$ *becomes* $(q_1, *, *, q_1, \varepsilon, \varepsilon)$*, etc.*

If $\mathcal{T}$ *receives the input string* $e + e * e$*, it goes through the following sequence of configuration transitions*
$(e + e * e, q_0, Z_0) \vdash (e + e * e, q_1, EZ_0, 1) \vdash (e + e * e, q_1, E + TZ_0, 14) \vdash (e + e * e, q_1, e + TZ_0, 146) \vdash (+e * e, q_1, +TZ_0, 146) \vdash (e * e, q_1, TZ_0, 146) \vdash (e * e, q_1, T * FZ_0, 1467) \vdash (e * e, q_1, e * FZ_0, 14678) \vdash (*e, q_1, *FZ_0) \vdash (e, q_1, FZ_0, 14678) \vdash (e, q_1, eZ_0, 146789) \vdash (\varepsilon, q_1, Z_0) \vdash (\varepsilon, q_F, \varepsilon, 146789)$. *Therefore it accepts the string and produces the corresponding output* 146789. *Notice however, that* $\mathcal{T}$ *– and the underlying* $\mathcal{A}$*– are nondeterministic; thus, there are also several sequences of transitions beginning with the same* c_0 *that do not accept the input.*

It is immediate to conclude, from the above example, that the language accepted by $\mathcal{T}$ *is exactly* $L(GAE_1)$*. Notice also that, if we modify the syntax-tree of* GAE_1 *corresponding to the derivation of the string* $e + e * e$ *by replacing each internal node with the label of the production that rewrites it, the output produced by* $\mathcal{T}$ *is exactly the result of a post-order, leftmost visit of the tree (where the leaves have been erased.) Such a classic linearization of the tree is in natural one-to-one correspondence with the grammar's* leftmost derivation *of the frontier of the tree; a leftmost (resp. rightmost) derivation of a CFG grammar is such that, at every step the leftmost (resp. rightmost) nonterminal character is the lhs of the applied rule. It is immediate to verify that for every derivation of a CFG there are an equivalent leftmost and an equivalent rightmost one, i.e., derivations that produce the same terminal string.*[4]

We can therefore conclude that our PDA $\mathcal{A}$ *and PDT* $\mathcal{T}$*, which have been algorithmically derived from* GAE_1*, are, respectively, a* –nondeterministic– *recognizer and a* –nondeterministic– *parser for* $L(GAE_1)$*.*

The fact that the obtained recognizer/parser is nondeterministic deserves special attention. It is well-known that $L(GAE_1)$ is deterministic, despite the fact the $\mathcal{A}$ is not; on the other hand, unlike RL, DCFL are a strict subset of CFL. Thus, if we want to recognize or parse a generic CFL, in principle we must simulate all possible computations of a nondeterministic PDA (or PDT); this approach clearly raises a critical complexity issue.

For instance, consider the analysis of any string in Σ^* by the –necessarily nondeterministic– PDA accepting $L = \{ww^R \mid w \in \{a, b\}^*\}$: in the worst case at each move the automaton "splits" its computation in two branches by guessing whether it reached the middle of the input sentence or not; in the first case the computation proceeds deterministically to verify that from that point on the input is the mirror image of what has been read so far, whereas the other branch of the computation proceeds still waiting for the middle of the string and splitting again and again at each step. Thus, the total number of different computations equals the length of the input string, say n, and each of them has in turn an $O(n)$ length; therefore, simulating the behavior of such a nondeterministic machine by means of a deterministic

[4] This property does not hold for more general classes of grammars.

algorithm to check whether at least one of its possible computations accepts the input has an $O(n^2)$ total complexity.

The above example can be generalized [5] in the following way: on the one hand we have

Statement 1. *Every CFL can be recognized in real-time, i.e. in a number of moves equal to the length of the input sentence, by a, possibly nondeterministic, PDA.*

One way to prove the statement is articulated in two steps:
1) First, an original CFG generating L is transformed into the *Greibach normal form (GNF)*:

Definition 3.3. *A CFG is in Greibach normal form [17] iff the rhs of all its productions belongs to ΣV_N^*.*

The procedure given in [25] to transform any CF grammar into the normal form essentially is based on transforming any *left recursion*, i.e. a derivation such as $A \overset{*}{\Rightarrow} A\alpha$ into an equivalent right one $B \overset{*}{\Rightarrow} \alpha B$.
2) Starting from a grammar in GNF the procedure to build an equivalent PDA therefrom can be "optimized" by:

- restricting Γ to V_N only;
- when a symbol A is on top of the stack, a single move reads the next input symbol, say a, and –nondeterministically– replaces A in the stack with the suffix α of the rhs of a production $A \to a\alpha$, *if any* (otherwise the string is rejected).

Notice that such an automaton is real-time since there are no more ε-moves but, of course, it may still be nondeterministic. If the grammar in GNF is such that there are no two distinct productions of the type $A \to a\alpha$, $A \to a\beta$, then the automaton built in this way is a real-time DPDA that –enriched as a DPDT– is able to build leftmost derivations of the grammar.

On the other hand, Statement 1 leaves us with the natural question: "provided that purely nondeterministic machines are not physically available and at most can be approximated by parallel machines which however cannot exhibit an unbounded parallelism, how can we come up with some deterministic parsing algorithm for general CFL and which complexity can exhibit such an algorithm?". In general it is well-known that simulating a nondetermistic device with complexity[6] $f(n)$ by means of a deterministic algorithm may expose to the risk of even an exponential complexity $O(2^{f(n)})$. For this reason on the one hand many applications, e.g., compiler construction, have restricted their interest to the subfamily of DCFL; on the other hand intense research has been devoted to design efficient deterministic parsing algorithms for general CFL by departing from the approach of simulating PDA. The case of parsing DCFL will be examined in Section 3.1.1; parsing general CFL, instead, is of minor interest in this paper, thus we simply mention the two most famous and efficient of such algorithms, namely the one due to Cocke, Kasami, and Younger, usually referred to as CKY and described, e.g., in [17], and the one by Earley reported in [19]; they both have an $O(n^3)$ time complexity.[7]

To summarize, CFL are considerably more general than RL but they require parsing algorithms to assign a given sentence an appropriate (tree-shaped and usually not immediately visible) structure, and they lose several closure and decidability properties typical of the simpler RL. Not surprisingly, therefore, much, largely unrelated, research has been devoted to face both such challenges; in both cases major successes have been obtained by introducing suitable subclasses of the general language family: on the one hand parsing can be accomplished for DCFL much more efficiently than for nondeterministic ones (furthermore in many cases, such as e.g, for programming languages, nondeterminism and even ambiguity are features that are better to avoid than to pursue); on the other hand various subclasses of CFL have been defined that retain some or most of the properties of RL yet increasing their generative power. In the next subsection we summarize the major results obtained for parsing DCFL; in Section 4 instead we will introduce several different subclasses of CFL –typically structured ones– aimed at preserving important closure and decidability properties. So far the two goals have been pursued within different research areas and by means of different subfamilies of CFL. As anticipated in the introduction, however, we will see in Section 5 that one of such families allows for major improvements on both sides.

3.1.1. Parsing context-free languages deterministically

We have seen that, whereas any CFL can be recognized by a nondeterministic PDA –and parsed by a nondeterministic PDT– that operates in real-time, the best deterministic algorithms to parse general CFL, such as CKY and Early's ones have a $O(n^3)$ complexity, which is considered not acceptable in many fields such as programming language compilation.

For this and other reasons many application fields restrict their attention to DCFL; DPDA can be easily built, with no loss of generality, in such a way that they can operate in linear time, whether as pure recognizers or as parsers and language translators: it is sufficient, for any DPDA, to effectively transform it into an equivalent *loop-free* one, i.e. an automaton that does not perform more than a constant number, say k, of ε-moves before reading a character from the input or popping some element from the stack (see, e.g., [17]). In such a way the whole input x is analyzed in at most $h \cdot |x|$ moves, where h is a constant depending on k and the maximum length of the string that the automaton can push on the stack in a single move (both k and h can be effectively computed by the transformation into the loop-free form).

[5] We warn the reader that this generalization does not include the complexity bound $O(n^2)$ which refers only to the specific example, as it will be shown next.

[6] As usual we assume as the complexity of a nondeterministic machine the length of the shortest computation that accepts the input or of the longest one if the string is rejected.

[7] We also mention (from [17]) that in the literature there exists a variation of the CKY algorithm due to Valiant that is completely based on matrix multiplication and therefore has the same asymptotic complexity of this basic problem, at the present state of the art $O(n^{2.37})$.

In general, however, it is not possible to obtain a DPDA that recognizes its language in real-time. Consider, for instance, the language $L = \{a^m b^n c^n d^m \mid m, n \geq 1\} \cup \{a^m b^+ e d^m \mid m \geq 1\}$: intuitively, a DPDA recognizing L must first push the as onto the stack; then, it must also store on the stack the subsequent bs since it must be ready to compare their number with the following cs, if any; after reading the bs, however, if it reads the e it must necessarily pop all bs by means of n ε-moves before starting the comparison between the as and the ds.

Given that, in general, it is undecidable to state whether a CFL is deterministic or not [26], the problem of automatically building a deterministic automaton, *if any*, from a given CFG is not trivial and deserved much research. In this section we will briefly recall two major approaches to the problem of deterministic parsing. We do not go deep into their technicalities, however, because the families of languages they can analyze are not of much interest from the point of view of algebraic and logical characterizations. It is Section 5, instead, where we introduce a class of languages that allows for highly efficient deterministic parsing *and* exhibits practically all desirable algebraic and logic properties.

Top-down deterministic parsing

We have seen that any the CFG can be effectively transformed in GNF and observed that, if in such grammars there are no two distinct productions of the type $A \to a\alpha$, $A \to a\beta$, then the automaton built therefrom is deterministic. The above particular case has been generalized leading to the definition of *LL grammars*, so called because they allow to build deterministic parsers that scan the input left-to-right and produce the leftmost derivation thereof. Intuitively, an $LL(k)$ grammar is such that, for any leftmost derivation

$$S\#^k \overset{*}{\Rightarrow} xA\alpha\#^k$$

it is possible to decide deterministically which production to apply to rewrite nonterminal A by "looking ahead" at most k terminal characters of the input string that follows x.[8] Normally, the practical application of LL parsing is restricted to the case $k = 1$ to avoid memorizing and searching too large tables. In general this choice allows to cover a large portion of programming language syntax even if it is well-known that not all DCFL can be generated by LL grammars. For instance no LL grammar can generate the deterministic language $\{a^n b^n \mid n \geq 1\} \cup \{a^n c^n \mid n \geq 1\}$ since the decision on whether to join the a's with the b's or with the c's clearly requires an unbounded look-ahead.

Example 3.4. *The following grammar is a simple transformation of GAE_1 in LL form; notice that the original left recursion $E \Rightarrow E + T$ has been replaced by a right one $E \overset{*}{\Rightarrow} T + E$ and similarly for nonterminal T.*

$$\begin{array}{ll} GAE_{LL}: & S \to E\# \\ & E \to TE' \\ & E' \to +E \mid \varepsilon \\ & T \to FT' \\ & T' \to *T \mid \varepsilon \\ & F \to e. \end{array}$$

For instance, consider a PDA $\mathcal{A}_L$ derived form GAE_{LL} in the same way as the automaton $\mathcal{A}$ was derived from GAE_1 in Example 3.2, and examine how it analyzes the string $e * e + e$: after pushing deterministically onto the stack the nonterminal string $FT'E'$, with F on the top, $\mathcal{A}_L$ deterministically reads e and pops F since there is only one production in GAE_{LL} rewriting F. At this point it must choose whether to replace T' with $*T$ or simply erase it: by looking-ahead one more character, it finds a $*$ and therefore chooses the first option (otherwise it would have found either a $+$ or the $\#$). Then, the analysis proceeds similarly.

Bottom-up deterministic parsing

So far the PDA that we used to recognize or parse CFL operate in a top-down manner by trying to build leftmost derivation(s) that produce the input string starting from grammar's axiom. Trees can be traversed also in a bottom-up way, however; a typical way of doing so is visiting them in *leftmost post-order*, i.e. scanning their frontier left-to right and, as soon as a string of children is identified, writing them followed by their father, then proceeding recursively until the root is reached. For instance such a visit of the syntax-tree by which GAE_1 generates the string $e + e * e$ is $eE + eT * eFTES$ which is the reverse of the rightmost derivation $S \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * e \Rightarrow E + e * e \Rightarrow e + e * e$.

Although PDA are normally introduced in the literature in such a way that they are naturally oriented towards building leftmost derivations, it is not difficult to adapt their formalization in such a way that they model a bottom-up string analysis: the automaton starts reading the input left-to-right and pushes the read character on top of the stack; as soon as –by means of its finite memory control device– it realizes that a whole rhs is on top of the stack it replaces it by the corresponding lhs. If the automaton must also produce a parse of the input, it is sufficient to let it output, e.g., the label of the used production at the moment when the rhs is replaced by the corresponding lhs. For instance, in the case of the above string $e+e*e$ its output would be 689741. A formal definition of PDA operating in the above way is given, e.g., in [27] and reported in Section 4.3.2. This type of operating by a PDA is called *shift-reduce parsing* since it consists in *shifting* characters from the input to the stack and *reducing* them from a rhs to the corresponding lhs.

Not surprisingly, the "normal" behavior of such a bottom-up parser is, once again, nondeterministic: in our example once the rhs e is identified, the automaton must apply a reduction either to F or to T or to E. Even more critical is the choice that the automaton must take after having read the substring $e + e$ and having (if it did the correct reductions) $E + T$ on top

[8] The "tail" of k $\#$ characters is a simple trick to allow for the look ahead when the reading head of the automaton is close to the end of the input.

of the stack: in this case the string could be the rhs of the rule $E \rightarrow E+T$, and in that case the automaton should reduce it to E but the T on the top could also be the beginning of another rhs, i.e., $T * F$, and in such a case the automaton should go further by shifting more characters before doing any reduction; this is just the opposite situation of what happens with IDL (discussed in Subsection 4.2), where the current input symbol allows the machine to decide whether to apply a reduction or to further shift characters from the input.

"Determinizing" bottom-up parsers has been an intense research activity during the 1960's, as well as for their top-down counterpart: in most cases the main approach to solve the problem is the same as in the case of top-down parsing, namely "looking ahead" to some more characters –usually one–. In Section 5 we thoroughly examine one of the earliest practical deterministic bottom-up parsers and the class of languages they can recognize, namely Floyd's *operator precedence languages*. The study of these languages, however, has been abandoned after a while due to advent of a more powerful class of grammars –the LR ones, defined and investigated by Knuth [16], whose parsers proceed Left-to-right as well as the LL ones but produce (the reverse of) Rightmost derivations. LR grammars in fact, unlike LL and operator precedence ones, generate all DCFL.

3.2. Logic characterization of context-free languages

The lack of the basic closure properties also hampers a natural extension of the logic characterization of RL to CFL: in particular the classic inductive construction outlined in Section 2.1 strongly hinges on the correspondence between logical connectives and Boolean operations on sub-languages. Furthermore, the linear structure of RL allows any move of a FSA to depend only on the current state which is associated with a position $\boldsymbol{x}$ of the input string and on the input symbol located at position $\boldsymbol{x}+1$; the typical nested structure of CFL sentences, instead, imposes that the move of the PDA may depend on information stored in the stack, which in turn may depend on information read from the input much earlier than the current move.

Despite these difficulties some interesting results concerning a logic characterization of CFL have been obtained. In particular it is worth mentioning the characterization proposed in [28]. The key idea is to enrich the syntax of the second order logic with a *matching relation symbol M* which takes as arguments two string position symbols $\boldsymbol{x}$ and $\boldsymbol{y}$: a matching relation interpreting M must satisfy the following axioms:

- $M(\boldsymbol{x},\boldsymbol{y}) \Rightarrow \boldsymbol{x} < \boldsymbol{y}$: $\boldsymbol{y}$ always follows $\boldsymbol{x}$;
- $M(\boldsymbol{x},\boldsymbol{y}) \Rightarrow \nexists z(z \neq \boldsymbol{x} \land z \neq \boldsymbol{y} \land (M(\boldsymbol{x},z) \lor M(z,\boldsymbol{y}) \lor M(z,\boldsymbol{x}) \lor M(\boldsymbol{y},z)))$: M is one-to-one;
- $\forall \boldsymbol{x},\boldsymbol{y},z,\boldsymbol{w}((M(\boldsymbol{x},\boldsymbol{y}) \land M(z,\boldsymbol{w}) \land \boldsymbol{x} < z < \boldsymbol{y}) \Rightarrow \boldsymbol{x} < \boldsymbol{w} < \boldsymbol{y})$: M is *nested*, i.e., if we represent graphically $M(\boldsymbol{x},\boldsymbol{y})$ as an edge from $\boldsymbol{x}$ to $\boldsymbol{y}$ such edges cannot cross.

The matching relation is then used to represent the tree structure(s) associated with a CFL sentence: for instance consider the (ambiguous) grammar G_{amb}

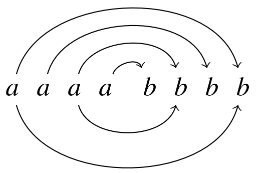


Figure 8: Two matching relations for *aaaabbbb*, one is depicted above and the other below the string.

$$G_{amb}: \quad \begin{aligned} S &\rightarrow A_1 \mid A_2 \\ A_1 &\rightarrow aaA_1bb \mid aabb \\ A_2 &\rightarrow aA_3b \\ A_3 &\rightarrow aA_2b \mid ab \end{aligned}$$

G_{amb} induces a natural matching relation between the positions of characters in its strings. For instance Figure 8 shows the two relations associated with the string *aaaabbbb*.

More generally we could state that for a grammar G and a sentence $x = a_1a_2...a_n \in L(G)$ with $\forall k, a_k \in \Sigma$, $(i,j) \in M$, when M is interpreted on x, iff $S \overset{*}{\Rightarrow}_G a_1a_2...a_{i-1}Aa_{j+1}...a_n \overset{*}{\Rightarrow}_G a_1a_2...a_n$. It is immediate to verify, however, that such a definition in general does not guarantee the above axioms for the matching relation: think e.g., to the previous grammar GAE_1 which generates arithmetic expressions. For this reason [28] adopts the *double Greibach normal form* (DGNF) which is an effective but non-trivial transformation of a generic CFG into an equivalent one where the first and last characters of any production are terminal.[9] It is now immediate to verify that a grammar in DGNF does produce a matching relation on its strings that satisfies all of its axioms.

Thanks to the new relation and the corresponding symbol [28] defines a second order logic CF$\mathcal{L}$ that characterizes CFL: the sentences of CF$\mathcal{L}$ are first-order formulas prefixed by a single existential quantifier applied to the second order variable M representing the matching relation. Thus, intuitively, a CF$\mathcal{L}$ sentence such as $\exists M(\phi)$ where M occurs free in ϕ and ϕ has no free occurrences of first-order variables is satisfied iff there is a structure defined by a suitable matching relation such that the positions that satisfy the occurrences of M in ϕ also satisfy the whole ϕ. For instance the sentence

$$\exists M, z \left(\begin{array}{l} \nexists \boldsymbol{x}(\mathrm{succ}(z,\boldsymbol{x})) \land M(0,z) \land \\ \forall \boldsymbol{x},\boldsymbol{y}(M(\boldsymbol{x},\boldsymbol{y}) \Rightarrow a(\boldsymbol{x}) \land b(\boldsymbol{y})) \land \\ \exists \boldsymbol{y} \forall \boldsymbol{x} \left(\begin{array}{c} (0 \leq \boldsymbol{x} < \boldsymbol{y} \Rightarrow a(\boldsymbol{x})) \land \\ (\boldsymbol{x} \geq \boldsymbol{y} \geq z \Rightarrow b(\boldsymbol{x})) \end{array} \right) \land \\ \left(\begin{array}{c} \forall \boldsymbol{x},\boldsymbol{y} \left(M(\boldsymbol{x},\boldsymbol{y}) \Rightarrow \begin{array}{c} (\boldsymbol{x} > 0 \Rightarrow M(\boldsymbol{x}-1,\boldsymbol{y}+1)) \land \\ (\boldsymbol{x} < \boldsymbol{y}-2 \Rightarrow M(\boldsymbol{x}+1,\boldsymbol{y}-1))) \end{array} \right) \\ \lor \\ \forall \boldsymbol{x},\boldsymbol{y} \left(M(\boldsymbol{x},\boldsymbol{y}) \Rightarrow \begin{array}{c} (\boldsymbol{x} > 1 \Rightarrow M(\boldsymbol{x}-2,\boldsymbol{y}+2) \land \\ \neg M(\boldsymbol{x}-1,\boldsymbol{y}+1)) \land \\ (\boldsymbol{x} < \boldsymbol{y}-4 \Rightarrow M(\boldsymbol{x}+2,\boldsymbol{y}-2) \land \\ \neg M(\boldsymbol{x}+1,\boldsymbol{y}-1)) \end{array} \right) \end{array} \right) \end{array} \right)$$

[9] To be more precise, the normal form introduced in [28] is further specialized, but for our introductory purposes it is sufficient to consider any DGNF. Notice also that the term DGNF is clearly a "symmetric variant" of the original GNF (see Subsection 3.1) but is not due to the same author and serves different purposes.

is satisfied by all and only the strings of $L(G_{amb})$ with both the M relations depicted in Figure 8.

After having defined the above logic, [28] proved its equivalence with CFL in a fairly natural way but with a few non-trivial technicalities: with an oversimplification, from a given CFG in DGNF a corresponding logic formula is built inductively in such a way that $M(\boldsymbol{x}, \boldsymbol{y})$ holds between the positions of leftmost and rightmost leaves of any subtree of a grammar's syntax-tree; conversely, from a given logic formula a tree-language, i.e., a set of trees, is built such that the frontiers of its trees are the sentences of a CFL. However, as the authors themselves admit, this result has a minor potential for practical applications due to the lack of closure under complementation. The need to resort to the DGNF puts severe constraints on the structures that can be associated with the strings, a priori excluding, e.g., linear structures typical of RL; nonetheless the introduction of the M relation opened the way for further important developments as we will show in the next sections.

4. Structured context-free languages

RL sentences have a fixed, right or left, linear structure; CFL sentences have a more general tree-structure, of which the linear one is a particular case, which normally is not immediately visible in the sentence and, in case of ambiguity, it may even happen that the same sentence has several structures. R. McNaughton, in his seminal paper [3], was probably the first one to have the intuition that, if we "embed the sentence structure in the sentence itself" in some sense making it visible from the frontier of the syntax-tree (as it happens implicitly with RL since their structure is fixed a priori), then many important properties of RL still hold for such special class of CFL.

Informally, we name such CFL *structured* or *visible structure* CFL. The first formal definition of this class given by McNaughton is that of *parenthesis languages*, where each subtree of the syntax-tree has a frontier embraced by a pair of parentheses; perhaps the most widely known case of such a language, at least in the field of programming languages is the case of LISP. Subsequently several other equivalent or similar definitions of structured languages, have been proposed in the literature; not surprisingly, an important role in this field is played by *tree-languages* and their related recognizing machines, *tree-automata*. Next we browse through a selection of such language families and their properties, starting from the seminal one by McNaughton.

4.1. Parenthesis grammars and languages

Definition 4.1. *For a given terminal alphabet* Σ *let* [,] *be two symbols* $\notin \Sigma$. *A parenthesis grammar (PG) with alphabet* $\Sigma \cup \{[,]\}$ *is a CFG whose productions are of the type* $A \to [\alpha]$, *with* $\alpha \in V^*$.

It is immediate to build a parenthesis grammar naturally associated with any CFG: for instance the following PG is derived from the GAE_1 of Example 3.1:

$$\begin{aligned} GAE_{[]}: \quad & S \to [E] \mid [T] \mid [F] \\ & E \to [E + T] \mid [T * F] \mid [e] \\ & T \to [T * F] \mid [e] \\ & F \to [e] \end{aligned}$$

It is also immediate to realize that, whereas GAE_1 does not make immediately visible in the sentence $e + e * e$ that $e * e$ is the frontier of a subtree of the whole syntax-tree, $GAE_{[]}$ generates $[[[e]+[[e]*[e]]]]$ (but not $[[[[e]+[e]]*[e]]]$), thus making the structure of the syntax-tree immediately visible in its parenthesized frontier.

Sometimes it is convenient, when building a PG from a normal one, to omit parentheses in the rhs of *renaming rules*, i.e., rules whose rhs reduces to a single nonterminal, since such rules clearly do not significantly affect the shape of the syntax-tree. In the above example such a convention would avoid the useless outermost pair of parentheses.

Historically, PL are the first subfamily of CFL that enjoys major properties typical of RL. In particular, in this paper we are interested in their closure under Boolean operations, which has several major benefits, including the decidability of the containment problem. The key milestones that allowed McNaughton to prove this closure are:

- The possibility to apply all operations within a *structure universe*, i.e., to a universe of syntax-trees rather than to the "flat" Σ^*; a natural way to define such a universe is based on the notion of *stencil* defined as follows.

  Definition 4.2. *A* stencil *of a terminal alphabet* Σ *is a string in* $(\Sigma \cup \{N\})^*$. *For any given CFG G* –not necessarily PG [10] – *a* stencil grammar G_S *is naturally derived therefrom by projecting any nonterminal of G into the unique nonterminal N, possibly erasing duplicated productions.*

  Let us now consider the stencils of PG.

  Definition 4.3. *For any PG G the* structure universe *of G is the –parenthesis– language* $L(G_S)$. *For any set of PG,* $\mathcal{PG}$, *its structure universe is the union of the structure universes of its members.*

  Clearly $L(G') \subseteq L(G_S)$ for any G' such that $G'_S = G_S$. For instance the structure universe of the parenthesized versions of the above grammars GAE_1 and GAE_2 is the language of the parenthesized version of GAE_{amb} which is also the stencil grammar of both of them, up to a renaming of nonterminal S to N.

- The possibility of building a normal form of any parenthesis grammar, called *backward deterministic normal form (BDNF) with no repeated rhs*, i.e., such that there

[10] We will see in Section 5 that this concept is important also to parse other classes of languages

are no two rules with the same rhs. Such a construction has been defined by McNaughton by assimilating PG's stencils to the terminal characters of regular grammars, whose nonterminal characters, in turn, are in one-to-one correspondence with the states of FSA. Then, on the basis of this analogy, the set of nonterminals of the grammar in normal form is the power set of the original one and the rule set is inductively built in the following, natural, way:

- Without loss of generality the procedure starts with a preliminary normal form where there are no renaming rules except for those that rewrite the axiom which are the only non-parenthesized ones; furthermore the axiom does not occur in any rhs and is the lhs exclusively of renaming rules (as, e.g. the parenthesized version of GAE_1, without the parentheses in the rules rewriting S).
- If there are several rules $\{S_1\} \rightarrow \alpha, \{S_2\} \rightarrow \alpha, ...$, then such rules are *replaced* by the unique rule $\{S_1 \cup S_2 \cup ...\} \rightarrow \alpha$. For instance if we have $A \rightarrow \alpha \mid \beta$ and $B \rightarrow \alpha \mid \gamma$, we remain with $A \rightarrow \beta$, $B \rightarrow \gamma$, and $\{A, B\} \rightarrow \alpha$
- for all rules where $S_1, S_2, ...$ occur the *new rules* where $\{S_1 \cup S_2 \cup ...\}$ replaces each occurrence of $S_1, S_2, ...$ are *added*.
- The new axiom S is created and, for each set S_j containing a nonterminal that is the rhs of a rule rewriting the axiom of the original grammar, the rule $S \rightarrow S_j$ is added.
- The procedure is iterated until no new nonterminals and no new rules are generated. At this point useless nonterminals and rules are erased.

On the basis of these two fundamental properties deriving the effective closure w.r.t. Boolean operations within a given universe of stencils is a natural extension of the well-known operations for RL (notice that RL are a special case of structured languages whose stencils are only linear, i.e., of the type aN, a (or Na, a) for $a \in \Sigma$) by further pursuing the analogy between grammar's nonterminals and automaton's states:

- Given a PG G in BDNF, the complement of $L(G)$ w.r.t. its structure universe is obtained in the following way:
  - Let $A_{err} \notin V_n$ be a new nonterminal and let h be the homomorphism that maps every element of $V_n \cup \{A_{err}\}$ into N and every element of Σ into itself; then *complete* G's production set P with productions $A_{err} \rightarrow \alpha$ for all $\alpha \in (V \cup \{A_{err}\})^*$ such that $h(\alpha)$ is in the production set of G_S but α is not in P;
  - The new (renaming) productions with S as the lhs have as rhs the complement w.r.t. V_n of the original ones plus $S \rightarrow A_{err}$.

  Notice that in general, if k is the homomorphism that erases the parentheses, it is not the case that $k(L(G_S) \setminus L(G)) = \Sigma^* \setminus k(L(G))$ not even if $k(L(G_S) = \Sigma^*$.
- The intersection between two languages sharing the same set of stencils (if not, build the union of the two sets) is obtained by building a new nonterminal alphabet that is the cartesian product of the original ones and composing the production sets in the natural way [11]. Then, by De Morgan's laws, the closure w.r.t. union is also obtained.

As usual, an immediate corollary of these closure properties is the decidability of the containment problem for two languages belonging to the same structure universe.

McNaughton also showed in [3] how any PG can be effectively transformed into an equivalent one with a minimum number of nonterminals: again, the procedure he provided is a natural but nontrivial extension of the well-known one for minimizing the states of FSA. We do not report, however, the technicalities of this result since it is not part of our main stream.

Given that the trees associated with sentences generated by PG are isomorphic to the sentences themselves, the parsing problem for such languages disappears and scales down to a simpler recognition problem as it happens for RL. Thus, rather than using the full power of general PDA or PDT for such a job, *tree-automata* have been proposed in the literature as a recognition device equivalent to PG as well as FSA are equivalent to regular grammars.

Intuitively, a tree-automaton (TA) traverses either top-down or bottom-up a labeled tree to decide whether to accept it or not, thus it is a tree-recognizer or tree-acceptor. Not surprisingly, the above constructions of the BDNF for PG and the proofs of closure properties strongly resemble the corresponding well-known constructions for FSA (and have been naturally rephrased in terms of TA). We do not go further into the exposition of TA, however; the interested reader can refer to the specialized literature, e.g, [4, 5], or to the larger version of this paper [20].

On the basis of the important results obtained by McNaughton in his seminal paper, many other families of CFL have been defined in several decades of literature with the general goal of extending (at least some of the) closure and decidability properties, and logic characterizations that make RL such a "nice" class despite its limits in terms of generative power. Most of such families maintain the key property of being "structured" in some generalized sense w.r.t. parenthesis languages.

In the following section we introduce so called *input-driven languages*, also known as *visibly pushdown languages* which received much attention in recent literature and exhibit a fairly complete set of properties imported from RL.

4.2. Input-driven or visibly pushdown languages

The concept of *input-driven CF language* has been introduced in [6] in the context of building efficient recognition algorithms for DCFL: according to [6] a DPDA is input-driven if the decision of the automaton whether to execute a *push move*, i.e. a move where a new symbol is stored on top of the stack, or a *pop move*, i.e. a move where the symbol on top of the stack

[11] Further obvious details are omitted.

is removed therefrom, or a move where the symbol on top of the stack is only updated, depends exclusively on the current input symbol rather than on the current state and the symbol on top of the stack as in the general case. Later, several equivalent definitions of the same type of pushdown automata, whether deterministic or not, have been proposed in the literature; among them, here we choose an early one proposed in [29], which better fits with the notation adopted in this paper than the later one in [8].

Definition 4.4. *Let the input alphabet Σ be partitioned into three disjoint alphabets, $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$, named, respectively, the* call *alphabet,* return *alphabet,* internal *alphabet. A* visibly pushdown automaton (VPA) *over $(\Sigma_c, \Sigma_r, \Sigma_i)$ is a tuple $(Q, I, \Gamma, Z_0, \delta, F)$, where*

- *Q is a finite set of states;*
- *$I \subseteq Q$ is the set of initial states;*
- *$F \subseteq Q$ is the set of final or accepting states;*
- *Γ is the finite stack alphabet;*
- *$Z_0 \in \Gamma$ is the special bottom stack symbol;*
- *δ is the transition relation, partitioned into three disjoint subrelations:*
  - *Call move: $\delta_c \subseteq Q \times \Sigma_c \times Q \times (\Gamma \setminus \{Z_0\})$,*
  - *Return move: $\delta_r \subseteq Q \times \Sigma_r \times \Gamma \times Q$,*
  - *Internal move: $\delta_i \subseteq Q \times \Sigma_i \times Q$.*

A VPA is deterministic iff I is a singleton, and δ_c, δ_r, δ_i are – possibly partial– functions:
$\delta_c : Q \times \Sigma_c \to Q \times (\Gamma \setminus \{Z_0\})$, $\delta_r : Q \times \Sigma_r \times \Gamma \to Q$, $\delta_i : Q \times \Sigma_i \to Q$.

The automaton configuration, the transition relation between two configurations, the acceptance of an input string, and the language recognized by the automaton are then defined in the usual way: for instance if the automaton reads a symbol a in Σ_c while is in the state q and has C on top of the stack, it pushes onto the stack a new symbol D and moves to state q' provided that (q, a, q', D) belongs to δ_c; conversely, if the automaton reads a symbol b in Σ_r while is in the state q and has D on top of the stack, it pops D from the stack and moves to state q' provided that (q, b, D, q') belongs to δ_r; in such a case we say the the two symbols read, respectively, during the call and the corresponding return move *match*. Notice that in this way the special symbol Z_0 can occur only at the bottom of the stack, during the computation. A language over an alphabet $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ recognized by some VPA is called a *visibly pushdown language (VPL)*.

The following remarks help put IDL, alias VPL, in perspective with other families of CFL.

- Once PDA are defined in a standard form where their moves either push a new symbol onto the stack or remove it therefrom or leave the stack unaffected, the two definitions of IDL and VPL are equivalent. Both names for this class of languages are adequate: on the one side, the attribute input-driven emphasizes that the type of automaton's move is determined exclusively on the basis of the current input symbol[12]; on the other side we can consider VPL as *structured languages* since the structure of their sentences is immediately *visible* thanks to the partitioning of Σ.
- VPL generalize McNaughton's parenthesis languages: open parentheses are a special case of Σ_c and closed ones of Σ_r; further generality is obtained by the possibility of performing an unbounded number of internal moves, actually "purely finite state" moves between two matching call and return moves and by the fact that VPA can accept unmatched return symbols at the beginning of a sentence as well as unmatched calls at its end; a motivation for the introduction of such a generality is offered by the need of modeling systems where a computation containing a sequence of procedure calls is suddenly interrupted by a special event such as an exception or an interrupt.
- Being VPL essentially structured languages, their corresponding automata are just recognizers rather than real parsers.
- VPa are *real-time*; in fact they read one input symbol for each move. We have mentioned that this property can be obtained for nondeterministic PDA recognizing any CFL but not for deterministic ones.
- Although VPL are studied mainly in connection with their recognizing automata, a class of CFG has also been defined that generates them (see e.g., [8]).

VPL have obtained much attention since they represent a major step in the path aimed at extending many, if not all, of the important properties of RL to structured CFL. They are closed w.r.t. all major language operations, namely the Boolean ones, concatenation, Kleene * and others; this also implies the decidability of the containment problem, which, together with the characterization in terms of a MSO logic, again extending the result originally stated for RL, opens the door to applications in the field of automatic verification.

A key step to achieve such important results is the possibility of effectively determinizing nondeterministic VPA. The basic idea is similar to the classic one that works for RL, i.e, to replace the uncertainty on the current state of a nondeterministic automaton with the subset of Q containing all states were the automaton could possibly abide. Unlike the case of FSA however, when the automaton executes a return move it is necessary to "match" the current state with the one that was entered at the time of the corresponding call; to do so the key idea is to "pair" the set of states nondeterministically reached at the time of a return move with those that were entered at the time of the corresponding call; intuitively, the latter ones are memorized

[12] We will see, however, that the same term can be interpreted in a more general way leading to larger classes of languages.

and propagated through the stack, whose alphabet is enriched with pairs of set states. As a result at the moment of the return it is possible to check whether some of the states memorized at the time of the call "match" with some of the states that can be currently held by the nondeterministic original automaton.

We do not go into the technical details of this construction, referring the reader to [29] for them; we just mention that, unlike the case of RL, the price to be paid in terms of size of the automaton to obtain a deterministic version from a nondetermistic one is $2^{O(s^2)}$, where s is the size of the original state set. In [8] the authors also proved that such a gap is unavoidable since there are VPL that are recognized by a nondeterministic VPA with a state set of cardinality s but are not recognized by any deterministic VPA with less than 2^{s^2} states. In Section 5.1 we will provide a similar proof of determinization for a more general class of automata.

Once a procedure to obtain a deterministic VPA from a nondeterministic one is available, closure w.r.t. Boolean operations follows quite naturally through the usual path already adopted for RL and parenthesis languages. Closure under other major language operations such as concatenation and Kleene * is also obtained without major difficulties but we do not report on it since those operations are not of major interest for this paper. Rather, we wish here to go back to the issue of logical characterization.

4.2.1. The logic characterization of visibly pushdown languages

We have seen in Section 3.2 that attempts to provide a logic characterization of general CFL produced only partial results due to the lack of closure properties and to the fact that CFL do not have an a priori fixed structure; in fact the characterization offered by [28] and reported here requires an existential quantification w.r.t. relation M that represents the structure of a string. Resorting to structured languages such as VPL instead allowed for a fairly natural generalization of the classical Büchi's result for RL.

The key steps to obtain this goal are [8]:

- Using the same relation M introduced in [28],[13] adding its symbol as a basic predicate to the MSO logic's syntax given in Section 2.1 for RL, and extending its interpretation in a fairly natural way: precisely $M(\boldsymbol{x}, \boldsymbol{y})$ holds between the positions $\boldsymbol{x}$ and $\boldsymbol{y}$ of two matching symbols; in the case of unmatched returns at the beginning of the sentence and unmatched calls at the end the conventional values $M(-\infty, \boldsymbol{y})$, $M(\boldsymbol{x}, +\infty)$ are stated. This turns out to be simpler and more effective in the case of structured languages since, being such languages a priori unambiguous (the structure of the sentence is the sentence itself), there is only one such relation among the string positions and therefore there is no need to quantify it. Furthermore the relation is obviously one-to-one with a harmless exception of $M(-\infty, \boldsymbol{y})$, $M(\boldsymbol{x}, +\infty)$.
- Repeating exactly the same path used for RL both in the construction of an automaton from a logic formula and in the converse one. This only requires the managing of the new M relation in both constructions; precisely:

In the construction from the MSO formula to VPA, the elementary automaton associated with the atomic formula $M(\boldsymbol{X}, \boldsymbol{Y})$, where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are the usual singleton second order variables for any pair of first order variables $\boldsymbol{x}$ and $\boldsymbol{y}$, is represented by the diagram of Figure 9 where, like in the same construction for RL, $\circ$ stands for any value of Σ for which the transition can be defined according to the alphabet partitioning, so that the automaton is deterministic, the second component of the triple corresponds to $\boldsymbol{X}$, and the third to $\boldsymbol{Y}$. We use here the following notation for depicting VPA: an arrow labeled a/B (resp., a, B, or a alone) is a push (resp., pop or internal) move.

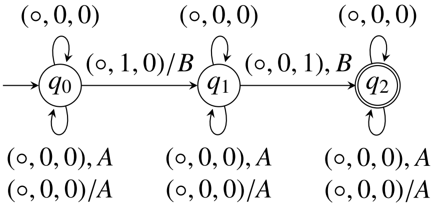


Figure 9: VPA associated with $M(\boldsymbol{X}, \boldsymbol{Y})$ atomic formula.

In the construction from the VPA to the MSO formula, besides variables $\boldsymbol{X}_i$ for encoding states, we also need variables to encode the stack. We introduce variables $\boldsymbol{C}_A$ and $\boldsymbol{R}_A$, for $A \in \Gamma$, to encode, respectively, the set of positions in which a *call* pushes A onto the stack, and in which a *return* pops A from the stack.

The following formula states that every pair $(\boldsymbol{x}, \boldsymbol{y})$ in M must belong, respectively, to exactly one $\boldsymbol{C}_A$ and exactly one $\boldsymbol{R}_A$:

$$\forall \boldsymbol{x}, \boldsymbol{y}\, (M(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \bigvee_{A \in \Gamma} \boldsymbol{x} \in \boldsymbol{C}_A \wedge \boldsymbol{y} \in \boldsymbol{R}_A) \wedge \\ \forall \boldsymbol{x} \bigwedge_{A \in \Gamma} \left(\begin{array}{c} \boldsymbol{x} \in \boldsymbol{C}_A \Rightarrow \neg \bigvee_{B \neq A} \boldsymbol{x} \in \boldsymbol{C}_B \\ \wedge \\ \boldsymbol{x} \in \boldsymbol{R}_A \Rightarrow \neg \bigvee_{B \neq A} \boldsymbol{x} \in \boldsymbol{R}_B \end{array} \right).$$

The following formula expresses the fact that, if the automaton is in state q_i and reads the symbol $a \in \Sigma_c$, then it moves to state q_j and pushes the symbol A onto the stack (without loss of generality, we assume that the original VPA is deterministic).

$$\forall \boldsymbol{x}, \boldsymbol{y} \left(\begin{array}{c} \boldsymbol{x} \in \boldsymbol{X}_i \wedge \mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \wedge a(\boldsymbol{y}) \\ \Rightarrow \\ \boldsymbol{y} \in \boldsymbol{C}_A \wedge \boldsymbol{y} \in \boldsymbol{X}_j \end{array} \right)$$

Symmetrically, return transitions $\delta_r(q_i, a, A) = q_j$ are formalized as follows:

$$\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \left(\begin{array}{c} \boldsymbol{y} \in \boldsymbol{X}_i \wedge \mathrm{succ}(\boldsymbol{y}, \boldsymbol{z}) \wedge M(\boldsymbol{x}, \boldsymbol{z}) \wedge \boldsymbol{z} \in \boldsymbol{R}_A \wedge a(\boldsymbol{z}) \\ \Rightarrow \\ \boldsymbol{z} \in \boldsymbol{X}_j \end{array} \right).$$

[13] Renamed *nesting relation* and denoted as $\rightsquigarrow$ or ν in later literature.

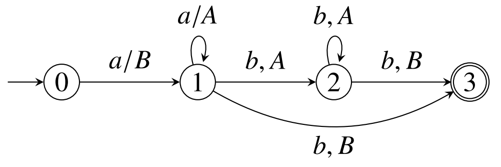


Figure 10: A VPA recognizing $\{a^n b^n \mid n > 0\}$.

> The remaining subformulas –for internal transitions, initial and final states– and the global formula quantifying second order variables, are the same as those for FSA.

Example 4.5. *Consider the alphabet $\Sigma = (\Sigma_c = \{a\}, \Sigma_r = \{b\}, \Sigma_i = \emptyset)$ and the VPA depicted in Figure 10.*

The MSO formula $\exists \boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{X}_3, \boldsymbol{C}_A, \boldsymbol{C}_B, \boldsymbol{R}_A, \boldsymbol{R}_B(\varphi_{\mathcal{A}} \wedge \varphi_M)$ is built on the basis of such an automaton, where φ_M is the conjunction of the formulas defined above, and $\varphi_{\mathcal{A}}$ is:

$$\begin{array}{l}
\exists \boldsymbol{z}(\nexists \boldsymbol{x}(\mathrm{succ}(\boldsymbol{x}, \boldsymbol{z})) \wedge \boldsymbol{z} \in \boldsymbol{X}_0) \wedge \\
\exists \boldsymbol{y}(\nexists \boldsymbol{x}(\mathrm{succ}(\boldsymbol{y}, \boldsymbol{x})) \wedge \boldsymbol{y} \in \boldsymbol{X}_3) \wedge \\
\forall \boldsymbol{x}, \boldsymbol{y}\, (\boldsymbol{x} \in \boldsymbol{X}_0 \wedge \mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \wedge a(\boldsymbol{y}) \Rightarrow \boldsymbol{y} \in \boldsymbol{C}_B \wedge \boldsymbol{y} \in \boldsymbol{X}_1) \wedge \\
\forall \boldsymbol{x}, \boldsymbol{y}\, (\boldsymbol{x} \in \boldsymbol{X}_1 \wedge \mathrm{succ}(\boldsymbol{x}, \boldsymbol{y}) \wedge a(\boldsymbol{y}) \Rightarrow \boldsymbol{y} \in \boldsymbol{C}_A \wedge \boldsymbol{y} \in \boldsymbol{X}_1) \wedge \\
\forall \boldsymbol{x}, \boldsymbol{y}, z\, (\boldsymbol{y} \in \boldsymbol{X}_1 \wedge \mathrm{succ}(\boldsymbol{y}, z) \wedge M(\boldsymbol{x}, z) \wedge z \in \boldsymbol{R}_A \wedge b(z) \Rightarrow z \in \boldsymbol{X}_2) \wedge \\
\forall \boldsymbol{x}, \boldsymbol{y}, z\, (\boldsymbol{y} \in \boldsymbol{X}_2 \wedge \mathrm{succ}(\boldsymbol{y}, z) \wedge M(\boldsymbol{x}, z) \wedge z \in \boldsymbol{R}_A \wedge b(z) \Rightarrow z \in \boldsymbol{X}_2) \wedge \\
\forall \boldsymbol{x}, \boldsymbol{y}, z\, (\boldsymbol{y} \in \boldsymbol{X}_2 \wedge \mathrm{succ}(\boldsymbol{y}, z) \wedge M(\boldsymbol{x}, z) \wedge z \in \boldsymbol{R}_B \wedge b(z) \Rightarrow z \in \boldsymbol{X}_3) \wedge \\
\forall \boldsymbol{x}, \boldsymbol{y}, z\, (\boldsymbol{y} \in \boldsymbol{X}_1 \wedge \mathrm{succ}(\boldsymbol{y}, z) \wedge M(\boldsymbol{x}, z) \wedge z \in \boldsymbol{R}_B \wedge b(z) \Rightarrow z \in \boldsymbol{X}_3)\,.
\end{array}$$

Other studies, e.g. [12], aimed at exploiting less powerful logics, such as variants of linear temporal ones to support a more practical automatic verification of VPL as it happened with great success with model-checking for RL; such attempts, however, are still in a preliminary stage and algorithmic model-checking is not the main focus of this paper; thus, we do not go deeper into this issue.

4.3. Other structured context-free languages

As we said, early work on parenthesis languages and tree-automata ignited many attempts to enlarge those classes of languages and to investigate their properties. Among them VPL have received much attention in the literature and in this paper, probably thanks to the completeness of the obtained results – closure properties and logic characterization. To give an idea of the vastness of this panorama and of the connected problems, and to help comparison among them, in this section we briefly mention a few more of such families with no attempt for exhaustiveness.

4.3.1. Balanced grammars

Balanced grammars (BG) have been proposed in [30] as a first approach to model mark-up languages such as XML by exploiting suitable extensions of parenthesis grammars. Basically a BG has a partitioned alphabet exactly in the same way as VPL; on the other hand any production of a BG has the form $A \to a\alpha b$ where $a \in \Sigma_c$, $b \in \Sigma_r$, and α is a regular expression[14] over $V_N \cup \Sigma_i$.

Since it is well-known that the use of regular expressions in the rhs of CF grammars can be replaced by a suitable expansion by using exclusively "normal" rules, we can immediately conclude that balanced languages, i.e. those generated by BG, are a proper subclass of VPL (e.g. they do not admit unmatched elements of Σ_c and Σ_r). Furthermore they are not closed under concatenation and Kleene * [30]; we are not aware of any logic characterization of these languages.

4.3.2. Height-deterministic languages

Height-deterministic languages, introduced in [27], are a more recent and fairly general way of describing CFL in terms of their structure. In a nutshell the hidden structure of a sentence is made explicit by making ε-moves visible, in that the original Σ alphabet is enriched as $\Sigma \cup \{\varepsilon\}$; if the original input string in Σ^* is transformed into one over $\Sigma \cup \{\varepsilon\}$ by inserting an explicit ε wherever a recognizing automaton executes an ε-move, we obtain a linear representation of the syntax-tree of the original sentence, so that the automaton can be used as a real parser. We illustrate such an approach by means of the following example.

Example 4.6. *Consider the language $L = L_1 \cup L_2$, with $L_1 = \{a^n b^n c^* \mid n \geq 0\}$, $L_2 = \{a^* b^n c^n \mid n \geq 0\}$ which is well-known to be inherently ambiguous since a string such as $a^n b^n c^n$ can be parsed both according to L_1's structure and according to L_2's one. A possible nondeterministic PDA $\mathcal{A}$ recognizing L could act as follows:*

- *$\mathcal{A}$ pushes all a's until it reaches the first b; at this point it makes a nondeterministic choice:*
  - *in one case it makes a "pause", i.e., an ε-move and enters a new state, say q_1;*
  - *in the other case it directly enters a different state, say q_2 with no "pause";*
- *from now on its behavior is deterministic; precisely:*
  - *in q_1 it checks that the number of b's equals the number of a's and then accepts any number of c's;*
  - *in q_2 it pushes the b's to verify that their number equals that of the c's.*

Thus, the two different behaviors of $\mathcal{A}$ when analyzing a string of the type $a^n b^n c^n$ would result in two different strings in the extended alphabet $\Sigma \cup \{\varepsilon\}$[15], namely $a^n \varepsilon b^n c^n$ and $a^n b^n c^n$; it is now simple to state a one-to-one correspondence between strings augmented with explicit ε and the different syntax-trees associated with the original input: in this example, $a^n \varepsilon b^n c^n$ corresponds to the structure of Figure 11 (a) and $a^n b^n c^n$ to that of Figure 11 (b). It is also easy to build other nondeterministic PDA recognizing L that "produce" different strings associated with different structures.

[14] A regular expression over a given alphabet V is built on the basis of alphabet's elements by applying union, concatenation, and Kleene * symbols; it is well-known that the class of languages definable by means of regular expressions is RL.

[15] This could be done explicitly by means of a nondeterministic transducer that outputs a special marker in correspondence of an ε-move, but we stick to the original [27] formalization where automata are used exclusively as acceptors without resorting to formalized transducers.

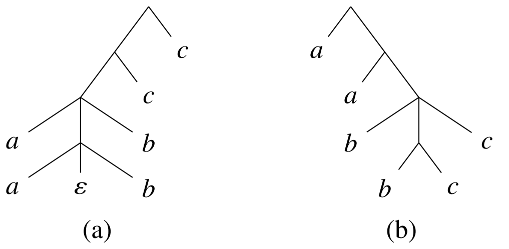


Figure 11: Different structures for $a^n \varepsilon b^n c^n$ (a) and $a^n b^n c^n$ (b), for $n = 2$.

Once the input string structures are made visible by introducing the explicit ε character, the characteristics of PDA, of their subclasses, and of the language subfamilies they recognize, are investigated by referring to the *heights of their stack*. Precisely:

- Any PDA is put into a *normalized form*, where
  - the δ relation is *complete*, i.e., it is defined in such a way that for every input string, whether accepted or not, the automaton scans the whole string: $\forall x \exists c (c_0 = (x, q_0, Z_0) \overset{*}{\longmapsto} c = (\varepsilon, q, \gamma))$;
  - every element of δ is exclusively in one of the forms: (q, A, o, q', AB), or (q, A, o, q', A), or $(q, A, o, q', \varepsilon)$, where $o \in (\Sigma \cup \{\varepsilon\})$;
  - for every $q \in Q$ all elements of δ having q as the first component either have $o \in \Sigma$ or $o = \varepsilon$, but not both of them.
- For any normalized $\mathcal{A}$ and word $w \in (\Sigma \cup \{\varepsilon\})^*$, $\mathcal{N}(\mathcal{A}, w)$ denotes the set of all stack heights reached by $\mathcal{A}$ after scanning w.
- $\mathcal{A}$ is said *height-deterministic* (HPDA) iff $\forall w \in (\Sigma \cup \{\varepsilon\})^*$, $|\mathcal{N}(\mathcal{A}, w)| = 1$.
- The family of height-deterministic PDA is named HPDA; similarly, HDPDA denotes the family of deterministic height-deterministic PDA, and HRDPDA that of deterministic, real-time (i.e., those that do not perform any ε-move) height-deterministic PDA. The same acronyms with a final L replacing the A denote the corresponding families of languages.

It is immediate to realize (Lemma 1 of [27]) that every PDA admits an equivalent normalized one. Example 4.6 provides an intuitive explanation that every PDA admits an equivalent HPDA (Theorem 1 of [27]); thus HPDL = CFL; also, any (normalized) DPDA is, by definition an HPDA and therefore a deterministic HPDA; thus HDPDL = DCFL. Finally, since every deterministic VPA is already in normalized form and is a real-time machine, VPL $\subset$ HRDPDL: the strict inclusion follows from the fact that $L = \{a^n b a^n\}$ cannot be recognized by a VPA since the same character a should belong both to Σ_c and to Σ_r.

Coupling the extension of the alphabet from Σ to $\Sigma \cup \{\varepsilon\}$ with the set $\mathcal{N}(\mathcal{A}, w)$ allows us to consider HPDL as a generalized kind of structured languages. As an intuitive explanation, let us go back to Example 4.6 and consider the two behaviors of $\mathcal{A}$ when parsing the string *aabbcc* once it has been "split" into *aabbcc* and *aaεbbcc*; the stack heights $\mathcal{N}(\mathcal{A}, w)$ for all their prefixes are, respectively: $(1, 2, 3, 4, 3, 2)$ and $(1, 2, 2, 1, 0, 0, 0)$ (if we do not count the bottom of the stack Z_0). In general, it is not difficult to associate every sequence of stack lengths during the parsing of an input string (in $(\Sigma \cup \{\varepsilon\})^*!$) with the syntax-tree visited by the HPDA.

As a consequence, the following fundamental definition of *synchronization* between HPDA can be interpreted as a form of structural equivalence.

Definition 4.7. *Two HPDA $\mathcal{A}$ and $\mathcal{B}$ are synchronized, denoted as $\mathcal{A} \sim \mathcal{B}$, iff $\forall w \in (\Sigma \cup \{\varepsilon\})^*$, $\mathcal{N}(\mathcal{A}, w) = \mathcal{N}(\mathcal{B}, w)$.*

It is immediate to realize that synchronization is an equivalence relation and therefore to associate an equivalence class $[\mathcal{A}]_\sim$ with every HPDA; we also denote as $\mathcal{A}$-HDPL the class of languages recognized by automata in $[\mathcal{A}]_\sim$. Then, in [27] the authors show that:

- For every *deterministic* HPDA $\mathcal{A}$ the class $\mathcal{A}$-HDPL is a Boolean algebra.[16]
- Real-time HPDA can be determinized (with a complexity of the same order as for VPA), so that the class of real-time HPDL coincides with HRDPDL.

On the other hand neither HRDPDL nor HDPDL are closed under concatenation and Kleene * [31] so that the gain obtained in terms of generative power w.r.t. VPL has a price in terms of closure properties. We are not aware of logic characterizations for this class of structured languages.

Let us also mention that other classes of structured languages based on some notion of synchronization have been studied in the literature; in particular, in [27] the authors compare their class with those of [32] and [33]. Finally, we acknowledge that our survey does not consider some logic characterization of tree or even graph languages which refer either to very specific families (such as, e.g. star-free tree languages [34]) and/or to an alphabet of basic elements, e.g., arcs connecting tree or graph nodes [35], which departs from the general framework of string (structured) languages.

5. Operator precedence languages

Operator precedence grammars (OPG) have been introduced by R. Floyd in his pioneering paper [13] with the goal of building efficient, deterministic, bottom-up parsers for programming languages. In doing so he was inspired by the hidden structure of arithmetic expressions which suggests to "give precedence" to, i.e, to execute first multiplicative operations w.r.t. the additive ones, as we illustrated through Example 3.1. Essentially, the goal of deterministic bottom-up parsing is to unambiguously decide when a complete rhs has been identified

[16] If the automaton is not deterministic only closures under union and intersection hold.

so that we can proceed with replacing it with the unique corresponding lhs with no risk to apply some roll-back if another possible reduction was the right one. Floyd achieved such a goal by suitably extending the notion of precedence between arithmetic operators to all grammar terminals. OPG obtained a considerable success thanks to their simplicity and to the efficiency of the parsers obtained therefrom; incidentally, some independent studies also uncovered interesting algebraic properties ([14]) which have been exploited in the field of grammar inference ([15]). As we anticipated in the introduction, however, the study of these grammars has been dismissed essentially because of the advent of other classes, such as the LR ones, which can generate all DCFL; OPG instead do not have such power as we will see soon, although they are able to cover most syntactic features of normal programming languages, and parsers based on OPG are still in practical use (see, e.g., [18]).

Only recently we renewed our interest in this class of grammars and languages for two different reasons that are the object of this survey. On the one side in fact, OPL, despite being apparently not structured, since they require and have been motivated by parsing, have shown rather surprising relations with various families of structured languages; as a consequence it has been possible to extend to them all the language properties investigated in the previous sections. On the other side, OPG enable parallelizing their parsing in a natural and efficient way, unlike what happens with other parsers which are compelled to operate in a strict left-to-right fashion, thus obtaining considerable speed-up thanks to the wide availability of modern parallel HW architectures.

Therefore, after having resumed the basic definitions and properties of OPG and their languages, we show, in Section 5.1, that they considerably increase the generative power of structured languages but, unlike the whole class of DCFL, they still enjoy all algebraic and logic properties that we investigated for such smaller classes. In Section 5.2 we show how parallel parsing is much better supported by this class of grammars than by the presently used ones.

Definition 5.1. *A grammar rule is in operator form if its rhs has no adjacent nonterminals; an operator grammar (OG) contains only such rules.*

Notice that the grammars considered in Example 3.1 are OG. Furthermore any CF grammar can be recursively transformed into an equivalent OG one [17].

Next, we introduce the notion of precedence relations between elements of Σ: we say that a is *equal in precedence* to b iff the two characters occur consecutively, or at most with one nonterminal in between, in some rhs of the grammar; in fact, when evaluating the relations between terminal characters for OPG, nonterminals are inessential, or "transparent". The letter a *yields precedence* to b iff a can occur at the immediate left of a subtree whose leftmost *terminal* character is b (again whether there is a nonterminal character at the left of b or not is inessential). Symmetrically, a *takes precedence* over b iff a occurs as the rightmost *terminal* character of a subtree and b is its following terminal character. These concepts are formally defined as follows.

	$+$	$*$	e
$+$	$\gtrdot$	$\lessdot$	$\lessdot$
$*$	$\gtrdot$	$\gtrdot$	$\lessdot$
e	$\gtrdot$	$\gtrdot$	

Figure 12: The OPM of the GAE_1 of Example 3.1.

Definition 5.2. *For an OG G and a nonterminal A, the left and right terminal sets are*

$$\mathcal{L}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} Ba\alpha\}$$
$$\mathcal{R}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} \alpha a B\} \text{ where } B \in V_N \cup \{\varepsilon\}.$$

The grammar name G will be omitted unless necessary to prevent confusion.

For an OG G, let α, β range over $(V_N \cup \Sigma)^*$ and $a, b \in \Sigma$. The three binary operator precedence (OP) relations are defined as follows:

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

For an OG G, the operator precedence matrix (OPM) $M = OPM(G)$ is a $|\Sigma| \times |\Sigma|$ array that, for each ordered pair (a, b), stores the set M_{ab} of OP relations holding between a and b.

For the grammar GAE_1 of Example 3.1 the left and right terminal sets of nonterminals E, T and F are, respectively:

$\mathcal{L}(E) = \{+, *, e\}$, $\mathcal{L}(T) = \{*, e\}$, $\mathcal{L}(F) = \{e\}$, $\mathcal{R}(E) = \{+, *, e\}$, $\mathcal{R}(T) = \{*, e\}$, and $\mathcal{R}(F) = \{e\}$.

Figure 12 displays the OPM associated with the grammar of GAE_1 of Example 3.1 where, for an ordered pair (a, b), a is one of the symbols shown in the first column of the matrix and b one of those occurring in its first row. Notice that, unlike the usual arithmetic relations denoted by similar symbols, the above precedence relations do not enjoy any of the transitive, symmetric, reflexive properties.

Definition 5.3. *An OG G is an* operator precedence *or* Floyd grammar *(*OPG*) iff $M = OPM(G)$ is a conflict-free matrix, i.e., $\forall a, b$, $|M_{ab}| \leq 1$. An* operator precedence language (OPL) *is a language generated by an OPG.*

A conflict-free matrix associates to every string at most only one structure, as we will show next; this aspect, paired with a way of deterministically choosing rules' rhs to be reduced, are the basis of Floyd's natural bottom-up deterministic parsing algorithm. This latter feature is enabled by introducing the *Fischer normal form.*

Definition 5.4. *An OPG is in* Fischer normal form *(FNF) iff it is invertible, i.e., no two rules have the same rhs, has no empty* *rules, i.e., rules whose rhs is ε, except possibly $S \to \varepsilon$, and no renaming rules, i.e, rules whose rhs is a single nonterminal, except possibly those whose lhs is S.*

For every OPG an equivalent one in FNF can effectively be built [36, 17]. The core part of the construction is avoiding repeated rhs, which is obtained in the same way as for parenthesis grammars (BDNF, see Section 4.1). A FNF grammar (manually) derived from GAE_1 of Example 3.1 is GAE_{FNF}:

$$\begin{aligned}
&S \to E \mid T \mid F \\
&E \to E + T \mid E + F \mid T + T \mid F + F \mid F + T \mid T + F \\
&T \to T * F \mid F * F \\
&F \to e
\end{aligned}$$

We can now see how the precedence relations of an OPG can drive the deterministic parsing of its sentences: consider again the sentence $e + e * e$; add to its boundaries the conventional symbol # which implicitly yields precedence to any terminal character and to which every terminal character takes precedence, and evaluate the precedence relations between pairs of consecutive symbols; they are displayed below:

$$\# \lessdot e \gtrdot + \lessdot e \gtrdot * \lessdot e \gtrdot \#.$$

A *handle* is a candidate rhs, and is included within a pair $\lessdot$, $\gtrdot$, with only $\doteq$ between consecutive terminals in it. The three occurrences of e enclosed within the pair $(\lessdot, \gtrdot)$, identify handles and are the rhs of production $F \to e$. Thanks to the FNF, there is no doubt on the corresponding lhs; therefore they can be reduced to F. Notice that such a reduction could be applied *in any order, possibly even in parallel*; this feature will be exploited later in Section 5.2.

More generally, let us consider the traditional left-to-right bottom-up OP parser, reported as Algorithm 1, in a slightly generalized variant that allows for nonterminals in the input string, and permits the presence of $(X, \gtrdot)$ pairs in the stack, facts that will be necessary in a parallel setting.

In the case of string $e + e * e$, Algorithm 1 is called with $\alpha = e + e * e\#$, $head = 1$, $end = 6$, $\mathcal{S} = (\#, \bot)$. The complete run is reported in Figure 13. We observe that, thanks to the FNF, the bottom-up shift-reduce algorithm works deterministically until the axiom is reached (more precisely, reductions stop at the single nonterminal E since the renaming rules of S are inessential for parsing) and a syntax-tree of the original sentence – represented by the mirror image of the rightmost derivation– is built. To avoid making the notation uselessly cumbersome, however, we omit specifying output operations both in the algorithm and in the table, since they are identical to the extension already described in Section 3.1 for general CF parsers.

This first introduction to OPG allows us to draw some early important properties thereof:

- In some sense OPL are *input-driven* even if they do not match exactly the definition of these languages: in fact, the decision of whether to apply a push operation (at the beginning of a rhs) or a shift one (while scanning a rhs) or a pop one (at the end of a rhs) depends only on terminal characters but not on a *single* one, as a look-ahead of one more terminal character is needed.[17]

step	$\mathcal{S}$	$\alpha(head)\ldots\#$	Y op X
0	$(\#, \bot)$	$e + e * e\#$	$\# \lessdot e$
1	$(e, \lessdot)(\#, \bot)$	$+e * e\#$	$e \gtrdot +$
2	$(F, \bot)(\#, \bot)$	$+e * e\#$	$\# \lessdot +$
3	$(+, \lessdot)(F, \bot)(\#, \bot)$	$e * e\#$	$+ \lessdot e$
4	$(e, \lessdot)(+, \lessdot)(F, \bot)(\#, \bot)$	$*e\#$	$e \gtrdot *$
5	$(F, \bot)(+, \lessdot)(F, \bot)(\#, \bot)$	$*e\#$	$+ \lessdot *$
6	$(*, \lessdot)(F, \bot)(+, \lessdot)(F, \bot)(\#, \bot)$	$e\#$	$* \lessdot e$
7	$(e, \lessdot)(*, \lessdot)(F, \bot)(+, \lessdot)(F, \bot)(\#, \bot)$	$\#$	$e \gtrdot \#$
8	$(F, \bot)(*, \lessdot)(F, \bot)(+, \lessdot)(F, \bot)(\#, \bot)$	$\#$	$* \gtrdot \#$
9	$(T, \bot)(+, \lessdot)(F, \bot)(\#, \bot)$	$\#$	$+ \gtrdot \#$
10	$(E, \bot)(\#, \bot)$	$\#$	$\# \doteq \#$

Figure 13: A run of Algorithm 1, with grammar GAE_{FNF} and input string $e + e * e$.

- The above characteristic is also a major reason why OPL, though allowing for efficient deterministic parsing of various practical programming languages [18, 37, 13], do not cover the whole family DCFL. Consider the language $L = \{0a^nb^n \mid n \geq 0\} \cup \{1a^nb^{2n} \mid n \geq 0\}$: a DPDA can easily "remember" the first character in its state; then push all the a's onto the stack and, when it reaches the bs decide whether to scan one or two bs for every a depending on its memory of the first read character. On the other hand, it is clear that any grammar generating L would necessarily exhibit some precedence conflict: intuitively, the string bb can be generated paired with a single a either by means of a single production $A \to aAbb$, which would generate the conflict $b \doteq b$ and $b \gtrdot b$, or by means of two consecutive steps $A \implies aBb$, $B \implies b$, which would generate the conflict $a \doteq b$, $a \lessdot b$.[18]

- We like to consider OPL as *structured languages* in a generalized sense since, once the OPM is given, the structure of their sentences is immediately defined and univocally determinable as it happens. e.g., with VPL once the alphabet is partitioned into call, return, and internal alphabet. However, we would be reluctant to label OPL as *visible* since there is a major difference between parenthesis-like terminals which make the language sentence isomorphic to its syntax-tree, and precedence relations which help *building* the tree but are computed only during the parsing. Indeed, not all of them are immediately visible in the original sentence: e.g., in some cases such as in the above sentence $\# \lessdot F + \lessdot F * F \gtrdot \#$ precedence relations are not even matched so that they can be assimilated to real parentheses only when they mark a complete

[17] As it happens in other deterministic parsers such as LL or LR ones (see Section 3.1.1).

[18] The above L is instead LL (see Section 3.1.1), while the language $\{a^nb^n \mid n \geq 1\} \cup \{a^nc^n \mid n \geq 1\}$ is OPL but not LL; thus, OPL and LL languages are uncomparable.

Algorithm 1 : OP-parsing($\alpha, head, end, \mathcal{S}$)

```
Remark. $\alpha \in V^*$ is the input string, head and end are integers marking respectively the first and the last character of the portion of $\alpha$ to be parsed; $\mathcal{S}$ is the initial stack contents, and contains pairs $(Z, p) \in (V \cup \{\#\}) \times \{\lessdot, \doteq, \gtrdot, \perp\}$. The $p$ component encodes the precedence relation found between two consecutive terminal symbols; thus it is always $\perp$ when $Z \in V_N$.
When the algorithm is called in sequential mode: $\alpha = \beta\#$, for some $\beta$, $head = 1$, $end = |\alpha|$, $\mathcal{S} = (\#, \perp)$.

1. Let $X = \alpha(head)$ and consider the precedence relation between the top-most terminal $Y$ found in $\mathcal{S}$ and $X$.
2. If $Y \lessdot X$ then push $(X, \lessdot)$; $head := head + 1$.
3. If $Y \doteq X$ then push $(X, \doteq)$; $head := head + 1$.
4. If $X \in V_N$ then push $(X, \perp)$; $head := head + 1$.
5. If $Y \gtrdot X$ then consider $\mathcal{S}$:
   (a) if $\mathcal{S}$ does not contain any $\lessdot$ then push $(X, \gtrdot)$; $head := head + 1$
   (b) else let $\mathcal{S} = (X_n, p_n) \dots (X_i, \lessdot)(X_{i-1}, p_{i-1}) \dots (X_0, p_0)$ where $\forall j, i < j \leq n, p_j \neq \lessdot$
       i. if $X_{i-1} \in V_N$ (hence $p_{i-1} = \perp$) and there exists a rule $A \to X_{i-1}X_i \dots X_n$ then replace $(X_n, p_n) \dots (X_i, \lessdot)(X_{i-1}, p_{i-1})$ in $\mathcal{S}$ with $(A, \perp)$;
       ii. if $X_{i-1} \in V_T \cup \{\#\}$ and there exists a rule $A \to X_i \dots X_n$ then replace $(X_n, p_n) \dots (X_i, \lessdot)$ in $\mathcal{S}$ with $(A, \perp)$;
       iii. otherwise start an error recovery procedure.
6. If ($head < end$) or ($head = end$ and $\mathcal{S} \neq (B, \perp)(a, \perp)$), for $B \in V_N, a \in \Sigma \cup \{\#\}$ then repeat from step (1); else return $\mathcal{S}$.
```

rhs. In summary, we would consider that OPL are structured (by the OPM) as well as PL (by explicit parentheses), VPL (by alphabet partitioning), and other families of languages; however, we would intuitively label them at most as "semi-visible" since making their structure visible requires some partial parsing, though not necessarily a complete recognition.

5.1. Algebraic and logic properties of operator precedence languages

OPL enjoy all algebraic and logic properties that have been illustrated in the previous sections for much less powerful families of structured languages.

As a first step we introduce the notion of a *chain* as a formal description of the intuitive concept of "semi-visible structure". To illustrate the following definitions and properties we will continue to make use of examples inspired by arithmetic expressions but we will enrich such expressions with, possibly nested, explicit parentheses as the visible part of their structure. For instance the following grammar GAE_P is a natural enrichment of previous GAE_1 to generate arithmetic expressions that involve parenthesized subexpressions (we use the slightly modified symbols '⦇' and '⦈' to avoid overloading with other uses of parentheses).

$$
\begin{aligned}
GAE_P: \quad & S \to E \mid T \mid F \\
& E \to E + T \mid T * F \mid e \mid \llparenthesis E \rrparenthesis \\
& T \to T * F \mid e \mid \llparenthesis E \rrparenthesis \\
& F \to e \mid \llparenthesis E \rrparenthesis
\end{aligned}
$$

Definition 5.5 (Operator precedence alphabet). *An* operator precedence (OP) alphabet *is a pair (Σ, M) where Σ is an alphabet and M is a conflict-free operator precedence matrix, i.e. a $|\Sigma \cup \{\#\}|^2$ array that associates at most one of the operator precedence relations: $\doteq$, $\lessdot$ or $\gtrdot$ with each ordered pair (a, b). As stated above the delimiter # yields precedence to other terminals and other terminals take precedence over it (with the special case $\# \doteq \#$ for the final reduction of renaming rules.) Since such relations are stated once and forever, we do not explicitly display them in OPM figures.*

If $M_{ab} = \{\circ\}$, with $\circ \in \{\lessdot, \doteq, \gtrdot\}$,we write $a \circ b$. For $u, v \in \Sigma^*$ we write $u \circ v$ if $u = xa$ and $v = by$ with $a \circ b$.

Definition 5.6 (Chains). *Let (Σ, M) be a precedence alphabet.*

- *A* simple chain *is a word $a_0a_1a_2 \dots a_na_{n+1}$, written as ${}^{a_0}[a_1a_2 \dots a_n]^{a_{n+1}}$, such that: $a_0, a_{n+1} \in \Sigma \cup \{\#\}$, $a_i \in \Sigma$ for every $i : 1 \leq i \leq n$, $M_{a_0a_{n+1}} \neq \emptyset$, and $a_0 \lessdot a_1 \doteq a_2 \dots a_{n-1} \doteq a_n \gtrdot a_{n+1}$.*
- *A* composed chain *is a word $a_0x_0a_1x_1a_2 \dots a_nx_na_{n+1}$, with $x_i \in \Sigma^*$, where ${}^{a_0}[a_1a_2 \dots a_n]^{a_{n+1}}$ is a simple chain, and either $x_i = \varepsilon$ or ${}^{a_i}[x_i]^{a_{i+1}}$ is a chain (simple or composed), for every $i : 0 \leq i \leq n$. Such a composed chain will be written as ${}^{a_0}[x_0a_1x_1a_2 \dots a_nx_n]^{a_{n+1}}$.*
- *The body of a chain ${}^{a}[x]^{b}$, simple or composed, is the word x.*

Example 5.7. *Figure 14 (a) depicts the OPM(GAE_P), whereas Figure 14 (b) represents the "semi-visible" structure induced by the operator precedence alphabet of grammar GAE_P for the expression $\#e + e * \llparenthesis e + e \rrparenthesis \#$: ${}^{\#}[e]^{+}$, ${}^{+}[e]^{*}$, ${}^{\llparenthesis}[e]^{+}$, ${}^{+}[e]^{\rrparenthesis}$ are simple chains and ${}^{\#}[x_0 + x_1]^{\#}$ with $x_0 = e$, $x_1 = e * \llparenthesis e + e \rrparenthesis$, ${}^{+}[y_0 * y_1]^{\#}$ with $y_0 = e$, $y_1 = \llparenthesis e + e \rrparenthesis$, ${}^{*}[\llparenthesis w_0 \rrparenthesis]^{\#}$ with $w_0 = e + e$, ${}^{\llparenthesis}[z_0 + z_1]^{\rrparenthesis}$ $z_0 = e$, $z_1 = e$, are composed chains.*

Notice that, thanks to the fact that the OPM is conflict-free, for any string in $\#\Sigma^*\#$, there is at most one way to build, inductively, a composed chain of (Σ, M).

Definition 5.8 (Compatible word). *A word w over (Σ, M) is* compatible *with M iff the two following conditions hold:*

- *For each pair of letters c, d, consecutive in w, $M_{cd} \neq \emptyset$;*
- *for each substring x of $\#w\#$ such that $x = a_0 x_0 a_1 x_1 a_2 \ldots a_n x_n a_{n+1}$, if $a_0 \lessdot a_1 \doteq a_2 \ldots a_{n-1} \doteq a_n \gtrdot a_{n+1}$ and, for every $0 \leq i \leq n$, either $x_i = \varepsilon$ or ${}^{a_i}[x_i]^{a_{i+1}}$ is a chain (simple or composed), then $M_{a_0 a_{n+1}} \neq \emptyset$.*

For instance, the word $e + e * \llparenthesis e + e \rrparenthesis$ is compatible with the operator precedence alphabet of grammar GAE_P, whereas $e + e * \llparenthesis e + e \rrparenthesis \llparenthesis e + e \rrparenthesis$ is not.

Thus, given an OP alphabet, the set of possible chains over Σ^* represents the universe of possible structured strings compatible with the given OPM.

5.1.1. Operator precedence automata

Despite abstract machines being the classical way to formalize recognition and parsing algorithms for any family of formal languages, and despite OPL having been invented just with the purpose of supporting deterministic parsing, their theoretical investigation has been abandoned before a family of automata completely equivalent to their generative grammars appeared in the literature. Only recently, when the numerous still unexplored benefits obtainable from this family appeared clear to us, we filled up this hole with the herewith resumed definition ([38]). The formal model presented in this paper is a "traditional" left-to-right automaton, although, as we already anticipated while illustrating Algorithm 1 and will thoroughly exploit in the Section 5.2 a distinguishing feature of OPL is that their parsing can be started in arbitrary positions with no harm nor loss of efficiency. This choice is explained by the need to extend and to compare results already reported for other language families. The original slightly more complicated version of this model was introduced in [39].

Definition 5.9 (Operator precedence automaton). *An* operator precedence automaton (OPA) *is a tuple $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ where:*

- *(Σ, M) is an operator precedence alphabet,*
- *Q is a set of states (disjoint from Σ),*
- *$I \subseteq Q$ is the set of initial states,*
- *$F \subseteq Q$ is the set of final states,*
- *$\delta \subseteq Q \times (\Sigma \cup Q) \times Q$ is the transition relation, which is the union of three disjoint relations:*

$$\delta_{shift} \subseteq Q \times \Sigma \times Q, \quad \delta_{push} \subseteq Q \times \Sigma \times Q, \quad \delta_{pop} \subseteq Q \times Q \times Q.$$

An OPA is deterministic iff

- *I is a singleton*
- *All three components of δ are –possibly partial– functions:*

$$\delta_{shift} : Q \times \Sigma \to Q, \; \delta_{push} : Q \times \Sigma \to Q, \; \delta_{pop} : Q \times Q \to Q.$$

We represent an OPA by a graph with Q as the set of vertices and $\Sigma \cup Q$ as the set of edge labelings. The edges of the graph are denoted by different shapes of arrows to distinguish the three types of transitions: there is an edge from state q to state p labeled by $a \in \Sigma$ denoted by a dashed (respectively, normal) arrow iff $(q, a, p) \in \delta_{\text{shift}}$ (respectively, $\in \delta_{\text{push}}$) and there is an edge from state q to state p labeled by $r \in Q$ and denoted by a double arrow iff $(q, r, p) \in \delta_{\text{pop}}$.

To define the semantics of the automaton, we need some new notations.

We use letters $p, q, p_i, q_i, \ldots$ to denote states in Q. Let Γ be $\Sigma \times Q$ and let $\Gamma' = \Gamma \cup \{Z_0\}$ be the *stack alphabet*; we denote symbols in Γ' as $[a,\ q]$ or Z_0. We set $symbol([a,\ q]) = a$, $symbol(Z_0) = \#$, and $state([a,\ q]) = q$. Given a stack contents $\Pi = \pi_n \ldots \pi_2 \pi_1 Z_0$, with $\pi_i \in \Gamma$, $n \geq 0$, we set $symbol(\Pi) = symbol(\pi_n)$ if $n \geq 1$, $symbol(\Pi) = \#$ if $n = 0$.

As usual, a *configuration* of an OPA is a triple $c = \langle w,\ q,\ \Pi \rangle$, where $w \in \Sigma^*\#$, $q \in Q$, and $\Pi \in \Gamma^* Z_0$.

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 $\langle ax,\ p,\ \Pi \rangle \vdash \langle x,\ q,\ [a,\ p]\Pi \rangle$, with $(p, a, q) \in \delta_{\text{push}}$;

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

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

Observe that shift and pop moves are never performed when the stack contains only Z_0.

Push and shift moves update the current state of the automaton according to the transition relations δ_{push} and δ_{shift}, respectively: push moves put a new element on 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*. Pop moves remove the element on top of the stack, and update the state of the automaton according to δ_{pop} on 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 moves the input symbol is used only to establish the $\gtrdot$ relation and it remains available for the following move.

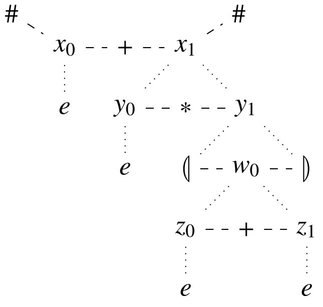
The language accepted by the automaton is defined as:

$$L(\mathcal{A}) = \left\{ x \mid \langle x\#,\ q_I,\ Z_0 \rangle \overset{*}{\vdash} \langle \#,\ q_F,\ Z_0 \rangle, q_I \in I, q_F \in F \right\}.$$

Example 5.10. *The OPA depicted in Figure 15 accepts the language of arithmetic expressions generated by grammar GAE_P. The same figure also shows the syntax-tree of the sentence $e + e * \llparenthesis e + e \rrparenthesis$ and an accepting computation on this input.*

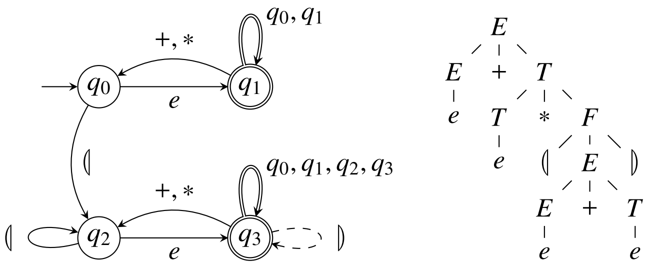
	$+$	$*$	$\llparenthesis$	$\rrparenthesis$	e
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$*$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$\llparenthesis$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
$\rrparenthesis$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
e	$\gtrdot$	$\gtrdot$		$\gtrdot$	

(a)



(b)

Figure 14: OPM of grammar GAE_P (a) and structure of the chains in the expression $\#e + e * \llparenthesis e + e \rrparenthesis \#$ (b).



input	state	stack
$e + e * \llparenthesis e + e\rrparenthesis\#$	q_0	Z_0
$+e * \llparenthesis e + e\rrparenthesis\#$	q_1	$[e,\ q_0]Z_0$
$+e * \llparenthesis e + e\rrparenthesis\#$	q_1	Z_0
$e * \llparenthesis e + e\rrparenthesis\#$	q_0	$[+,\ q_1]Z_0$
$*\llparenthesis e + e\rrparenthesis\#$	q_1	$[e,\ q_0][+,\ q_1]Z_0$
$*\llparenthesis e + e\rrparenthesis\#$	q_1	$[+,\ q_1]Z_0$
$\llparenthesis e + e\rrparenthesis\#$	q_0	$[*,\ q_1][+,\ q_1]Z_0$
$e + e\rrparenthesis\#$	q_2	$[\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$+e\rrparenthesis\#$	q_3	$[e,\ q_2][\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$+e\rrparenthesis\#$	q_3	$[\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$e\rrparenthesis\#$	q_2	$[+,\ q_3][\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$\rrparenthesis\#$	q_3	$[e,\ q_2][+,\ q_3][\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$\rrparenthesis\#$	q_3	$[+,\ q_3][\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$\rrparenthesis\#$	q_3	$[\llparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$\#$	q_3	$[\rrparenthesis,\ q_0][*,\ q_1][+,\ q_1]Z_0$
$\#$	q_3	$[*,\ q_1][+,\ q_1]Z_0$
$\#$	q_3	$[+,\ q_1]Z_0$
$\#$	q_3	Z_0

Figure 15: Automaton and example of computation for the language of Example 5.10. Recall that shift, push and pop transitions are denoted by dashed, normal and double arrows, respectively.

Notice the similarity of the above definition of OPA with that of VPA (Definition 4.4) and with the normalized form for PDA given in Section 4.3.2. This similarity, on the other hand, produces some remarkable difference between the sequence of moves of an OPA and the execution flow of Algorithm 1: whereas a shift move of the OPA does not change the length of the stack but only the contents of its top, Algorithm 1 pushes the read symbol onto the stack.

Showing the equivalence between OPG and OPA, though somewhat intuitive, requires to overcome a few non-trivial technical difficulties, mainly in the path from OPG to OPA. Here we offer just an informal description of the rationale of the two constructions and an illustrating example; the full proof of the equivalence between OPG and OPA can be found in [38].

For convenience and with no loss of generality, let G be an OPG with no empty rules, except possibly $S \to \varepsilon$, and no renaming rules, except possibly those whose lhs is S, an OPA $\mathcal{A}$ equivalent to G is built in such a way that a successful computation thereof corresponds to building bottom-up the mirror image of a rightmost derivation of G: $\mathcal{A}$ 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, if there are several rules with the same rhs) the nonterminal at the lhs.

Each state of $\mathcal{A}$ contains two pieces of information: the first component represents the prefix of the rhs under construction, whereas the second component is used to recover the rhs *previously under construction* (see Figure 16) whenever all rhs's nested below have been completed. Without going into the details of the construction and the formal equivalence proof between G and $\mathcal{A}$, we further illustrate the rationale of the construction through the following Example.

Example 5.11. *Consider again grammar GAE_P. Figure 17 shows the first part of an accepting computation of the automaton derived therefrom. Consider, for instance, step 3 of the computation: at this point the automaton has already reduced (nondeterministically) the first e to E and has pushed the following + onto the stack, paired with the state from which it*

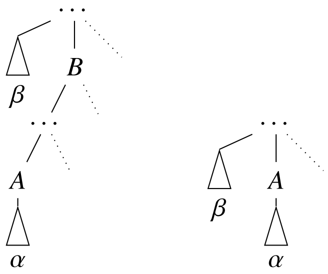


Figure 16: When parsing α, the prefix previously under construction is β.

was coming; thus, its new state is $\langle E+, \varepsilon\rangle$; at step 6, instead, the state is $\langle T*, E+\rangle$ because at this point the automaton has built the $T*$ part of the current rhs and remembers that the prefix of the suspended rhs is $E+$. The computation partially shown in Figure 17 is equal to that of Figure 15 up to a renaming of the states; the shape of syntax-trees and consequently the sequence of push, shift and pop moves in OPL depends only on the OPM, not on the visited states.

step	input	state	stack
0	$e + e * \llparenthesis e + e \rrparenthesis \#$	$\langle \varepsilon, \varepsilon \rangle$	Z_0
1	$+e * \llparenthesis e + e \rrparenthesis \#$	$\langle e, \varepsilon \rangle$	$[e,\ \langle \varepsilon, \varepsilon \rangle] Z_0$
2	$+e * \llparenthesis e + e \rrparenthesis \#$	$\langle E, \varepsilon \rangle$	Z_0
3	$e * \llparenthesis e + e \rrparenthesis \#$	$\langle E+, \varepsilon \rangle$	$[+,\ \langle E, \varepsilon \rangle] Z_0$
4	$*\llparenthesis e + e \rrparenthesis \#$	$\langle e, \varepsilon \rangle$	$[e,\ \langle E+, \varepsilon \rangle][+,\ \langle E, \varepsilon \rangle] Z_0$
5	$*\llparenthesis e + e \rrparenthesis \#$	$\langle T, E+ \rangle$	$[+,\ \langle E, \varepsilon \rangle] Z_0$
6	$\llparenthesis e + e \rrparenthesis \#$	$\langle T*, E+ \rangle$	$[*,\ \langle T, E+ \rangle][+,\ \langle E, \varepsilon \rangle] Z_0$

Figure 17: Partial accepting computation of the automaton built from grammar GAE_P.

The converse construction from OPA to OPG is somewhat simpler; in essence, from a given OPA $\mathcal{A} = (\Sigma, M, Q, I, F, \delta)$ a grammar G is built whose nonterminals are 4-tuples $(a, q, p, b) \in \Sigma \times Q \times Q \times \Sigma$, written as $\langle {}^a p, q^b \rangle$. For simplicity we assume that the OPM has no circularities in the $\doteq$ relation: in this way there is an upper bound to the length of P's rhs; in the (seldom [19]) case where this hypothesis is not verified we can resort to a generalized version of OPG which allows for rhs of grammar productions that include regular expressions [40] as it has been done in other extended forms of CFG. G's rules are built on the basis of $\mathcal{A}$'s chains as follows (particular cases are omitted for simplicity):

- for every simple chain ${}^{a_0}[a_1 a_2 \ldots a_n]^{a_{n+1}}$, if there is a sequence of $\mathcal{A}$'s transitions that, while reading the body of the chain starting from q_0 leaves $\mathcal{A}$ in q_{n+1}, include the rule
$$\langle {}^{a_0} q_0, {q_{n+1}}^{a_{n+1}} \rangle \longrightarrow a_1 a_2 \ldots a_n$$
- for every composed chain ${}^{a_0}[x_0 a_1 x_1 a_2 \ldots a_n x_n]^{a_{n+1}}$, add the rule
$$\langle {}^{a_0} q_0, {q_{n+1}}^{a_{n+1}} \rangle \longrightarrow \Lambda_0 a_1 \Lambda_1 a_2 \ldots a_n \Lambda_n$$
if there is a sequence of $\mathcal{A}$'s transitions that, while reading the body of the chain starting from q_0 leaves $\mathcal{A}$ in q_{n+1}, and, for every $i = 0, 1, \ldots, n$, $\Lambda_i = \varepsilon$ if $x_i = \varepsilon$, otherwise $\Lambda_i = \langle {}^{a_i} q_i, {q'_i}^{a_{i+1}} \rangle$ if $x_i \neq \varepsilon$ and there is a path leading from q_i to q'_i when traversing x_i.

Since the size of G's nonterminal alphabet is bounded and so is the number of possible productions thanks to the hypothesis of absence of circularities in M, the above procedure eventually terminates when no new rules are added to P.

We have seen that a fundamental issue to state the properties of most abstract machines is their determinizability: in the cases examined so far we have realized that the positive basic result holding for RL extends to the various versions of structured CFL, though at the expenses of more intricate constructions and size complexity of the deterministic versions obtained from the nondeterministic ones, but not to the general CF family. Given that OPL have been invented just with the motivation of supporting deterministic parsing, and given that they belong to the general family of structured languages, it is not surprising to find that for any nondeterministic OPA with s states an equivalent deterministic one can be built with $2^{O(s^2)}$ states, as it happens for the analogous construction for VPL: in [38] besides giving a detailed construction for the above result, it is also noticed that, thanks to the fact the construction of an OPA from an OPG in FNF produces an automaton already deterministic –since the grammar has no repeated rhs–, building a deterministic OPA from an OPG by first putting the OPG into FNF produces an automaton of an exponentially smaller size than the other way around.

5.1.2. Operator precedence vs other structured languages

A distinguishing feature of OPL is that, on the one hand they have been defined to support deterministic parsing, i.e., the construction of the syntax-tree of any sentence which is not immediately visible in the sentence itself but, on the other hand, they can still be considered as structured in the sense that their syntax-trees are univocally determined once an OPM is fixed, as it happens when we enclose grammar's rhs within parentheses or we split the terminal alphabet into $\Sigma_c \cup \Sigma_r \cup \Sigma_i$. It is therefore natural to compare their generative power with that of other structured languages.

In this respect, the main result of this section is that *OPL strictly include VPL*, which in turn strictly include parenthesis languages and the languages generated by balanced grammars (discussed in Section 4.3.1).

This result was originally proved in [31]. To give an intuitive explanation of this claim consider the following remarks:

[19] A "theoretical" example of OPL that cannot be generated by an OPG without $\doteq$-circularities is
$$L = \{a^n (bc)^n \mid n \geq 1\} \cup \{b^n (ca)^n \mid n \geq 1\} \cup \{c^n (ab)^n \mid n \geq 1\}.$$
However, we are not aware of OPL used in practical applications that require such OPM.

	Σ_c	Σ_r	Σ_i
Σ_c	$\lessdot$	$\doteq$	$\lessdot$
Σ_r	$\gtrdot$	$\gtrdot$	$\gtrdot$
Σ_i	$\gtrdot$	$\gtrdot$	$\gtrdot$

Legend
Σ_c denotes "calls"
Σ_r denotes "returns"
Σ_i denotes internal characters

Figure 18: A partitioned matrix, where Σ_c, Σ_r,Σ_i are set of terminal characters. A precedence relation in position Σ_α, Σ_β means that relation holds between all symbols of Σ_α and all those of Σ_β.

- Sequences $\in \Sigma_i^*$ can be assimilated to regular "sublanguages" with a linear structure; if we conventionally assign to them a left-linear structure, this can be obtained through an OPM where every character, but those in Σ_c, takes precedence over all elements in Σ_i; by stating instead that all elements of Σ_c yield precedence to the elements in Σ_i we obtain that after every call the OPA pushes and pops all subsequent elements of Σ_i, as an FSA would do without using the stack.

- All elements of Σ_r produce a pop from the stack of the corresponding element of Σ_c, if any; thus we obtain the same effect by letting them take precedence over all other terminal characters.

- A VPA performs a push onto its stack when (and only when) it reads an element of Σ_c, whereas an OPA pushes the elements to which some other element yields precedence; thus, it is natural to state that whenever on top of the stack there is a call symbol, possibly after having visited a subtree whose result is stored as the state component in the top of the stack together with the terminal symbol, such a symbol yields precedence to the following call (roughly, open parentheses yield precedence to other open parentheses and closed parentheses take precedence over other closed parentheses).

- Once the whole subsequence embraced by two matching call and return is scanned and possibly reduced, the two terminals are faced, with the possible occurrence of an irrelevant nonterminal in between, and therefore the call must be equal in precedence to the return.

- Finally, the usual convention that # yields precedence to everything and everything takes precedence over # enables the managing of possible unmatched returns at the beginning of the sentence and unmatched calls at its end.

In summary, for every VPA $\mathcal{A}$ with a given partitioned alphabet Σ, an OPM such as the one displayed in Figure 18 and an OPA $\mathcal{A}'$ defined thereon can be built such that $L(\mathcal{A}') = L(\mathcal{A})$.

In [31] it is also shown the converse property, i.e., that whenever an OPM is such that the terminal alphabet can be partitioned into three disjoint sets Σ_c, Σ_r,Σ_i such that the OPM has the shape of Figure 18, any OPL defined on such an OPM is also a VPL. Strict inclusion of VPL within OPL follows from the fact that VPA –which are a subclass of DPDA– recognize VPL in real-time, whereas OPL include also languages that cannot be recognized by any DPDA (see Section 3.1.1); there are also real-time OPL[20] such as

$$L = \{b^n c^n \mid n \geq 1\} \cup \{f^n d^n \mid n \geq 1\} \cup \{e^n (fb)^n \mid n \geq 1\}$$

that are not VPL. In fact, strings of type $b^n c^n$ impose that b is a call and c a return; for similar reasons, f must be a call and d a return. Strings of type $e^n (fb)^n$ impose that at least one of b and f must be a return, a contradiction for a VP alphabet. In conclusion we have the following result:

Theorem 5.12. *VPL are the subfamily of OPL whose OPM is a* partitioned matrix*, i.e., a matrix whose structure is depicted in Figure 18.*

As a corollary OPL also strictly include balanced languages and parenthesis languages. OPL are instead uncomparable with HRDPDL: we have already seen that the language $L_1 = \{a^n b a^n\}$ is an HRDPDL but it is neither a VPL nor an OPL since it necessarily requires a conflict $a \lessdot a$ and $a \gtrdot a$; conversely, the previous language $L_2 = \{a^m b^n c^n d^m \mid m, n \geq 1\} \cup \{a^m b^+ e d^m \mid m \geq 1\}$ can be recognized by an OPA but by no HRDPDA (see Section 3.1.1).

The increased power of OPL over other structured languages goes far beyond the mathematical containment properties and opens several application scenarios that are hardly accessible by "traditional" structured languages. The field of programming languages was the original motivation and source of inspiration for the introduction of OPL; arithmetic expressions, used throughout this paper as running examples, are just a small but meaningful subset of such languages and we have emphasized from the beginning that their partially hidden structure cannot be "forced" to the linearity of RL, nor can always be made explicit by the insertion of parentheses.

VPL too have been presented as an extension of parenthesis languages with the motivation that not always calls, e.g. procedure calls, can be matched by corresponding returns: a sudden closure, e.g. due to an exception or an interrupt or an unexpected end may leave an open chain of suspended calls. Such a situation, however, may need a generalization that cannot be formalized by the VPL formalism, since in VPL unmatched calls can occur only at the end of a string.[21] Imagine, for instance, that the occurrence of an interrupt while serving a chain of calls imposes to abort the chain to serve immediately the interrupt; after serving the interrupt, however, the normal system behavior may be resumed with new calls and corresponding returns even if some previous calls have been lost due to the need to serve the interrupt with high priority. Various, more or less

[20] When we say that an OPL L is real-time we mean, as usual, that there is an abstract machine, in particular a DPDA, recognizing it that performs exactly $|x|$ moves for every input string x; this is not to say that an OPA accepting L operates in real-time, since OPA's pop moves are defined as ε moves. A real-time subclass of OPA could be defined but has not yet been done so far.

[21] Recently, such a weakness of VPL has been acknowledged in [41] where the authors introduced *colored VPL* to cope with the above problem; the extended family, however, still does not reach the power of OPL ([41]).

sophisticated, policies can be designed to manage such systems and can be adequately formalized as OPL. The next example describes a first simple case of this type; other more sophisticated examples of the same type and further ones inspired by different application fields can be found in [38].

Example 5.13 (Managing interrupts). *Consider a software system that is designed to serve requests issued by different users but subject to interrupts. Precisely, assume that the system manages "normal operations" according to a traditional LIFO policy, and may receive and serve some interrupts denoted as int.*

We model its behavior by introducing an alphabet with two pairs of calls and returns: call and ret denote the call to, and return from, a normal procedure; int, and serve denote the occurrence of an interrupt and its serving, respectively. The occurrence of an interrupt provokes discarding possible pending calls not already matched by corresponding rets; furthermore when an interrupt is pending, i.e., not yet served, calls to normal procedures are not accepted and consequently corresponding returns cannot occur; interrupts however, can accumulate and are served themselves along a LIFO policy. Once all pending interrupts have been served the system can accept new calls and manage them normally.

Figure 19 (a) shows an OPM that assigns to sequences on the above alphabet a structure compatible with the described priorities. Then, a suitable OPA can specify further constraints on such sequences; for instance the automaton of Figure 19 (b) restricts the set of sequences compatible with the matrix by imposing that all int are eventually served and the computation ends with no pending calls; furthermore unmatched serve *and ret are forbidden. E.g., the string call.call.ret.int.serve.call.ret is accepted through the sequence of states* $q_0 \xrightarrow{call} q_1 \xrightarrow{call} q_1 \overset{ret}{\dashrightarrow} q_1 \overset{q_1}{\Longrightarrow} q_1 \overset{q_0}{\Longrightarrow} q_0 \xrightarrow{int} q_2 \overset{serve}{\dashrightarrow} q_2 \overset{q_0}{\Longrightarrow} q_0 \xrightarrow{call} q_1 \overset{ret}{\dashrightarrow} q_1 \overset{q_0}{\Longrightarrow} q_0$*; on the contrary, a sequence beginning with call.serve would not be accepted.*

5.1.3. Closure and decidability properties

Structured languages with *compatible structures* often enjoy many closure properties typical of RL; noticeably, families of structured languages are often Boolean algebras. The intuitive notion of compatible structure is formally defined for each family of structured languages; for instance two VPL have compatible structure iff their tri-partition of Σ is the same; two height-deterministic PDA languages (HPDL) have compatible structure if they are synchronized. In the case of OPL, the notion of structural compatibility is naturally formalized through the OPM.

Definition 5.14. *Given two OPM M_1 and M_2, we define set inclusion and union:*

$$M_1 \subseteq M_2 \text{ if } \forall a, b : (M_1)_{ab} \subseteq (M_2)_{ab}$$

$$M = M_1 \cup M_2 \text{ if } \forall a, b : M_{ab} = (M_1)_{ab} \cup (M_2)_{ab}.$$

Two matrices are compatible *if their union is conflict-free. A matrix is* total *(or* complete*) if it contains no empty cell.*

The following theorem has been proved originally in [14] by exploiting some standard forms of OPG that have been applied to grammar inference problems [15].

Theorem 5.15. *For any conflict-free OPM M the OPL whose OPM is contained in M are a Boolean algebra. In particular, if M is complete, the top language of its associated algebra is Σ^* with the structure defined by M.*

Notice however, that the same result could be proved in a simpler and fairly standard way by exploiting OPA and their traditional composition rules (which pass through determinization to achieve closure under complement). As usual in such cases, thanks to the decidablity of the emptiness problem for general CFL, a major consequence of Boolean closures is the following corollary.

Corollary 5.16. *The inclusion problem between OPL with compatible OPM is decidable.*

Closure under concatenation and Kleene * has been proved more recently in [31]; whereas such closures are easily proved or disproved for many families of languages, the technicalities to achieve this result are not trivial for OPL; however we do not go into their description since closure or non-closure w.r.t. these operations is not of major interest in this paper.

5.1.4. Logic characterization

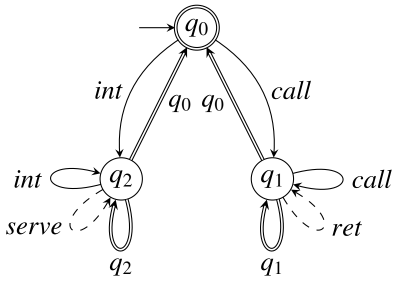
Achieving a logic characterization of OPL has probably been the most difficult job in the recent revisit of these languages and posed new challenges w.r.t. the analogous path previously followed for RL and then for CFL [28] and VPL [8]. In fact we have seen that moving from linear languages such as RL to tree-shaped ones such as CFL led to the introduction of the relation M between the positions of leftmost and rightmost leaves of any subtree (generated by a grammar in DGNF); the obtained characterization in terms of first-order formulas existentially quantified w.r.t. the M relation (which is a representation of the sentence structure) however, was suffering from the lack of closure under complementation of CFL [28]; the same relation instead proved more effective for VPL thanks to the fact that they are structured and enjoy all necessary closures.

To the best of our knowledge, however, *all previous characterizations* of formal languages in terms of logics that refer to string positions (for instance, there is significant literature on the characterization of various subclasses of RL in terms of first-order or temporal logics, see, e.g., [2]) *have been given for languages whose recognizing automata operate in real-time.* This feature is the key that allows, in the exploitation of MSO logic, to state a natural correspondence between automaton's state q_i and second-order variable $\boldsymbol{X}_i$ in such a way that the value of $\boldsymbol{X}_i$ is the set of positions where the state visited by the automaton is q_i.

OPL instead include also DCFL that cannot be recognized by any DPDA in real-time and, as a consequence, there are positions where the recognizing OPA traverses different configurations with different states. As a further consequence, the M

	call	*ret*	*int*	*serve*
call	⋖	≐	⋗	⋗
ret	⋗	⋗	⋗	⋗
int			⋖	≐
serve	⋗	⋗	⋗	⋗

(a)



(b)

Figure 19: OPM (a) and automaton (b) for the language of Example 5.13.

relation adopted for CFL and VPL is not anymore a one-to-one relation since the same position may be the position of the left/rightmost leaf of several subtrees of the whole syntax-tree; this makes formulas such as the key ones given in Section 4.2.1 meaningless.

The following key ideas helped overtaking the above difficulties:

- A new relation μ replaces the original M adopted in [28] and [8]; μ is based on the look-ahead-look-back mechanism which drives the (generalized) input-driven parsing of OPL based on precedence relations: thus, whereas in $M(\boldsymbol{x},\boldsymbol{y})$ $\boldsymbol{x},\boldsymbol{y}$ denote the positions of the extreme leaves of a subtree, in $\mu(\boldsymbol{x},\boldsymbol{y})$ they denote the position of the *context* of the same subtree, i.e., respectively, of the character that yields precedence to the subtree's leftmost leaf, and of the one over which the subtree's rightmost leaf takes precedence.

  Formally, $\mu(\boldsymbol{x},\boldsymbol{y})$ holds in a string $\#w\#$ iff $\#w\# = w_1 a w_2 b w_3, |w_1| = \boldsymbol{x}, |w_1 a w_2| = \boldsymbol{y}$, and ${}^a[w_2]^b$ is a chain. The new μ relation is not one-to-one as well, but, unlike the original M, its parameters $\boldsymbol{x},\boldsymbol{y}$ are not "consumed" by a pop transition of the automaton and remain available to be used in further automaton transitions of any type. In other words, μ holds between the positions 0 and $n+1$ of every chain, where, by convention, 0 is the position of the first # and $n+1$ is that of the last one (see Definition 5.6).

  For instance, Figure 20 displays the μ relation, graphically denoted by arrows, holding for the sentence $e + e * (\!| e + e |\!)$ generated by grammar GAE_P: we have $\mu(0,2)$, $\mu(2,4), \mu(5,7), \mu(7,9), \mu(5,9), \mu(4,10), \mu(2,10)$, and $\mu(0, 10)$. Such pairs correspond to contexts where a reduce operation is executed during the parsing of the string (they are listed according to their execution order).

  In general $\mu(\boldsymbol{x},\boldsymbol{y})$ implies $\boldsymbol{y} > \boldsymbol{x} + 1$, and a position $\boldsymbol{x}$ may be in relation μ with more than one position and vice versa. Moreover, if w is compatible with M, then $\mu(0, |w| + 1)$.

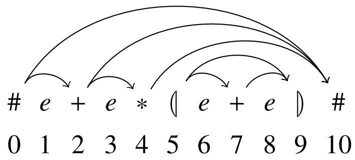


Figure 20: The string $e + e * (\!|e + e|\!)$, with positions and relation μ.

Example 5.17. *The following sentence of the MSO logic enriched with the μ relation defines, within the universe of strings compatible with the OPM of Figure 14(a), the language where parentheses are used only when they are needed (i.e. to give precedence to + over ∗).*

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \left(\begin{array}{c} \mu(\boldsymbol{x},\boldsymbol{y}) \land (\!|(\boldsymbol{x}+1) \land |\!)(\boldsymbol{y}-1) \\ \Rightarrow \\ (*(\boldsymbol{x}) \lor *(\boldsymbol{y})) \land \\ \exists \boldsymbol{z} \left(\begin{array}{c} \boldsymbol{x}+1<\boldsymbol{z}<\boldsymbol{y}-1 \land +(\boldsymbol{z}) \land \\ \neg\exists \boldsymbol{u} \exists \boldsymbol{v} \left(\begin{array}{c} \boldsymbol{x}+1<\boldsymbol{u}<\boldsymbol{z} \land (\!|(\boldsymbol{u}) \land \\ \boldsymbol{z}<\boldsymbol{v}<\boldsymbol{y}-1 \land |\!)(\boldsymbol{v}) \land \\ \mu(\boldsymbol{u}-1,\boldsymbol{v}+1)\end{array}\right)\end{array}\right)\end{array}\right)$$

- Since in every position there may be several states held by the automaton while visiting that position, instead of associating just one second-order variable to each state of the automaton we define three different sets of second-order variables, namely, $\boldsymbol{A}_0, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_N$, $\boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_N$ and $\boldsymbol{C}_0, \boldsymbol{C}_1, \ldots, \boldsymbol{C}_N$. Set $\boldsymbol{A}_i$ contains those positions of word w where state q_i may be assumed after a shift or push transition, i.e. after a transition that "consumes" an input symbol. Sets $\boldsymbol{B}_i$ and $\boldsymbol{C}_i$ encode a pop transition concluding the reading of the body of a chain ${}^a[w_0 a_1 w_1 \ldots a_l w_l]^{a_{l+1}}$ in a state q_i: set $\boldsymbol{B}_i$ contains the position of symbol a that precedes the corresponding push, whereas $\boldsymbol{C}_i$ contains the position of a_l, which is the symbol on top of the stack when the automaton performs the pop move relative to the whole chain.

  Figure 21 presents such sets for the example automaton of Figure 15, with the same input as in Figure 20. Notice that each position, except the last one, belongs to exactly one $\boldsymbol{A}_i$, whereas it may belong to several $\boldsymbol{B}_i$ and at most one $\boldsymbol{C}_i$.

We now outline how an OPA can be derived from an MSO logic formula making use of the new symbol μ and conversely.

From MSO formula to OPA

The construction from MSO logic to OPA essentially follows the lines given originally by Büchi, and reported in Section 2.1: once the original alphabet has been enriched and the formula has been put in the canonical form in the same way as described in Section 2.1, we only need to define a suitable automaton fragment to be associated with the new atomic formula $\mu(\boldsymbol{X}_i, \boldsymbol{X}_j)$; then, the construction of the global automaton corresponding to the global formula proceeds in the usual inductive way.

Figure 22 represents the OPA for atomic formula $\psi = \mu(\boldsymbol{X}, \boldsymbol{Y})$. As before, labels are triples belonging to $\Sigma \times \{0, 1\}^2$, where the first component encodes a character $a \in \Sigma$, the second the positions belonging to $\boldsymbol{X}$ (with 1) or not (with 0), while the third component is for $\boldsymbol{Y}$. The symbol $\circ$ is used as a shortcut for any value in Σ compatible with the OPM, so that the resulting automaton is deterministic.

The semantics of μ requires for $\mu(\boldsymbol{X}, \boldsymbol{Y})$ that there must be a chain ${}^a[w_2]^b$ in the input word, where a is the symbol at the only position in $\boldsymbol{X}$, and b is the symbol at the only position in $\boldsymbol{Y}$. By definition of chain, this means that a must be read, hence in the position represented by $\boldsymbol{X}$ the automaton performs either a push or a shift move (see Figure 22, from state q_0 to q_1), as pop moves do not consume input. After that, the automaton must read w_2. In order to process the chain ${}^a[w_2]^b$, reading w_2 must start with a push move (from state q_1 to state q_2), and it must end with one or more pop moves, before reading b (i.e. the only position in $\boldsymbol{Y}$ – going from state q_3 to q_F).

This means that the automaton, after a generic sequence of moves corresponding to visiting an irrelevant (for $\mu(\boldsymbol{X}, \boldsymbol{Y})$) portion of the syntax-tree, when reading the symbol at position $\boldsymbol{X}$ performs either a push or a shift move, depending on whether $\boldsymbol{X}$ is the position of a leftmost leaf of the tree or not. Then it visits the subsequent subtree ending with a pop labeled q_1; at this point, if it reads the symbol at position $\boldsymbol{Y}$, it accepts anything else that follows the examined fragment.

It is interesting to compare the diagram of Figure 22 with those of Figure 1 (c) and of Figure 9: the first one, referring to RL, uses two consecutive moves; the second one, referring to VPL, may perform an unbounded number of internal moves and of matching call-return pairs between the call-return pair in positions $\boldsymbol{x}$, $\boldsymbol{y}$; the OPA does the same as the VPA but needs a pair of extra moves to take into account the look-ahead-look-back implied by precedence relations.

From the OPA $\mathcal{A}$ to the MSO formula

In this case the overall structure of the logic formula φ satisfied by the sentences accpepted by a given OPA is the same as in the previous cases for RL and VPL, and is given below:

$$\varphi := \exists \boldsymbol{e} \begin{array}{l} \exists \boldsymbol{A}_0, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_N \\ \exists \boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_N \\ \exists \boldsymbol{C}_0, \boldsymbol{C}_1, \ldots, \boldsymbol{C}_N \end{array} \left(\mathrm{Start}_0 \wedge \varphi_\delta \wedge \bigvee_{q_f \in F} \mathrm{End}_f \right), \quad (1)$$

where the first and last subformulae encode the initial and final states of the run, respectively; formula φ_δ is defined as $\varphi_{\delta_{\text{push}}} \wedge \varphi_{\delta_{\text{shift}}} \wedge \varphi_{\delta_{\text{pop}}}$ and encodes the three transition functions of the automaton, which are expressed as the conjunction of *forward* and *backward* formulae. Variable $\boldsymbol{e}$ is used to refer to the *end* of a string.

The complete formalization of the δ transition relation as a collection of formulas relating the various variables $\boldsymbol{A}_i, \boldsymbol{B}_i, \boldsymbol{C}_i$, however, is much more involved than in the two previous cases. Here we only provide a few meaningful examples of such formulas, just to give the essential ideas of how they have been built; their complete set can be found in [38] together with the equivalence proof. Without loss of generality we *assume that the OPA is deterministic*.

Preliminarily, we introduce some notation to make the following formulas more understandable:

- When considering a chain ${}^a[w]^b$ we assume $w = w_0 a_1 w_1 \ldots a_\ell w_\ell$, with ${}^a[a_1 a_2 \ldots a_\ell]^b$ being a simple chain (any w_g may be empty). We denote by s_g the position of symbol a_g, for $g = 1, 2, \ldots, \ell$ and set $a_0 = a$, $s_0 = 0$, $a_{\ell+1} = b$, and $s_{\ell+1} = |w| + 1$.

- $\boldsymbol{x} \lessdot \boldsymbol{y}$ states that the symbol in position $\boldsymbol{x}$ yields precedence to the one in position $\boldsymbol{y}$ and similarly for the other precedence relations

- The fundamental abbreviation

  $$\mathrm{Tree}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y}) := \mu(\boldsymbol{x}, \boldsymbol{y}) \wedge \left(\begin{array}{c} (\boldsymbol{x} + 1 = \boldsymbol{z} \vee \mu(\boldsymbol{x}, \boldsymbol{z})) \wedge \\ \neg \exists \boldsymbol{t}(\boldsymbol{z} < \boldsymbol{t} < \boldsymbol{y} \wedge \mu(\boldsymbol{x}, \boldsymbol{t})) \wedge \\ (\boldsymbol{v} + 1 = \boldsymbol{y} \vee \mu(\boldsymbol{v}, \boldsymbol{y})) \wedge \\ \neg \exists \boldsymbol{t}(\boldsymbol{x} < \boldsymbol{t} < \boldsymbol{v} \wedge \mu(\boldsymbol{t}, \boldsymbol{y})) \end{array} \right)$$

  is satisfied, for every chain ${}^a[w]^b$ embraced within positions $\boldsymbol{x}$ and $\boldsymbol{y}$, by a (unique, maximal) $\boldsymbol{z}$ such that $\mu(\boldsymbol{x}, \boldsymbol{z})$, if $w_0 \neq \varepsilon$, $\boldsymbol{z} = \boldsymbol{x} + 1$ if instead $w_0 = \varepsilon$; symmetrically for $\boldsymbol{y}$ and $\boldsymbol{v}$. In particular, if w is the body of a simple chain, then $\mu(0, \ell + 1)$ and $\mathrm{Tree}(0, 1, \ell, \ell + 1)$ are satisfied; if it is the body of a composed chain, then $\mu(0, |w| + 1)$ and $\mathrm{Tree}(0, s_1, s_\ell, s_{\ell+1})$ are satisfied. If $w_0 = \varepsilon$ then $s_1 = 1$, and if $w_\ell = \varepsilon$ then $s_\ell = |w|$. In the example of Figure 20 relations $\mathrm{Tree}(2, 3, 3, 4)$, $\mathrm{Tree}(2, 4, 4, 10)$, $\mathrm{Tree}(4, 5, 9, 10)$, $\mathrm{Tree}(5, 7, 7, 9)$ are satisfied, among others.

- The shortcut $Q_i(\boldsymbol{x}, \boldsymbol{y})$ is used to represent that $\mathcal{A}$ is in state q_i when at position $\boldsymbol{x}$ and the next position to read, possibly after scanning a chain, is $\boldsymbol{y}$. Since the automaton is not real time, we must distinguish between the case of push and shift moves, which occur when the automaton reads the character immediately following the current one (case $\mathrm{Succ}_i(\boldsymbol{x}, \boldsymbol{y})$), and the case when the automaton

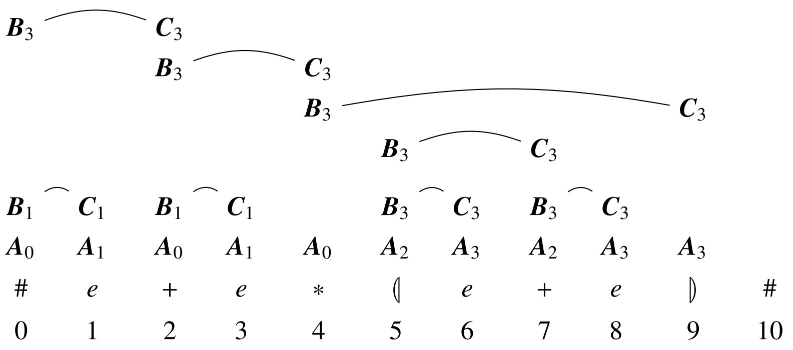

B_3 C_3
B_3 C_3
B_3 C_3
B_3 C_3
B_1 C_1 B_1 C_1 B_3 C_3 B_3 C_3
A_0 A_1 A_0 A_1 A_0 A_2 A_3 A_2 A_3 A_3
# e + e * ⦇ e + e ⦈ #
0 1 2 3 4 5 6 7 8 9 10

Figure 21: The string of Figure 20 with $\boldsymbol{B}_i$, $\boldsymbol{A}_i$, and $\boldsymbol{C}_i$ evidenced for the automaton of Figure 15. Pop moves of the automaton are represented by linked pairs $\boldsymbol{B}_i$, $\boldsymbol{C}_i$.

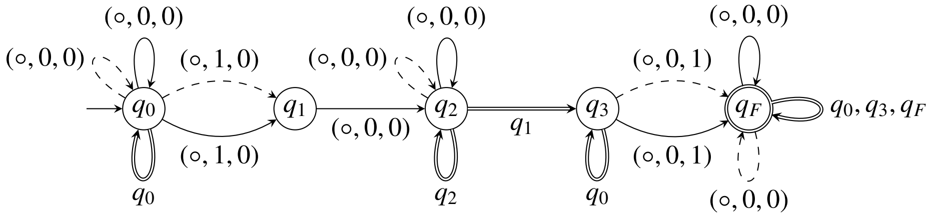


Figure 22: OPA for atomic formula $\mu(\boldsymbol{X}, \boldsymbol{Y})$.

visits a whole subtree –i.e, scans a chain– through a sequence of moves terminating with a pop transition in state q_i and the next character to be read is in position $\boldsymbol{y}$ (case $\text{Next}_i(\boldsymbol{x}, \boldsymbol{y})$).

$$\begin{aligned}
\text{Succ}_k(\boldsymbol{x}, \boldsymbol{y}) &:= \; \boldsymbol{x} + 1 = \boldsymbol{y} \wedge \boldsymbol{x} \in \boldsymbol{A}_k \\
\text{Next}_k(\boldsymbol{x}, \boldsymbol{y}) &:= \; \mu(\boldsymbol{x}, \boldsymbol{y}) \wedge \boldsymbol{x} \in \boldsymbol{B}_k \wedge \\
&\quad\;\; \exists \boldsymbol{z}, \boldsymbol{v} \, (\text{Tree}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y}) \wedge \boldsymbol{v} \in \boldsymbol{C}_k) \\
Q_i(\boldsymbol{x}, \boldsymbol{y}) &:= \; \text{Succ}_i(\boldsymbol{x}, \boldsymbol{y}) \vee \text{Next}_i(\boldsymbol{x}, \boldsymbol{y}).
\end{aligned}$$

E.g., with reference to Figures 20 and 21, $\text{Succ}_2(5, 6)$, $\text{Next}_3(5, 9)$, and $\text{Next}_3(5, 7)$ hold.

We can now show a meaningful sample of the various formulas that code the automaton's transition relation.

- The subformulas representing the initial and final states of the parsing of a string of length $\boldsymbol{e}$ are defined as follows.

  $$\text{Start}_i := 0 \in \boldsymbol{A}_i \wedge \neg \bigvee_{j \neq i} (0 \in \boldsymbol{A}_j)$$

  where i is the index of the *only* initial state, and

  $$\begin{aligned}\text{End}_f := \neg \exists \boldsymbol{y} (\boldsymbol{e} + 1 < \boldsymbol{y}) \wedge \; \text{Next}_f(0, \boldsymbol{e} + 1) \\ \wedge \neg \bigvee_{j \neq f} (\text{Next}_j(0, \boldsymbol{e} + 1)).\end{aligned}$$

  where f is the index of *any* final state.

- For each transition $\delta_{\text{push}}(q_i, c) = q_k$, the following formula states that if $\mathcal{A}$ is in position $\boldsymbol{x}$ and state q_i and reads the character in position $\boldsymbol{y}$, it goes to state q_k.

  $$\varphi_{push\_fw}(i, c, k) := \forall \boldsymbol{x}, \boldsymbol{y} \, (\boldsymbol{x} \lessdot \boldsymbol{y} \wedge c(\boldsymbol{y}) \wedge Q_i(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \boldsymbol{y} \in \boldsymbol{A}_k)$$

  For instance, with reference to the automaton of Figure 15 which recognizes arithmetic expressions with parentheses, we would build the formulas

  $$\forall \boldsymbol{x}, \boldsymbol{y} \, (\boldsymbol{x} \lessdot \boldsymbol{y} \wedge e(\boldsymbol{y}) \wedge Q_0(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \boldsymbol{y} \in \boldsymbol{A}_1)$$

  and

  $$\forall \boldsymbol{x}, \boldsymbol{y} \, (\boldsymbol{x} \lessdot \boldsymbol{y} \wedge e(\boldsymbol{y}) \wedge Q_2(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \boldsymbol{y} \in \boldsymbol{A}_3)$$

  and similar ones for other terminal characters. Notice that the original formula given in Section 2.1 for RL can be seen as a particular case of the above one.

- Conversely, if $\mathcal{A}$ is in state q_k after a push starting from position $\boldsymbol{x}$ and reading character c, in that position it must have been in a state q_i such that $\delta_{\text{push}}(q_i, c) = q_k$:

  $$\varphi_{push\_bw}(c, k) := \forall \boldsymbol{x}, \boldsymbol{y} \left(\begin{array}{c} \boldsymbol{x} \lessdot \boldsymbol{y} \;\wedge\; c(\boldsymbol{y}) \;\wedge\; \boldsymbol{y} \in \boldsymbol{A}_k \wedge \\ (\boldsymbol{x} + 1 = \boldsymbol{y} \;\vee\; \mu(\boldsymbol{x}, \boldsymbol{y})) \\ \Rightarrow \\ \bigvee_{i | \delta_{\text{push}}(q_i, c) = q_k} Q_i(\boldsymbol{x}, \boldsymbol{y}) \end{array} \right)$$

  which, in the case of the automaton of Figure 15 would produce, e.g., the formulas

  $$\forall \boldsymbol{x}, \boldsymbol{y} \left(\begin{array}{c} \boldsymbol{x} \lessdot \boldsymbol{y} \;\wedge\; ⦇(\boldsymbol{y}) \;\wedge\; \boldsymbol{y} \in \boldsymbol{A}_2 \wedge \\ (\boldsymbol{x} + 1 = \boldsymbol{y} \;\vee\; \mu(\boldsymbol{x}, \boldsymbol{y})) \end{array} \Rightarrow Q_0(\boldsymbol{x}, \boldsymbol{y}) \vee Q_1(\boldsymbol{x}, \boldsymbol{y}) \right)$$

  and

  $$\forall \boldsymbol{x}, \boldsymbol{y} \left(\begin{array}{c} \boldsymbol{x} \lessdot \boldsymbol{y} \;\wedge\; e(\boldsymbol{y}) \;\wedge\; \boldsymbol{y} \in \boldsymbol{A}_3 \wedge \\ (\boldsymbol{x} + 1 = \boldsymbol{y} \;\vee\; \mu(\boldsymbol{x}, \boldsymbol{y})) \end{array} \Rightarrow Q_2(\boldsymbol{x}, \boldsymbol{y}) \right)$$

  (since there is only one push transition labeled e leading to q_3) and all the similar ones referring to the other terminals and states.

- The formulas coding the shift transitions are similar to the previous ones and therefore omitted.

- To define $\varphi_{\delta_{\text{pop}}}$ we introduce the shortcut $\text{Tree}_{i,j}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y})$, which represents the fact that $\mathcal{A}$ is ready to perform a pop transition from state q_i having on top of the stack state q_j; such pop transition corresponds to the reduction of the portion of string between positions $\boldsymbol{x}$ and $\boldsymbol{y}$ (excluded).

  $$\text{Tree}_{i,j}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) := \text{Tree}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \wedge Q_i(\boldsymbol{v},\boldsymbol{y}) \wedge Q_j(\boldsymbol{x},\boldsymbol{z}).$$

  Formula $\varphi_{\delta_{\text{pop}}}$ is thus defined as the conjunction of three collections of formulas. As before, the forward (abbreviated with the subscript fw) formulas give the sufficient conditions for two positions to be in the sets $\boldsymbol{B}_k$ and $\boldsymbol{C}_k$, when performing a pop move, and the backward formulas state symmetric necessary conditions. For each transition $\delta_{\text{pop}}(q_i, q_j) = q_k$ we write

  $$\varphi_{pop\_fw}(i,j,k) := \forall \boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y} \begin{pmatrix} \text{Tree}_{i,j}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \Rightarrow \\ \boldsymbol{x} \in \boldsymbol{B}_k \wedge \boldsymbol{v} \in \boldsymbol{C}_k \end{pmatrix}$$

  For each state q_k:

  $$\varphi_{pop\_bwB}(k) := \forall \boldsymbol{x} \begin{pmatrix} \boldsymbol{x} \in \boldsymbol{B}_k \Rightarrow \\ \exists \boldsymbol{y},\boldsymbol{z},\boldsymbol{v} \bigvee_{i,j \mid \delta_{\text{pop}}(q_i,q_j)=q_k} \text{Tree}_{i,j}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \end{pmatrix}$$

  $$\varphi_{pop\_bwC}(k) := \forall \boldsymbol{v} \begin{pmatrix} \boldsymbol{v} \in \boldsymbol{C}_k \Rightarrow \\ \exists \boldsymbol{x},\boldsymbol{y},\boldsymbol{z} \bigvee_{i,j \mid \delta_{\text{pop}}(q_i,q_j)=q_k} \text{Tree}_{i,j}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \end{pmatrix}$$

  In the case of the automaton of Figure 15 we would have two φ_{pop_fw} formulas with $i = \{0, 1\}, j = 1, k = 1$ and 4 with $i = \{0, 1, 2, 3\}, j = 3, k = 3$; the following φ_{pop_bwB} formulas:

  $$\forall \boldsymbol{x} \begin{pmatrix} \boldsymbol{x} \in \boldsymbol{B}_1 \Rightarrow \\ \exists \boldsymbol{y},\boldsymbol{z},\boldsymbol{v}\, \text{Tree}_{0,1}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \vee \text{Tree}_{1,1}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \end{pmatrix}$$

  $$\forall \boldsymbol{x} \begin{pmatrix} \boldsymbol{x} \in \boldsymbol{B}_3 \Rightarrow \\ \exists \boldsymbol{y},\boldsymbol{z},\boldsymbol{v} \bigvee_{i=0\ldots3} \text{Tree}_{i,3}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y}) \end{pmatrix}$$

  and similarly for φ_{pop_bwC} formulas.

5.2. *Local parsability for parallel parsers*

Let us now go back to the original motivation that inspired Floyd when he invented the OPG family, namely supporting efficient, deterministic parsing. In the introductory part of this section we noticed that the mechanism of precedence relations isolates the grammar's rhs from their context so that they can be reduced to the corresponding lhs independently from each other. This fact guarantees that, in whichever order such reductions are applied, at the end a complete grammar derivation will be built; such a derivation corresponds to a visit of the syntax-tree, not necessarily leftmost or rightmost, and its *construction has no risk of applying any back-tracks* as it happens instead in nondeterministic parsing. We call this property *local parsability property*, which intuitively can be defined as the possibility of applying deterministically a bottom-up, shift-reduce parsing algorithm by inspecting only an a priori bounded portion of any string. Various formal definitions of this concept have been given in the literature, the first one probably being the one proposed by Floyd himself in [42]; a fairly general definition of local parsability and a proof that OPG enjoy it can be found in [43].

Local parsability, however, has the drawback that it loses chances of deterministic parsing when the information on how to proceed with the parsing is arbitrarily far from the current position of the parser, as we noticed in the case of language $L = \{0a^nb^n \mid n \geq 0\} \cup \{1a^nb^{2n} \mid n \geq 0\}$. We therefore have a trade-off between the size of the family of recognizable languages, which in the case of LR grammars is the whole DCFL class (see Section 3.1.1), and the constraint of proceeding rigorously left-to-right for the parser. So far this trade-off has been normally solved in favor of the generality in the absence of serious counterparts in favor of the other option. We argue, however, that the massive advent of parallel processing, even in the case of small architectures such as those of tablets and smartphones, could dramatically change the present state of affairs. On the one hand parallelizing parsers such as LL or LR ones requires reintroducing a kind of nondeterministic *guess* on the state of the parser in a given position, which in most cases voids the benefits of exploiting parallel processors (see [43] for an analysis of previous literature on various attempts to develop parallel parsers). On the other hand, OPL are from the beginning oriented toward parallel analysis whereas their previous use in compilation shows that they can be applied to a wide variety of practical languages, and further more as suggested by other examples given here and in [38].

Next we show how we exploited the local parsability property of OPG to realize a complete and general parallel parser for these grammars. A first consequence of the basic property is the following statement.

Statement 2. *For every substring $a\delta b$ of $\gamma a\delta b\eta \in V^*$ derivable from S, there exists a unique string α, called the* irreducible string, *deriving δ such that $S \overset{*}{\Rightarrow} \gamma a\alpha b\eta \overset{*}{\Rightarrow} \gamma a\delta b\eta$, and the precedence relations between the consecutive terminals of $a\alpha b$ do not contain the pattern $\lessdot (\doteq)^* \gtrdot$. Therefore there exists a factorization $a\alpha b = \zeta\theta$ into two possibly empty factors such that the left factor does not contain $\lessdot$ and the right factor does not contain $\gtrdot$.*

On the basis of the above statement, Algorithm 1 may receive as input a portion of a string, not necessarily enclosed within the delimiters #, and the resulting output stack $\mathcal{S}$ can be split in two parts, i.e. one that stores the substring ζ and does not contain $\lessdot$; one that stores θ and does not contain $\gtrdot$. We call such stacks $\mathcal{S}^L$ and $\mathcal{S}^R$, respectively. The complete description of a parallel OP parser based on splitting an input string on arbitrary fragments is reported in Algorithm 2.

We illustrate it through an example based on the grammar GAE_{FNF}, which is a FNF of GAE_1. If we supply to the parser the partial string $+e * e * e + e$, we obtain $\zeta = +T+$, $\theta = e$ and $+T+ \overset{*}{\Rightarrow} +e * e * e+$ since $+ \gtrdot +$ and $+ \lessdot e$.

At this point it is fairly easy to let several such generalized parsers work in parallel:

- Suppose to use k parallel processors, also called *workers*; then split the input into k chunks; given that an OP parser needs a look-ahead-look-back of one character, the

Algorithm 2 : Parallel-parsing(β, k)

1. Split the input string β into k substrings: $\#\beta_1\beta_2 \dots \beta_k\#$.
2. Launch k instances of Algorithm 1, where, for each $1 \le i \le k$, the parameters are $\mathcal{S} = (a_i, \perp)$, $\alpha = \beta_i b_i$, $head = |\beta_1\beta_2 \dots \beta_{i-1}| + 1$, $end = |\beta_1\beta_2 \dots \beta_i| + 1$; a_i is the last symbol of β_{i-1}, and b_i the first of β_{i+1}. Conventionally $\beta_0 = \beta_{k+1} = \#$. The result of this pass are $k' \le k$ stacks, where each $\mathcal{S}_i$ of them can be split in two parts: $\mathcal{S}^L_i$ and $\mathcal{S}^R_i$, the first not containing $\lessdot$, and the second not containing $\gtrdot$.
3. Repeat:
   - (a) For each adjacent non-empty stack pairs $(\mathcal{S}^L_i, \mathcal{S}^R_i)$ and $(\mathcal{S}^L_{i+1}, \mathcal{S}^R_{i+1})$, launch an instance of Algorithm 1, with:
     $\mathcal{S} = \mathcal{S}^R_i(a, \perp)$, where $a \in V$ is the top symbol in $\mathcal{S}^L_i$,
     $\alpha = X_2 \dots X_n$, where $\mathcal{S}^L_{i+1} = (X_n, p_n)(X_{n-1}, p_{n-1}) \dots (X_2, p_2)(X_1, p_1)$,
     $head = 1$, $end = |\alpha|$.
   - (b) Until either we have a single, un-splittable stack $\mathcal{S}'$ or the computation is aborted and some error recovery action is taken.
4. Return $\mathcal{S}'$.

chunks must overlap by one character for each consecutive pair. For instance, the global input string $\#e + e + e * e * e + *e + e\#$, with $k = 3$ could be split as shown below:

$$\# \overbrace{e + e}^{1} + \overbrace{e * e * e +}^{2} e \overbrace{* e + e}^{3} \#$$

where the unmarked symbols + and e are shared by the adjacent segments. The splitting can be applied arbitrarily, although in practice it seems natural to use segments of approximately equal length and/or to apply some heuristic criterion (for instance, if possible one should avoid particular cases where only $\lessdot$ or $\gtrdot$ relations occur in a single chunk so that the parser could not produce any reduction).

- Each chunk is submitted to one of the workers which produces a partial result in the form of the pair $(\mathcal{S}^L, \mathcal{S}^R)$ (notice that some of those partial stacks may be empty). In our example, the three chunks submitted to workers, and the resulting stacks are the following:

  1. Input 1: $\mathcal{S} = (\#, \perp), \alpha = e + e+$;
     output 1: $\mathcal{S} = (+, \lessdot)(E, \perp)(\#, \perp)$.
  2. Input 2: $\mathcal{S} = (+, \perp), \alpha = e * e * e + e$;
     output 2: $\mathcal{S} = (e, \gtrdot)(+, \lessdot)(T, \perp)(+, \perp)$.
  3. Input 3: $\mathcal{S} = (e, \perp), \alpha = *e + e\#$;
     output 3: $\mathcal{S} = (\#, \gtrdot)(F, \perp)(+, \gtrdot)(F, \perp)(*, \gtrdot)(e, \perp)$.

  Hence, the corresponding output stack pairs are:
  $\mathcal{S}^L_1 = \varepsilon$, $\mathcal{S}^R_1 = (+, \lessdot)(E, \perp)(\#, \perp)$;
  $\mathcal{S}^L_2 = (+, \lessdot)(T, \perp)(+, \perp)$, $\mathcal{S}^R_2 = (e, \gtrdot)$;
  $\mathcal{S}^L_3 = (\#, \gtrdot)(F, \perp)(+, \gtrdot)(F, \perp)(*, \gtrdot)(e, \perp)$, $\mathcal{S}^R_3 = \varepsilon$.

- The partial results are then combined in new input strings, containing also nonterminals, and starting stacks, and the process is iterated until a short enough single chunk is processed and the original input string is accepted or rejected. In our example, the number of workers becomes two, with the following inputs and outputs:

  1. Input 1: $\mathcal{S} = (+, \lessdot)(E, \perp)(\#, \perp), \alpha = T+$;
     output 1: $\mathcal{S} = (+, \lessdot)(E, \perp)(\#, \perp)$.
  2. Input 2: $\mathcal{S} = (e, \lessdot)(+, \perp), \alpha = *F + F\#$;
     output 2: $\mathcal{S} = (\#, \gtrdot)(F, \perp)(+, \gtrdot)(T, \perp)(+, \perp)$.

  In practice it may be convenient to build the new segments to be supplied to the workers by facing an $\mathcal{S}^R$ with the following $\mathcal{S}^L$ so that the likelihood of applying many new reductions in the next pass is increased. For instance the $\zeta = +T+$ part produced by the parsing of the second chunk could be paired with the $\mathcal{S}^R = \#E+$ part obtained from the parsing of the first chunk, producing the string $\#E + T+$ to be supplied to a worker for the new iteration. Some experience shows that quite often optimal results in terms of speed-up are obtained with 2, at most 3 passes of parallel parsing.

[43] describes in detail *PAPAGENO, a PArallel PArser GENeratOr*[22] built on the basis of the above algorithmic schema. It has been applied to several real-life data definition, or programming, languages including JSON, XML, JavaScript, and Lua and different HW architectures. The paper also reports on the experimental results in terms of the obtained speed-up compared with standard sequential parser generators as Bison. Being able to parse documents in parallel is getting more and more important nowadays, e.g. for Big Data processing, where often documents are in XML or JSON. Another important aspect is power efficiency: think e.g. about a Web rendering engine on a smartphone, which must process a complex HTML5 page with plenty of JavaScript code in it.

6. Concluding remarks

The main goal of this paper is to show that an old-fashioned and almost abandoned family of formal languages indeed offers

[22] PAPAGENO is freely available at `https://github.com/PAPAGENO-devels/papageno` under GNU license.

considerable new benefits in apparently unrelated application fields of high interest in modern applications, i.e., automatic property verification and parallelization. In the first field OPL significantly extend the generative power of the successful class of VPL still maintaining all of their properties: to the best of our knowledge, OPL are the largest class of languages closed under all major language operations and provided with a complete classification in terms of MSO logic.

Various other results about this class of languages have been obtained or are under development, which have not been included in this paper for length limits. We mention here just the most relevant or promising ones with appropriate references for further reading.

- The theory of OPL for languages of finite length strings has been extended in [38] to ω-languages, i.e. languages of infinite length strings: the obtained results perfectly parallel those originally obtained by Büchi and others for RL and subsequently extended to other families, noticeably VPL [8]; in particular, ω-OPL lose determinizability in case of Büchi acceptance criterion as it happens for RL and VPL.

- Some investigation is going on to devise more tractable automatic verification algorithms than those allowed by the full characterization of these languages in terms of MSO logic. On this respect, the state of the art is admittedly still far from the success obtained with model checking exploiting various forms of temporal logics for FSA and several extensions thereof such as, e.g., timed automata [44]. Some interesting preliminary results have been obtained for VPL by [12] and for a subclass of OPL in [45].

- The local parsability property can be exploited not only to build parallel parsers but also to make them *incremental*, in such a way that when a large piece of text or software code is locally modified its analysis should not be redone from scratch but only the affected part of the syntax-tree is "plugged" in the original one with considerable saving; furthermore incremental and/or parallel parsing can be naturally paired with incremental and/or parallel semantic processing, e.g. realized through the classic schema of *attribute evaluation* [46, 19]. Some early results on incremental software verification by exploiting the locality property are reported in [47]. We also mention ongoing work on parallel XML-based query processing.

- A seminal paper by Schützemberger [48] introduced the concept of *weighted languages* as RL where each word is given a weight in a given algebra which may represent some "attribute" of the word such as importance or probability. Later, these weighted languages too have been characterized in terms of MSO logic [49] and such a characterization has also been extended to VPL [50] and ω-VPL [51]. Recently, our two research groups started a joint research on *weighted OPL* which confirmed that the concepts of weighted languages too can be extended to OPL as it happened for all other properties investigated in this paper [52].

Acknowledgments. We acknowledge the contribution to our research given by Alessandro Barenghi, Stefano Crespi Reghizzi, Violetta Lonati, Angelo Morzenti, and Federica Panella. We also thank the anonymous reviewers for their careful reading and valuable suggestions.

References

[1] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (1986) 244–263.

[2] E. A. Emerson, Temporal and modal logic, in: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), 1990, pp. 995–1072.

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

[4] J. Thatcher, Characterizing derivation trees of context-free grammars through a generalization of finite automata theory, Journ. of Comp. and Syst.Sc. 1 (1967) 317–322.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, Available on: `http://www.grappa.univ-lille3.fr/tata`, release October, 12th 2007 (2007).

[6] K. Mehlhorn, Pebbling mountain ranges and its application of DCFL-recognition, in: Automata, languages and programming (ICALP-80), Vol. 85 of LNCS, 1980, pp. 422–435.

[7] B. von Braunmühl, R. Verbeek, Input-driven languages are recognized in log n space, in: Proceedings of the Symposium on Fundamentals of Computation Theory, Lect. Notes Comput. Sci. 158, Springer, 1983, pp. 40–51.

[8] R. Alur, P. Madhusudan, Adding nesting structure to words, J. ACM 56 (3).

[9] J. R. Büchi, Weak Second-Order Arithmetic and Finite Automata, Mathematical Logic Quarterly 6 (1-6) (1960) 66–92.

[10] C. C. Elgot, Decision problems of finite automata design and related arithmetics, Trans. Am. Math. Soc. 98 (1) (1961) 21–52.

[11] B. A. Trakhtenbrot, Finite automata and logic of monadic predicates (in Russian), Doklady Akademii Nauk SSR 140 (1961) 326–329.

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

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

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

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

[16] D. E. Knuth, On the translation of languages from left to rigth, Information and Control 8 (6) (1965) 607–639.

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

[18] D. Grune, C. J. Jacobs, Parsing techniques: a practical guide, Springer, New York, 2008.

[19] S. Crespi Reghizzi, L. Breveglieri, A. Morzenti, Formal Languages and Compilation, Second Edition, Texts in Computer Science, Springer, 2013.

[20] D. Mandrioli, M. Pradella, Generalizing input-driven languages: theoretical and practical benefits, CoRR abs/1705.00984.
URL `http://arxiv.org/abs/1705.00984`

[21] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[22] W. Thomas, Handbook of theoretical computer science (vol. B), MIT Press, Cambridge, MA, USA, 1990, Ch. Automata on infinite objects, pp. 133–191.

[23] M. Frick, M. Grohe, The complexity of first-order and monadic second-order logic revisited, Ann. Pure Appl. Logic 130 (1-3) (2004) 3–31. `doi:10.1016/j.apal.2004.01.007`.
URL `https://doi.org/10.1016/j.apal.2004.01.007`

[24] J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, A. Sandholm, Mona: Monadic second-order logic in practice, in: Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019, 1995.

[25] S. A. Greibach, A new normal-form theorem for context-free phrase structure grammars, J. ACM 12 (1) (1965) 42–52.

[26] J. Hartmanis, J. E. Hopcroft, What makes some language theory problems undecidable, J. Comput. Syst. Sci. 4 (4) (1970) 368–376.

[27] D. Nowotka, J. Srba, Height-Deterministic Pushdown Automata, in: L. Kucera, A. Kucera (Eds.), MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings, Vol. 4708 of LNCS, Springer, 2007, pp. 125–134.

[28] C. Lautemann, T. Schwentick, D. Thérien, Logics for context-free languages, in: L. Pacholski, J. Tiuryn (Eds.), Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers, Vol. 933 of Lecture Notes in Computer Science, Springer, 1994, pp. 205–216.

[29] R. Alur, P. Madhusudan, Visibly Pushdown Languages, in: STOC: ACM Symposium on Theory of Computing (STOC), 2004.

[30] J. Berstel, L. Boasson, Balanced Grammars and Their Languages, in: W. B. et al. (Ed.), Formal and Natural Computing, Vol. 2300 of LNCS, Springer, 2002, pp. 3–25.

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

[32] D. Fisman, A. Pnueli, Beyond Regular Model Checking, FSTTCS: Foundations of Software Technology and Theoretical Computer Science 21.

[33] D. Caucal, Synchronization of Pushdown Automata, in: O. H. Ibarra, Z. Dang (Eds.), Developments in Language Theory, Vol. 4036 of LNCS, Springer, 2006, pp. 120–132.

[34] A. Potthoff, W. Thomas, Regular tree languages without unary symbols are star-free, in: Z. Ésik (Ed.), Fundamentals of Computation Theory, 9th International Symposium, FCT '93, Szeged, Hungary, August 23-27, 1993, Proceedings, Vol. 710 of Lecture Notes in Computer Science, Springer, 1993, pp. 396–405.

[35] D. Caucal, On infinite transition graphs having a decidable monadic theory, Theor. Comput. Sci. 290 (1) (2003) 79–115.

[36] M. J. Fischer, Some properties of precedence languages, in: STOC '69: Proc. first annual ACM Symp. on Theory of Computing, ACM, New York, NY, USA, 1969, pp. 181–190.

[37] K. De Bosschere, An Operator Precedence Parser for Standard Prolog Text, Softw., Pract. Exper. 26 (7) (1996) 763–779.

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

[39] V. Lonati, D. Mandrioli, M. Pradella, Precedence Automata and Languages, in: 6th Int. Computer Science Symposium in Russia (CSR), Vol. 6651 of LNCS, 2011, pp. 291–304.

[40] S. Crespi Reghizzi, M. Pradella, Higher-order operator precedence languages, in: E. Csuhaj-Varjú, P. Dömösi, G. Vaszil (Eds.), Proceedings 15th International Conference on Automata and Formal Languages, AFL 2017, Debrecen, Hungary, September 4-6, 2017., Vol. 252 of EPTCS, 2017, pp. 86–100. doi:10.4204/EPTCS.252.11.
URL https://doi.org/10.4204/EPTCS.252.11

[41] R. Alur, D. Fisman, Colored nested words, in: Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings, 2016, pp. 143–155. doi:10.1007/978-3-319-30000-9_11.
URL http://dx.doi.org/10.1007/978-3-319-30000-9_11

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

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

[44] R. Alur, D. L. Dill, A Theory of Timed Automata, Theor. Comput. Sci. 126 (2) (1994) 183–235.

[45] V. Lonati, D. Mandrioli, F. Panella, M. Pradella, First-order logic definability of free languages, in: L. D. Beklemishev, D. V. Musatov (Eds.), Computer Science - Theory and Applications - 10th International Computer Science Symposium in Russia, CSR 2015, Listvyanka, Russia, July 13-17, 2015, Proceedings, Vol. 9139 of Lecture Notes in Computer Science, Springer, 2015, pp. 310–324.

[46] D. E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145.

[47] D. Bianculli, A. Filieri, C. Ghezzi, D. Mandrioli, Syntactic-semantic incrementality for agile verification, Sci. Comput. Program. 97 (2015) 47–54.

[48] M. P. Schützenberger, On the definition of a family of automata, Information and Control 4 (2-3) (1961) 245–270.

[49] M. Droste, P. Gastin, Weighted automata and weighted logics, Theor. Comput. Sci. 380 (1-2) (2007) 69–86.

[50] M. Droste, B. Pibaljommee, Weighted nested word automata and logics over strong bimonoids, Int. J. Found. Comput. Sci. 25 (5) (2014) 641.

[51] M. Droste, S. Dück, Weighted automata and logics for infinite nested words, Inf. Comput. 253 (2017) 448–466. doi:10.1016/j.ic.2016.06.010.
URL https://doi.org/10.1016/j.ic.2016.06.010

[52] M. Droste, S. Dück, D. Mandrioli, M. Pradella, Weighted operator precedence languages, in: K. G. Larsen, H. L. Bodlaender, J. Raskin (Eds.), 42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark, Vol. 83 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 31:1–31:15. doi:10.4230/LIPIcs.MFCS.2017.31.
URL https://doi.org/10.4230/LIPIcs.MFCS.2017.31