POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

Operator Precedence Languages: Theory and Applications

Doctoral Dissertation of:
Federica Panella

Advisor:
Prof. Matteo Pradella

Tutor:
Prof. Luciano Baresi

Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2015 – XXVII

ABSTRACT

Operator Precedence Languages (OPLs) were introduced in the 1960s by Robert Floyd to support deterministic and efficient parsing of context-free languages. Recently, interest in this class of languages has been renewed thanks to a few distinguishing properties that make them attractive for exploiting various modern technologies in two main contexts: automatic software verification techniques, as model checking, and parallel and incremental parsing of programming and data-description languages.

This thesis provides a complete theory of OPLs and investigates the properties that allow for their application in these different fields.

Along a first line of research, we complement the results on this class of languages that have been proved in the last half a century, which characterized them in terms of equivalent classes of grammars, recognizing automata and a Monadic second-order logic; the study of their algebraic properties, furthermore, has qualified them as the largest class of deterministic context-free languages enjoying closure under all main language operations (Boolean ones, concatenation, Kleene * and others), strictly including renowned families of formalisms as parentheses languages and Visibly Pushdown Languages (VPLs). In this dissertation we extend research on OPLs to the field of ω-languages, i.e., languages consisting of strings of infinite length, which can model the behavior of systems with never-ending computations (such as operating systems, control systems, web services). We introduce an automata and Monadic second-order logic-based characterization for this class of languages and we prove their closure properties and the decidability of the emptiness problem, showing that they admit a decidable model checking problem. Furthermore, we study logic formalisms simpler than Monadic second-order logic to define suitable subclasses of OPLs.

On a second line of investigation, this dissertation deals with a further property enjoyed by OPLs that is not exhibited by other families of deterministic context-free languages such as LR and LL, namely their local parsability. Local parsability means that parsing of any substring of a string according to a grammar depends only on information that can be obtained from a local analysis of the portion of the substring under processing and hence is not influenced by parsing of other substrings. The lack of this property implies that parsing algorithms for, e.g., LR and LL languages are inherently sequential and cannot exploit the speedup achievable by a parallel execution on modern multi-core computing platforms: in fact, if an input string is split into several parts, analyzed in parallel by different processing nodes, the parsing actions may require communication among the different processors, with considerable additional overhead. This thesis studies and exploits the local parsability property of OPLs to enable efficient parallel parsing of data description languages (such as, e.g., the JSON standard data format) and programming languages (as, e.g., Lua and JavaScript) and presents a schema for parallelizing also the lexical analysis phase. The algorithms for parallel parsing and lexing have been implemented in a prototype tool (PAPAGENO), which we validated with an extensive experimental campaign, showing that they achieve significant, near-linear speedups on modern multicore architectures, overcoming state of the art sequential parsers and lexers generated by, e.g., Bison and Flex. We exploit the local parsability property enjoyed by OPLs also for efficient parallel querying of large structured and semistructured documents: we examine, as a case study, an extension of the parallel OP parsing algorithm allowing parallel XPath querying of XML documents on multicore machines.

SOMMARIO

I linguaggi Operator Precedence (OPL) furono introdotti da Robert Floyd nel 1963 con l'obiettivo di definire algoritmi deterministici ed efficienti per il parsing di linguaggi di programmazione, ma furono abbandonati pochi anni dopo in seguito all'introduzione di tecniche più generali di parsing, basate sulla classe dei linguaggi LR, che consentono l'analisi dell'intera famiglia dei linguaggi deterministici context-free. L'interesse nella classe dei linguaggi Operator Precedence è ripreso solo di recente con la scoperta di alcune interessanti proprietà di cui essi godono, che rendono possibile l'applicazione di questo formalismo in due principali contesti, naturalmente distanti fra loro: per la specifica e verifica automatica di sistemi software tramite model-checking e per l'elaborazione in parallelo (parsing e interrogazione) di linguaggi di programmazione e descrizione di dati.

Questa tesi studia i fondamenti teorici degli OPL ed in particolare le proprietà che li rendono un formalismo promettente in questi campi applicativi.

In primo luogo, viene proseguita l'attività di ricerca su questa classe di linguaggi risalente agli ultimi decenni, che ha sviluppato una caratterizzazione degli OPL in termini di classi di grammatiche, automi e una logica monadica del secondo ordine. Lo studio delle proprietà algebriche degli OPL, inoltre, ha dimostrato che essi rappresentano la più ampia classe di linguaggi deterministici context-free che gode delle proprietà di chiusura rispetto alle principali operazioni su linguaggi (Booleane, concatenazione, stella di Kleene e altre), includendo strettamente famiglie di linguaggi come linguaggi a parentesi o i linguaggi Visibly Pushdown (VPL). In questa tesi viene presentata una teoria completa degli OPL, estendendo lo stato dell'arte della ricerca su questo formalismo al campo dei linguaggi omega, i.e., linguaggi che consistono di stringhe infinite di simboli e che rappresentano un modello adatto a descrivere il comportamento di sistemi che eseguono le loro computazioni ininterrottamente, come nell'ambito di sistemi operativi, sistemi di controllo, web service. Viene introdotta una caratterizzazione per la classe degli OPL di stringhe infinite basata sulla defini-

zione di opportune famiglie di automi e una logica monadica del secondo ordine, che estende la logica definita per le classi di OPL di parole di lunghezza finita. Inoltre, si dimostra la validità delle proprietà di chiusura e la decidibilità del problema della emptiness per questa classe di linguaggi, mostrando che essi ammettono un problema di model checking decidibile. Vengono quindi analizzate logiche più semplici della logica monadica del secondo ordine per definire opportune sottoclassi degli OPL.

Questa tesi affronta inoltre una seconda linea di ricerca, che si fonda sullo studio della proprietà di local parsability degli OPL. La proprietà di local parsability implica che, nel parsing di una stringa di un linguaggio Operator Precedence, le azioni di shift o di riduzione da eseguire dipendono solo da una analisi locale della posizione corrente della stringa da elaborare e non sono influenzate dal parsing di altri fattori distanti della parola stessa. Le azioni di parsing possono essere quindi compiute con la certezza che non saranno mai condizionate nè invalidate da operazioni di backtracking dovute all'elaborazione di altre porzioni dell'intera stringa. Questa proprietà rende i linguaggi Operator Precedence ideali per il parsing parallelo su architetture multicore, a differenza dei più generali, classici, linguaggi LR per i quali la proprietà di parsabilità locale invece non sussiste. Una stringa di un linguaggio Operator Precedence può essere infatti suddivisa in diversi segmenti, ciascuno dei quali può essere elaborato in parallelo da un diverso processore. La divisione in segmenti può essere completamente arbitraria perchè le azioni di parsing di una stringa sono locali e non dipendono dal contesto precedente. La possibilità di scegliere in modo arbitrario i punti di suddivisione della stringa è una differenza fondamentale che distingue questo approccio di parsing parallelo dai tentativi proposti storicamente in letteratura come per i parser LR, che sono inerentemente sequenziali: suddividendo arbitrariamente la stringa in diversi segmenti, il parsing LR di un segmento da parte di un processore può dipendere da informazioni deducibili solo dall'analisi di altri fattori della parola e che non sono quindi disponibili. Nel parsing parallelo si rende di conseguenza necessaria o una comunicazione tra i processori, con una aggravio sulle performance, o l'assunzione di ipotesi sulle azioni di parsing da compiere con conseguente successivo invalidamento dell'elaborazione compiuta in caso si renda necessario backtracking. Gli approcci al parsing parallelo proposti storicamente in letteratura richiedono quindi, per motivi di performance, che le sottostringhe elaborate dai diversi processori corrispondano ad

opportune e ben definite unità sintattiche del linguaggio: ad esempio blocchi di istruzioni o cicli while per una stringa di codice di un linguaggio di programmazione di cui eseguire il parsing in parallelo. Il vincolo di non poter suddividere il testo in modo arbitrario ha però il significativo svantaggio rispetto al parsing Operator Precedence di rendere la suddivisione dipendente dal particolare linguaggio e soprattutto di impedire una distribuzione equa dei segmenti di stringa da processare, e quindi del carico, sui diversi processori.

Questa tesi studia la proprietà di local parsability degli OPL e ne analizza l'applicazione per il parsing efficiente in parallelo di linguaggi di descrizione di dati, come JSON, e di linguaggi di programmazione, come Lua e JavaScript; inoltre presenta uno schema per parallelizzare anche la fase di analisi lessicale. Gli algoritmi per l'analisi parallela sintattica e lessicale sono stati implementati in un prototipo (chiamato PAPAGENO), e sono stati validati con un'estesa campagna sperimentale, che ha mostrato come essi siano in grado di raggiungere speedup notevoli, quasi lineari, sulle moderne architetture multicore, ottenendo prestazioni significativamente migliori rispetto ai parser LR o ai lexer sequenziali generati da strumenti classici quali Bison e Flex.

Inoltre, questa tesi analizza l'applicazione della proprietà di local parsability di cui godono gli OPL per consentire l'interrogazione efficiente in parallelo di documenti strutturati o semistrutturati di grandi dimensioni: si presenta, come caso di studio, un'estensione dell'algoritmo di parsing in parallelo per gli OPL per l'elaborazione in parallelo di query XPath su documenti XML su architetture multiprocessore.

ACKNOWLEGMENTS

I would like to thank my advisor, Matteo Pradella, for his guidance through the years of my Ph.D.

I'd also like to thank Dino Mandrioli, Violetta Lonati, Stefano Crespi Reghizzi and Alessandro Barenghi for their advices and the pleasant collaboration on the research work of this thesis.

Finally, thanks to my family and friends.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I.

Prologue

1 INTRODUCTION

Nowadays computing systems have reached an impressive level of complexity.

Software systems are expected to perform increasingly complex and interdependent tasks and pervade every aspect of human life in several contexts. Furthermore, computing systems in numerous application domains (ranging from genomics and medical research to financial data processing, web services and network monitoring) are required to parse, search and analyze impressively huge volumes of structured and unstructured data.

To deal with these dimensions of complexity, on the one side, it is necessary to introduce formalisms apt to model complex (often non finite-state) systems, along with tractable verification procedures to check adequate safety or correctness requirements on their behavior, so that dire consequences due to malfunctions can be prevented. On the other side, there is an urgent and pressing need for models and technologies that can process massive datasets fast and efficiently.

In this thesis we address these two facets of complexity of software systems at the application and data level in a unifying framework: we study the old, and for long almost forgotten, formalism of Operator Precedence Languages (OPLs) and we investigate the theoretical properties that allow for its application in such diverse and challenging fields.

Operator precedence languages and their class of generating grammars (Operator Precedence Grammars or OPGs) were introduced half a century ago by Robert Floyd [47] with the major motivation of enabling efficient, deterministic parsing of programming languages. Floyd defined this new family of grammars by taking inspiration from the structure of arithmetic expressions, which is determined either by explicit parentheses or by the conventional, "hidden" *precedence* of multiplicative operators over additive ones. By generalizing this observation, Floyd defined three basic relations between terminal symbols, namely *yields* and *takes precedence* and *equal in* *precedence* (respectively denoted by symbols $\lessdot$, $\gtrdot$, $\doteq$), in such a way that the right hand side (r.h.s.) of an operator precedence grammar rule is enclosed within a pair $\lessdot$, $\gtrdot$, and $\doteq$ holds between consecutive terminal symbols thereof.

Subsequently, under the main motivation of grammar inference, it was shown that, once an operator precedence matrix (OPM) is given such that at most one relation holds between any two terminal characters, the family of OPLs sharing the given OPM is a Boolean algebra [35]. This result somewhat generalizes closure properties enjoyed by regular languages and by context-free languages whose structure, i.e., the syntax tree, is immediately visible in the terminal sentences, such as parenthesis languages [75] and tree-automata languages [22]. Such interesting algebraic properties enabled original inference algorithms, such as those proposed in [36]. After these initial results, however, the theoretical investigation of OPLs was almost abandoned because of the advent of more general grammars, mainly the LR ones [62], which support parsing algorithms for the whole class of deterministic context-free languages (whereas OPLs are a strict subclass thereof). Nevertheless OPG-based parsing remains of some interest thanks to its simplicity and efficiency and is still used, at least partially, in many practical cases [54].

Oblivion for OPLs continued only until decades later, when new branches of research began to flourish to deal with the two stated challenges in software verification and data analysis. Novel interest for parentheses-like classes of languages arose in fact from research on mark-up languages, such as XML and HTML, which were used to specify structured and semistructured documents; investigation of such classes was further motivated by the wish to extend to families of languages, larger than finite-state ones, the properties that enable model checking that were typically enjoyed by regular languages, i.e., the closure properties for Boolean operations and the decidability of the emptiness problem.

Among the results of this research activity, Visibly Pushdown Languages (VPLs) [11], previously known as Input-Driven Languages (IDLs) [98] certainly deserve a major role. In a nutshell IDLs alias VPLs are based on and extend original parenthesis languages [75], e.g., by allowing for unmatched closed and open parentheses at the beginning and end of a sentence, respectively. VPLs provide a simple and natural model for software verification, as they may represent the control flow of sequential computations in typical programming languages with nested, and even recursive, invocations of program modules, where parentheses symbols model occurrences of matching opening and closing scopes, as for calls and returns of procedures and methods. The parenthesis structure can model as well the matching of open and close-tag constructs in XML documents. However, being essentially a generalization of parenthesis languages, VPLs have some limits in expressive power: the structure of their strings is immediately transparent at the "surface sentence", unlike more general context-free languages; arithmetic expressions, e.g., which are found in practically every programming language, do not reflect in the sequence of the leaves of the syntax tree the internal structure of the tree, which can be built only by knowing that multiplication operators take precedence over the additive ones.

VPLs have been characterized both in terms of a subclass of context-free grammars and a particular class of automata, named Visibly Pushdown Automata (VPAs), which resemble classical pushdown automata but differ from them in that operations on the stack are input-driven. For every VPL, the input alphabet is partitioned into call, return and internal symbols, and the letters of the input word determine the corresponding transitions of the automaton, i.e. respectively push, pop and neutral moves (which leave the stack unchanged). Furthermore, they are closed w.r.t. to all fundamental language operations (Boolean, concatenation, Kleene *,...), like regular languages and unlike more general context-free families, and admit a decidable emptiness problem.

Along the field of research marked by VPLs, interest in OPLs has been renewed thanks to two unexpected properties thereof, that appear to be crucial for their application in the fields of verification and efficient data processing.

The OPL family strictly includes VPLs and other related classes such as balanced languages [16], and it was shown that OPLs are the largest known class of languages that enjoys all major closure properties that are typical of regular languages [34][1]. Herewith the first goal of this thesis is to apply to OPLs the same successful verifi-

1 Other language families falling in between input-driven and context-free languages, such as the height-deterministic family [79] or the synchronized pushdown languages [26], enjoy some but not all of the basic closure properties; furthermore such families are in general nondeterministic.

cation techniques formerly developed for regular languages, VPLs, and other –input driven– language families.

The second main property enjoyed by OPLs is their *local parsability*, which consists in the fact that the typical shift-reduce parsing algorithm associated with these languages can determine the replacing of a rule r.h.s. by the corresponding left hand side (l.h.s) exclusively on the basis of the embracing $\lessdot$ and $\gtrdot$ relations, i.e., independently on parts of the string that may be arbitrarily far from the considered r.h.s. This property is not enjoyed by more powerful grammars such as LR ones and allows for efficiently exploiting parallelism on modern computing platforms to parse large strings that formalize complex systems and their behavior. It is, in fact, possible to distribute the execution of the parsing algorithm among different independent machines and to recombine their results afterwards with efficient transformations. The inherent left-to-right sequential nature of parsing algorithms for LR or LL languages, instead, prevents them from exploiting the potential speedup deriving by the use of a data-parallel approach on multicore machines; as a consequence, in the last years almost none of the other attempts that have been made to parallelize the classical parsing algorithms for deterministic context-free languages has been successful. Moreover, the local parsability property makes OPLs amenable also for efficient parallel processing (searching or querying) of large structured and semistructured documents.

The exploitation of this property is the target of a recent and –so far– independent branch of research whose first results, as regards simple parsing, are documented in [14] and [12], and that in this thesis we further deepen and explore.

1.1 CONTRIBUTION

In this thesis we study the theory of OPLs as regards these fundamental properties and their application to infinite-state verification and parallel data processing. The main contributions of this work are summarized in the following.

- Along the path begun with [35] and resumed with [34, 70, 71], OPLs have been characterized by a class of grammars (OPGs), a new family of pushdown automata with the same recognizing power as the generative power of their

grammars (Operator Precedence Automata, or OPAs), and a suitable monadic second-order (MSO) logic, following a now classic approach of the literature rooted in the work by Büchi [23].
In this thesis we revise the model of operator precedence automaton, yielding a more elegant and simplified definition w.r.t. the original formulation proposed in [70], and we further investigate the class of OPLs extending them to ω-languages (i.e., languages of strings of infinite length) and defining the family of ωOPLs: ω-languages are, in fact, becoming more and more relevant in the literature due to the need of modeling systems whose behavior proceeds indefinitely, such as operating systems, control systems, etc. In this work, after introducing and comparing various forms of acceptance of infinite words for the model of OPAs, by paralleling classic literature of ω-regular languages, we study their main properties by pointing out which of them are preserved from the finite length case and which are lost. We then characterize the class of ωOPLs in terms of a MSO logic. The availability of a MSO logic formulation (for finite-length as well for infinite-length OPLs) allows, at least potentially, the definition of model-checking algorithms able to prove properties of languages defined either by means of generating grammars or by means of recognizing automata on finite or infinite words. Given the prohibitive complexity of decision algorithms based on MSO logic, however, it is common practice in the literature to resort to model-checking algorithms based on less powerful but simpler logics. Furthermore, we study a subclass of OPGs, namely Free Grammars (FrGs), which was originally introduced in 1970s for grammatical inference of programming languages, and we prove that their languages (FrLs) can be defined by formulae written in a first-order logic that restricts the MSO one defined for general OPLs. We show that FrLs suffer from some generative power limits; however, they enjoy some distinguishing properties that make it possible to define a system of interest by stepwise refinements, by inferring a "skeleton language" (notably, FrGs can be easily inferred on the basis of positive samples only of strings of the user's desired language) which can be further specified by stating additional required properties –in first-order logic– on its

behavior.
The results of these investigations have been presented in [69, 81, 68].

- On a complementary line of investigation, we studied and exploited the local parsability property of OPLs to enable efficient parallel parsing of programming languages, such as Lua and JavaScript, and data description languages, such as the JSON standard data format. Their minor lack of power w.r.t. deterministic context-free languages, in fact, does not prevent them from including most programming languages of practical interest.
  This thesis proposes a schema for parallelizing the lexical analysis phase in addition to the syntactic (parsing) one. Furthermore, it integrates the lexical analysis stage with a further processing step which is beneficial to the subsequent phase of syntactic analysis: the lexer is used to generate (and transform) a string of symbols in a form amenable for analysis by an OP parser, hence allowing for dealing also with languages whose grammar is not directly expressible as an OPG. We implemented a parallel lexer for languages as Lua and JSON and we validated the performance of the parallel lexing and parsing algorithms with an extensive experimental campaign. On the basis of the benchmark we considered so far, we showed significant, near-linear speedups on modern multi-core architectures (server and mobile).
  The results of this study have been presented in [12, 13].
  Finally, we exploited the local parsability property in contexts other than the parallelization of the initial compilation phases of lexical and syntactic analysis, analyzing its application to support a semantic analysis phase. As a consequence, we extended the efficient parallel parsing algorithm for OPLs to deal also with queries on (streaming or offline) data documents, which can be specified by rich data formats as JSON or domain specific languages that cannot be otherwise described by formalisms that are less expressive than OPLs.

1.2 THESIS STRUCTURE

The thesis is structured in two main parts.

In the first part we present a complete characterization of OPLs of finite and infinite-length words. More precisely, Chapter 2 provides all basic definitions and reminds the fundamental results on OPLs of finite words, presenting a complete grammar, automata and logic-based characterization thereof. Chapter 3 introduces ωOPLs, i.e., OPLs whose words have infinite length, a now classic extension of most language families due to the need of modeling never-ending computations, and we characterize them in terms of suitable classes of automata and a MSO logic. Chapter 4 deals with FrLs and presents a first-order logic definition for this class of languages.

The second part of this thesis outlines the basic theory on local parsability of OPLs: after describing the classical operator precedence parsing algorithm and its recent generalization to a parallel setting [14], Chapter 5 proposes a methodology for parallel lexical analysis and discusses the results of the experimental campaign we performed. Chapter 6, instead, illustrates an approach based on operator precedence parsing to parallelize query processing on structured and semistructured documents.

Finally, Chapter 7 draws some conclusions and hints at some future research directions.

Part II.

Theory of OPLs and ωOPLs

2

GRAMMAR, AUTOMATA AND LOGIC-BASED CHARACTERIZATION OF OPLS

This chapter presents a complete characterization of finite length OPLs, which was introduced in [70, 71]. Besides the original formalization in terms of the class of operator precedence grammars (OPGs) proposed by R. Floyd in 1960s, it is shown that OPLs can be characterized by a family of pushdown automata, which perfectly matches the generative power of OPGs, and by a MSO logic. We also recall the basic properties of OPLs: most notably, they represent the largest known family of languages closed w.r.t. to all classical language operations and that admits a decidable emptiness problem.

The chapter is structured as follows. Section 2.1 introduces the basic terminology and the definitions and properties of OPGs. Then, Section 2.2 presents a class of pushdown automata explicitly tailored at OPLs, providing some examples to show their usefulness in modeling various cases of practical interest. Section 2.2.2, in particular, shows the equivalence between deterministic and nondeterministic versions of these automata, which leads to an increase in state space size given by an exponential function with quadratic exponent. In Section 2.3 we illustrate, in a constructive way, the equivalence between OPGs and the new class of automata. We also report the complexity of decision problems for OPLs. Finally, Section 2.4 presents a characterization of OPLs in terms of an "equivalent" MSO logic: the starting point is the classic result by Büchi given for regular languages; however, its extension to OPLs required facing non-trivial technical problems even w.r.t. to previous similar extensions to other subclasses of context-free languages, such as VPLs [11].

2.1 OPERATOR PRECEDENCE GRAMMARS AND LANGUAGES

We recall some basic definitions and notations on formal languages and grammars; for terms not defined here, we refer to any classical textbook on formal language theory (e.g., [55]).

A *context-free* (CF) grammar is a 4-tuple $G = (N, \Sigma, P, S)$, where N is the nonterminal alphabet, Σ is the terminal alphabet, P the rule (or production) set, and $S \subseteq N$ the set of axioms[2].

The following naming convention will be adopted, unless otherwise specified: lowercase Latin letters $a, b, \ldots$ denote terminal characters; uppercase Latin letters $A, B, \ldots$ denote nonterminal characters; letters $u, v, \ldots$ denote terminal strings; and Greek letters $\alpha, \beta, \ldots$ denote strings over $\Sigma \cup N$. The strings may be empty, unless stated otherwise. The empty string is denoted ε. Given a string x, its length is denoted as $|x|$ and the i-th character in x, for $1 \le i \le |x|$, is denoted as $x[i]$.

A rule in P is denoted by $A \to \alpha$, where $A \in N$, $\alpha \in V^*$, where $V = \Sigma \cup N$ is the set of *grammar symbols*; A is called the left hand side (l.h.s.) of the production and α the right hand side (r.h.s.). An *empty rule* has ε as the r.h.s. A *renaming rule* has one nonterminal as r.h.s.

The symbol $\Rightarrow$ denotes the *direct derivation* relation between two strings in V^*: given $\alpha, \beta \in V^*$, α is said to directly derive β, written $\alpha \Rightarrow \beta$ if there exist $\alpha_1, \alpha_2, \alpha', \beta' \in V^*$ such that $\alpha = \alpha_1 \alpha' \alpha_2$ and $\beta = \alpha_1 \beta' \alpha_2$ and $\alpha' \to \beta'$ is in P. The transitive closure of $\Rightarrow$ is denoted by $\overset{+}{\Rightarrow}$, and its reflexive transitive closure, the *derivation* relation, is denoted by $\overset{*}{\Rightarrow}$. If $\alpha \overset{*}{\Rightarrow} \beta$ in h steps, we write $\alpha \overset{h}{\Rightarrow} \beta$.

Also, we distinguish between the terminal strings u derived by the grammar, i.e., such that $A \overset{*}{\Rightarrow} u$ (for an $A \in S$), and *sentential forms*, i.e., those $\alpha \in V^*$ such that $A \overset{*}{\Rightarrow} \alpha$ (for an $A \in S$). $L(G)$ denotes the language (of terminal strings) generated by G.

A grammar is *reduced* if every rule can be used to generate some string in Σ^*. It is *invertible* if no two rules have identical r.h.s.

2 This less usual but equivalent definition of axioms as a set has been adopted for parenthesis languages [75] and other input-driven languages; we chose it to simplify some notations and constructions.

In this initial chapter we will use arithmetic expressions, which are a small fraction of practically all programming languages, as a running example to introduce and explain the basic definitions, properties and constructions referring to OPLs.

Example 2.1. We consider arithmetic expressions with two operators, an additive one and a multiplicative one that takes precedence over the former, in the sense that, during the interpretation of the expression, multiplications must be executed before sums; as usual parentheses are used to specify a different precedence hierarchy between the two operations. Parentheses are denoted by the special symbols $\llparenthesis$ and $\rrparenthesis$ to avoid overloading with the use of the same symbol in all other formulae of this thesis. Figure 1 presents a grammar and the derivation tree of expression $n + n \times \llparenthesis n + n \rrparenthesis$ generated thereby; all nonterminals are also axioms.

Notice that the structure of the syntax tree (uniquely) corresponding to the input expression reflects the precedence order which drives computing the value attributed to the expression. This structure, however, is not immediately visible in the expression: in fact Figure 2 proposes a different grammar that generates the same expressions as the grammar of Figure 1 but would associate to the same sentence the syntax tree displayed in the right part of the figure. Yet another (ambiguous) grammar could generate both. If instead we used a parenthesis grammar to generate arithmetic expressions, it would produce the string $\llparenthesis n + \llparenthesis n \times \llparenthesis n + n \rrparenthesis \rrparenthesis \rrparenthesis$ instead of the previous one and the structure of the corresponding tree would be immediately visible in the expression. For this reason we say that such general grammars “hide” the structure associated with a sentence –even when they are unambiguous– whereas parenthesis grammars and other input-driven ones make the structure explicit in the sentences they generate.

A string on V^* is in *Operator Form* (OF) if it has no adjacent nonterminals. In an OF string α we say that two terminals are *consecutive* if they are at positions $\alpha[j], \alpha[j+1]$; or at positions $\alpha[j], \alpha[j+2]$ and $\alpha[j+1] \in N$. A rule is in *operator form* if its r.h.s. has no adjacent nonterminals (i.e., is in OF); an *operator grammar* (OG) contains just such rules. Notice that both grammars of Figure 1 and Figure 2 are OGs.

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T \times F \mid F \\ F &\to n \mid \llparenthesis E \rrparenthesis \end{aligned}$$

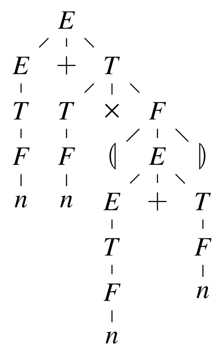


Figure 1.: A grammar generating arithmetic expressions with parentheses.

$$\begin{aligned} A &\to B \times A \mid B \\ B &\to B + C \mid C \\ C &\to n \mid \llparenthesis A \rrparenthesis \end{aligned}$$

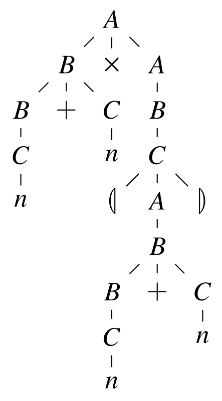


Figure 2.: A grammar generating the same arithmetic expression as that of Figure 1 and the corresponding tree where, instead, $+$ takes precedence over $\times$.

Every CF grammar $G = (N, \Sigma, P, S)$ admits an equivalent OG $G' = (N', \Sigma, P', S)$, where the size of N' is $O(|\Sigma| \cdot (|\Sigma| + k \cdot |P|))$ and that of P' is $O(|\Sigma| \cdot (|N| + k \cdot |\Sigma| \cdot |P|))$, k being the maximum length of P's r.h.s.s [55, 88].

The coming definitions for operator precedence grammars (OPGs) [47] are from [35] and [34], where they are also called *Floyd Grammars* or FGs.

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

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

where $B \in N \cup \{\varepsilon\}$. The grammar name G will be omitted unless necessary to prevent confusion.

For instance, for the grammar of Figure 1 the left and right terminal sets of nonterminals E, T and F are, respectively:

$$\begin{aligned} \mathcal{L}(E) &= \{+, \times, n, (\!|\} & \mathcal{R}(E) &= \{+, \times, n, |\!)\} \\ \mathcal{L}(T) &= \{\times, n, (\!|\} & \mathcal{R}(T) &= \{\times, n, |\!)\} \\ \mathcal{L}(F) &= \{n, (\!|\} & \mathcal{R}(F) &= \{n, |\!)\} \end{aligned}$$

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

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

Notice that, unlike the usual arithmetic relations denoted by similar symbols, the above precedence relations do not enjoy any of transitive, symmetric, reflexive properties.

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.

	$+$	$\times$	$\llparenthesis$	$\rrparenthesis$	n
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$\times$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$\llparenthesis$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
$\rrparenthesis$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
n	$\gtrdot$	$\gtrdot$		$\gtrdot$	

Figure 3.: The OPM of the grammar in Figure 1.

$A \to aAa \mid aBa$
$B \to b$

	a	b
a	$\{\lessdot, \doteq, \gtrdot\}$	$\lessdot$
b	$\gtrdot$	

Figure 4.: A grammar generating the language $\{a^n b a^n \mid n \geq 1\}$ and its OPM.

Figure 3 displays the OPM associated with the grammar of Figure 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 line.

As a further example, Figure 4 shows a CF grammar that generates the language $\{a^n b a^n \mid n \geq 1\}$ and its associated OPM. Note that $|M_{aa}| > 1$.

Given two OPMs M_1 and M_2, we define set inclusion and union:

$$\begin{aligned} M_1 \subseteq M_2 \quad & \text{if } \forall a, b : (M_1)_{ab} \subseteq (M_2)_{ab}, \\ M = M_1 \cup M_2 \quad & \text{if } \forall a, b : M_{ab} = (M_1)_{ab} \cup (M_2)_{ab} \end{aligned}$$

Definition 2.1 (Operator precedence grammar and language)**.** An OG G is an *operator precedence* or *Floyd grammar* (*OPG*) if, and only if, $M = OPM(G)$ 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.

From the above definition it is immediate to verify that both grammars of Figure 1 and Figure 2 are OPGs (with different OPMs).

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

The following definition of Fischer Normal Form is adapted from the original one [46] to take into account that in our basic definition of CF grammar S is a set rather than a singleton.

Definition 2.2 (Fischer Normal Form)**.** An OPG is in *Fischer normal form* (FNF) iff it is invertible, has no empty rule except possibly $A \to \varepsilon$, where A is an axiom not used elsewhere, and no renaming rules.

Let $G = (N, \Sigma, P, S)$ be an OPG; then an equivalent OPG $\widetilde{G} = (\widetilde{N}, \Sigma, \widetilde{P}, \widetilde{S})$ in FNF, can be built such that $\widetilde{N}$ is $\wp(N)$ and $|\widetilde{P}|$ is $O(|P| \cdot 2^{|N| \cdot \lceil \frac{k}{2} \rceil})$, where k is the maximum length of P's r.h.s.s [55].

A FNF (manually) derived from the grammar of Figure 1 is given below (Figure 5). Notice that in this case the size of the nonterminal alphabet and of the productions is much smaller than the worst case upper bound provided by the general construction.

$$\begin{array}{l} E \to \; E+T \;\mid\; E+F \;\mid\; T+T \;\mid\; F+F \;\mid\; F+T \;\mid\; T+F \\ T \to \; T \times F \;\mid F \times F \\ F \to \; n \;\mid\; (\!|E|\!) \;\mid\; (\!|T|\!) \mid\; (\!|F|\!) \end{array}$$

Figure 5.: OPG of arithmetic expressions of Figure 1 in FNF.

It is well-known that OPLs are a proper subfamily of deterministic context-free languages: for instance, it is impossible to generate the language $\{a^n b a^n\}$ without producing a precedence conflict, since matching n requires (at least) the conflicting precedences $a \lessdot a$ and $a \gtrdot a$. Despite this theoretical limitation OPLs have been successfully used to formalize many programming languages and to support their compilers, but there are several other examples of potential application of this model in different fields.

OPMs play a fundamental role in deterministic parsing of OPLs (the classical algorithm for parsing operator precedence languages is presented in detail in [54] and will be described in Chapter 5). In the view of defining automata to recognize/parse OPLs (Operator Precedence Automata or OPAs) we define OPMs on an extended alphabet: we use a special symbol # not in Σ to mark the beginning and the end of any string; this is consistent with the typical operator parsing technique which requires the look-back and lookahead of one character to determine the precedence relation between the symbols. The precedence relations in the OPM are implicitly extended to include #: the initial # can only yield precedence, and other symbols can only take precedence over the ending #.

Definition 2.3 (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).

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 $a \circ b$. The relations involving the # delimiter are constrained as stated above.

The notion of chain introduced by the following definitions provides a formal description of the intuitive concept of "invisible or hidden structure" discussed in Example 2.1.

Definition 2.4 (Chains)**.** Let (Σ, M) be an operator precedence alphabet.

- A *simple chain* is a word $a_0 a_1 a_2 \dots a_n a_{n+1}$, written as ${}^{a_0}[a_1 a_2 \dots a_n]^{a_{n+1}}$, such that: $a_0, a_{n+1} \in \Sigma \cup \{\#\}$, $a_i \in \Sigma$ for every $i : 1 \leq i \leq n$, $M_{a_0 a_{n+1}} \neq \emptyset$, 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 $a_0 x_0 a_1 x_1 a_2 \dots a_n x_n a_{n+1}$, with $x_i \in \Sigma^*$, where ${}^{a_0}[a_1 a_2 \dots a_n]^{a_{n+1}}$ 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 ${}^{a_0}[x_0 a_1 x_1 a_2 \dots a_n x_n]^{a_{n+1}}$.
- The *body* of a chain ${}^{a}[x]^{b}$, simple or composed, is the word x.

Example 2.2. The "hidden" structure induced by the operator precedence alphabet of Example 2.1 for the expression $\#n + n \times (\!|n + n|\!)\#$ is represented in Figure 6, where ${}^{\#}[x_0 + x_1]^{\#}$, ${}^{+}[y_0 \times y_1]^{\#}$, ${}^{\times}[(\!|w_0|\!)]^{\#}$, ${}^{(\!|}[z_0 + z_1]^{|\!)}$ are composed chains and ${}^{\#}[n]^{+}$, ${}^{+}[n]^{\times}$, ${}^{(\!|}[n]^{+}$, ${}^{+}[n]^{|\!)}$ are simple chains.

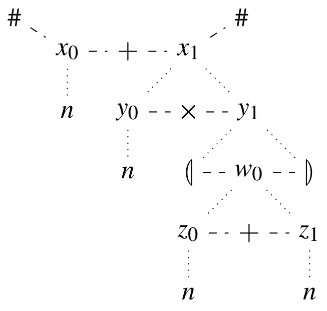


Figure 6.: Structure of the chains in the expression $\#n + n \times (\!|n + n|\!)\#$ of Example 2.2
.

Definition 2.5 (Depth of a chain)**.** Given a chain ${}^{a}[x]^{b}$ the *depth* $d(x)$ of its body x is defined recursively: $d(x) = 1$ if the chain is simple, whereas $d(x_0 a_1 x_1 \dots a_n x_n) = 1 + \max_i d(x_i)$. The depth of a chain is the depth of its body.

For instance, the composed chain ${}^{\#}[x_0 + x_1]^{\#}$ in Example 2.2 has depth 5.

Thus, if for an OPG G it is $OPM(G) = M$, the depth of a chain body x is the height of the syntax tree, if any, whose frontier is x.

Definition 2.6 (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 factor (substring) x of $\#w\#$ such that $x = a_0 x_0 a_1 x_1 a_2 \dots\ a_n x_n a_{n+1}$, if $a_0 \lessdot a_1 \doteq a_2 \dots a_{n-1} \doteq a_n \gtrdot a_{n+1}$ 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 $n + n \times (\!|n + n|\!)$ is compatible with the operator precedence alphabet of Example 2.1, whereas $n + n \times (\!|n + n|\!)(\!|n + n|\!)$ is not.

The chains fully determine the structure of the words; in particular, given an OP alphabet, each word in Σ^* compatible with M is assigned a tree-structure by the OPM M. If M is complete, then each word is compatible with M and the OPM M assigns a structure to any word in Σ^*. For this reason we say that OPLs somewhat generalize the notion of Input-driven languages, since their parsing is driven by the OPM which is defined on the terminal alphabet, but they also allow for generating sentences whose structure is "invisible" before parsing.

The equal in precedence relations of an OP alphabet are connected with an important parameter of the grammar, namely the length of the right hand sides of the rules. Clearly, a rule $A \to A_1 a_1 \ldots A_t a_t A_{t+1}$, where each A_i is a possibly missing nonterminal, is associated with relations $a_1 \doteq a_2 \doteq \ldots \doteq a_t$. If the $\doteq$ relation is *cyclic*, i.e., there exist $a_1, a_2, \ldots, a_n \in \Sigma$ $(n \geq 1)$ such that $a_1 \doteq a_2 \doteq \ldots \doteq a_n \doteq a_1$, there is *a priori* no finite bound on the length of the r.h.s. of a production. Otherwise the length is bounded by $2 \cdot c + 1$, where $c \geq 1$ is the length of the longest $\doteq$-chain.

Most literature [34, 70] assumed that all precedence matrices of OPLs are $\doteq$-cycle free. In the case of OPGs this prevents the risk of r.h.s. of unbounded length [35], but could be replaced by the weaker restriction of production's r.h.s. of bounded length, or could be removed at all by allowing such unbounded forms of grammars –e.g. with regular expressions as r.h.s. In our experience, such assumption helps to simplify notations and some technicalities of proofs; moreover we found that its impact in practical examples is minimal. Hence we accept a minimal loss of generation[3] power and we assume the simplifying assumption of $\doteq$-acyclicity: this hypothesis has an impact only on constructions involving grammars but is irrelevant for the definition of OP automata.

2.1.1 *Closure properties of OPLs*

Herein we present some basic properties of OPLs stated in [35, 34]. Preliminarily, notice that, since the union of two acyclic OPMs might be cyclic, when we consider, in the sequel, the union $M = M_1 \cup M_2$ of two OPMs M_1 and M_2 we always assume that M too is acyclic.

3 An example language that cannot be generated with an $\doteq$-acyclic OPM is the following: $\mathcal{L} = \{a^n(bc)^n \mid n \geq 0\} \cup \{b^n(ca)^n \mid n \geq 0\} \cup \{c^n(ab)^n \mid n \geq 0\}$

Statement 2.1. *[35] OPLs are closed with respect to Boolean operations. Precisely, given two OPLs L_1, L_2 with compatible OPMs M_1 and M_2, $L_1 \cap L_2$ and $L_1 \cup L_2$ are OPLs whose OPM is contained in $M_1 \cup M_2$; furthermore, let L_1^{max} be the OPL of all strings compatible with M_1, then $L_1^{max} \setminus L_1$ is an OPL whose OPM is contained in M_1. In particular, if M_1 is a complete OPM, L_1^{max} is Σ^* (where each sentence has a structure determined by M_1); then $\Sigma^* \setminus L_1$ is an OPL whose OPM is contained in M_1.*

Statement 2.2. *[34] OPLs are closed with respect to concatenation and Kleene $*$ operation. Precisely, given two OPLs L_1, L_2 with compatible OPMs M_1 and M_2, $L_1.L_2$ and L_1^* are OPLs whose OPM is compatible with $M_1 \cup M_2$ (resp. M_1). Notice that in this case the construction of the new grammars may introduce new precedence relations not existing in the original matrices. Furthermore, OPLs are closed under alphabetical homomorphisms that preserve conflict-freedom.*

2.1.2 *Language family relationships*

[34] compares the generative power and the structural adequacy of OPGs versus other well-known grammar or automata families: in particular, besides strictly including the classes of regular and balanced languages [17], the following fundamental results holds.

Statement 2.3. *[34] OPLs strictly include the family of VPLs. Precisely, VPLs are the subfamily of OPLs whose OPM is a* partitioned matrix*, i.e., a matrix whose structure is depicted in Figure 7.*

2.2 OPERATOR PRECEDENCE AUTOMATA

In this section we introduce a family of pushdown automata that recognize exactly OPLs. OPLs being naturally oriented towards bottom-up parsing, their accepting automata exhibit a typical shift-reduce attitude; they are considerably simpler, however, than other classical automata of this type such as LR ones.

	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"
i.e. a generalized version of open parentheses;
Σ_r denotes "returns"
i.e. a generalized version of closed parentheses;
Σ_i denotes internal characters
i.e., characters that are not pushed onto the stack and are managed exclusively by finite state control.

Figure 7.: A partitioned matrix, where Σ_c, Σ_r and 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.

Definition 2.7 (Operator precedence automaton)**.** A nondeterministic *operator precedence automaton (OPA)* is given by a tuple: $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ where:

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

$$\delta_{\text{shift}} : Q \times \Sigma \to \wp(Q) \qquad \delta_{\text{push}} : Q \times \Sigma \to \wp(Q) \qquad \delta_{\text{pop}} : Q \times Q \to \wp(Q)$$

We represent a nondeterministic OPA by a graph with Q as the set of vertices and $\Sigma \cup Q$ as the set of edge labelings. 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 if and only if $p \in \delta_{\text{shift}}(q, a)$ (respectively, $p \in \delta_{\text{push}}(q, a)$) and there is an edge from state q to state p labeled by $r \in Q$ and denoted by a double arrow if and only if $p \in \delta_{\text{pop}}(q, r)$.

To define the semantics of the automaton, we introduce some notations.

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

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

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

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

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

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

Notice that shift and pop moves are never performed when the stack contains only $\bot$.

Push and shift moves update the current state of the automaton according to the transition function δ_{push} and δ_{shift}, respectively: push moves put a new element on the top of the stack consisting of the input symbol together with the current state of the automaton, whereas shift moves update the top element of the stack by changing its input symbol only. The pop move removes the symbol on the top of the stack, and the state of the automaton is updated by 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 move the input symbol is used only to establish the $\gtrdot$ relation and it remains available for the following move.

We say that a configuration $\langle \bot,\ q_I,\ x\#\rangle$ is *initial* if $q_I \in I$ and a configuration $\langle \bot,\ q_F,\ \#\rangle$ is *accepting* if $q_F \in F$. The language accepted by the automaton is defined as:

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

Example 2.3. The OPA depicted in Figure 8 accepts the language of arithmetic expressions generated by the OPG of Example 2.1. The same figure also shows an accepting computation on input $n + n \times (\!| n + n |\!)$.

Therefore, an OPA selects an appropriate subset within the "universe" of strings in Σ^* compatible with M. This property somewhat resembles the fundamental Chomsky-Shützenberger Theorem, in that a universe of nested structures –a Dyck language– is restricted by means of an "intersection" with a finite state mechanism. For instance, the automaton of Figure 8 recognizes well-nested parenthesized arithmetic expressions and could be modified in such a way that parentheses are used only when needed to give the expression the desired meaning, i.e., a pair of parentheses containing a $+$ is necessary only if it is adjacent to a $\times$; parentheses enclosing only $\times$ should be avoided.

In the sequel we use the following notation to characterize OPA behavior: we use arrows $\longrightarrow$, $\dashrightarrow$ and $\Longrightarrow$ to denote push, shift and pop transitions, respectively.

Definition 2.8. Let $\mathcal{A}$ be an OPA. A *support* for a simple chain ${}^{a_0}[a_1 a_2 \ldots a_n]^{a_{n+1}}$ is any path in $\mathcal{A}$ of the form

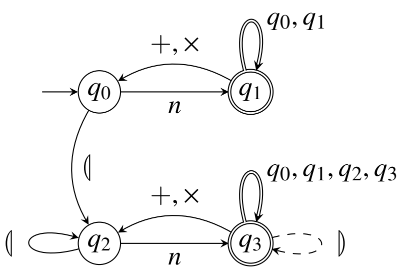
$$q_0 \xrightarrow{a_1} q_1 \dashrightarrow \ldots \dashrightarrow q_{n-1} \overset{a_n}{\dashrightarrow} q_n \overset{q_0}{\Longrightarrow} q_{n+1} \qquad (1)$$

Notice that the label of the last (and only) pop is exactly q_0, i.e. the first state of the path; this pop is executed because of relations $a_0 \lessdot a_1$ and $a_n \gtrdot a_{n+1}$.
A *support for the composed chain* ${}^{a_0}[x_0 a_1 x_1 a_2 \ldots a_n x_n]^{a_{n+1}}$ is any path in $\mathcal{A}$ of the form

$$q_0 \overset{x_0}{\rightsquigarrow} q'_0 \xrightarrow{a_1} q_1 \overset{x_1}{\rightsquigarrow} q'_1 \overset{a_2}{\dashrightarrow} \ldots \overset{a_n}{\dashrightarrow} q_n \overset{x_n}{\rightsquigarrow} q'_n \overset{q'_0}{\Longrightarrow} q_{n+1} \qquad (2)$$

where for every $i : 0 \leq i \leq n$:

- if $x_i \neq \varepsilon$, then $q_i \overset{x_i}{\rightsquigarrow} q'_i$ is a support for the (simple or composed) chain ${}^{a_i}[x_i]^{a_{i+1}}$



stack	state	current input
$\perp$	q_0	$n + n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[n,\ q_0]$	q_1	$+n \times \llparenthesis n + n \rrparenthesis \#$
$\perp$	q_1	$+n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ q_1]$	q_0	$n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ q_1][n,\ q_0]$	q_1	$\times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ q_1]$	q_1	$\times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1]$	q_0	$\llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0]$	q_2	$n + n \rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0][n,\ q_2]$	q_3	$+n \rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0]$	q_3	$+n \rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0][+,\ q_3]$	q_2	$n \rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0][+,\ q_3][n,\ q_2]$	q_3	$\rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0][+,\ q_3]$	q_3	$\rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\llparenthesis,\ q_0]$	q_3	$\rrparenthesis \#$
$\perp[+,\ q_1][\times,\ q_1][\rrparenthesis,\ q_0]$	q_3	$\#$
$\perp[+,\ q_1][\times,\ q_1]$	q_3	$\#$
$\perp[+,\ q_1]$	q_3	$\#$
$\perp$	q_3	$\#$

Figure 8.: Automaton and example of computation for the language of Example 2.3. Recall that shift, push and pop transitions are denoted by dashed, normal and double arrows, respectively.

- if $x_i = \varepsilon$, then $q'_i = q_i$.

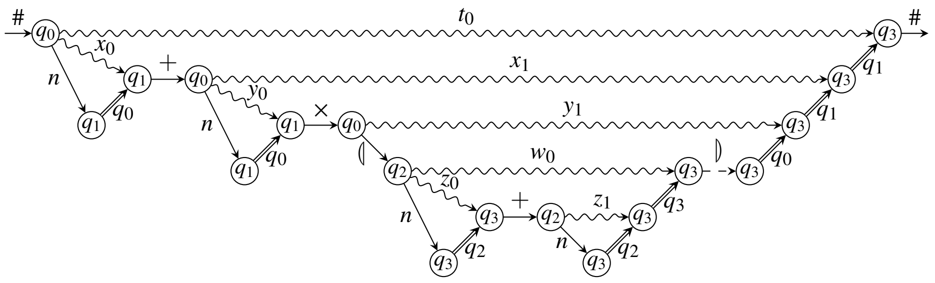


Figure 9.: Structure of chains and supports for the expression of Example 2.4.

Notice that the label of the last pop is exactly q_0'.
The support of a chain with body x will be denoted by $q_0 \overset{x}{\rightsquigarrow} q_{n+1}$.

Example 2.4. Figure 9 illustrates the supports of the chains that, for the OPA described in Example 2.3, compose the structure of the expression $n + n \times (\!| n + n |\!)$.

The chains fully determine the structure of the computation of any automaton on a word compatible with M. Indeed, let $\Pi \in \bot\Gamma^*$ with $symbol(\Pi) = a \lessdot x \gtrdot b$: an OPA $\mathcal{A}$ performs the computation $\langle \Pi,\ q,\ xb \rangle \vdash \langle \Pi,\ p,\ b \rangle$ without changing the portion Π of the stack, if and only if ${}^{a}[x]^{b}$ is a chain over (Σ, M) with a support $q \overset{x}{\rightsquigarrow} p$ in $\mathcal{A}$. The depth of x corresponds to the maximum number of push/pop pairs nested in the computation, i.e., the maximum height reached by the stack in one of the traversed configurations, minus the height of the stack in the starting configuration.

Notice that the context a, b of a chain is used by the automaton to build its support only because $a \lessdot x$ and $x \gtrdot b$; thus, the chain's body contains all information needed by the automaton to build the subtree whose frontier is that string, once it is understood that its first move is a push and its last one is pop. This is a distinguishing feature of OPLs, not shared by other deterministic languages: it is called the *locality principle* of OPLs, which is exploited, e.g., to build parallel (see Chapter 5) and/or incremental OP parsers.

With reference to Example 2.3 and Figure 9, the parsing of substring $n + n$ within the context $(\!|, |\!)$ is given by the computation

$$\langle \Pi,\ q_2,\ n+n\ |\!)\# \rangle \overset{*}{\vdash} \langle \Pi,\ q_3,\ |\!)\# \rangle \quad \text{with} \quad \Pi = \bot[+,\ q_1][\times,\ q_1][(\!|,\ q_0]$$

which corresponds to support $q_2 \overset{n}{\rightsquigarrow} q_3 \xrightarrow{+} q_2 \overset{n}{\rightsquigarrow} q_3 \overset{q_3}{\Longrightarrow} q_3$ of the composed chain ${}^{(\!|}[n+n]^{|\!)}$, where $q_2 \overset{n}{\rightsquigarrow} q_3$ is the support $q_2 \xrightarrow{n} q_3 \overset{q_2}{\Longrightarrow} q_3$ of the simple chains ${}^{(\!|}[n]^{+}$ and ${}^{+}[n]^{|\!)}$.

2.2.1 *Examples*

In this section we illustrate an example of application of OPLs, which cannot be modeled by traditional classes of languages with an "explicit" structure such as parenthesis languages and VPLs. We shall present in Chapter 3 examples in other interesting contexts (such as operating systems) which can be naturally modeled by OPAs recognizing ω-languages, and are not recognizable by VPAs as well. Other examples of application of OPLs to model systems in various application fields outside the traditional one of programming languages are given in [82].

Indeed, the most distinguishable feature of the structure of VPLs is that in their OPMs the $\doteq$ relation occurs always and only between open and closed parentheses (Σ_c and Σ_r elements in [10] notation, respectively). Unlike traditional parenthesis languages, however, in VPLs parentheses can remain unmatched, but only at the beginning (Σ_r elements) and end (Σ_c elements) of the input string, respectively. This initial extension, however, is not sufficiently general to cover several interesting cases where an "event" of special type, e.g. a rollback or an exception, should force flushing the stack of many pending elements, say write operations or procedure calls.

Example 2.5. OPAs can be used to model the run-time behavior of database systems, e.g., for modeling sequences of users' transactions with possible rollbacks. Other systems that exhibit an analogous behavior are revision control (or *versioning*) systems (such as subversion or git). As an example, consider a system for version management of files where a user can perform the following operations on documents: save them, access and modify them, undo one (or more) previous changes, restoring the previously saved version.

The following alphabet represents the user's actions: *sv* (for *save*), *wr* (for *write*, i.e. the document is opened and modified), *ud* (for a single *undo* operation), *rb* (for a *rollback* operation, where all the changes occurred since the previously saved version are discarded).

An OPA that models the traces of possible actions of the user on a given document is a single-state automaton $\langle \Sigma, M, \{q\}, \{q\}, \{q\}, \delta \rangle$, where $\Sigma = \{sv, rb, wr, ud\}$, M is:

$$M = \begin{array}{c|ccccc} & sv & rb & wr & ud & \# \\ \hline sv & \lessdot & \doteq & \lessdot & & \gtrdot \\ rb & \gtrdot & \gtrdot & \gtrdot & \gtrdot & \gtrdot \\ wr & \lessdot & \gtrdot & \lessdot & \doteq & \gtrdot \\ ud & \gtrdot & \gtrdot & \gtrdot & \gtrdot & \gtrdot \\ \# & \lessdot & & \lessdot & & \doteq \end{array}$$

and $\delta_{\text{push}}(q, a) = q, \forall a \in \{sv, wr\}$, $\delta_{\text{shift}}(q, a) = q, \forall a \in \{rb, ud\}$ and $\delta_{\text{pop}}(q, q) = q$. The precedence relations between the symbols are defined such that *sv* and *rb* represent a pair of matching open/closed parenthesis symbols, enforcing their occurrences in a word to be properly nested: for instance a computation such as *sv sv rb rb rb* is forbidden; analogous precedence relations hold also for the pair *wr* and *ud*. After a *sv* operation, a subsequent *wr* is recorded on the stack ($sv \lessdot wr$) and a *wr* is discarded and removed from the stack if an *ud* operation is performed ($wr \doteq ud, ud \gtrdot a$ for all $a \in \Sigma$). A user can save a document multiple times, possibly after writing it ($sv \lessdot sv$, $wr \lessdot sv$). A *rb* operation forces flushing the stack of all pending changes (wr, ud) until the last *sv* included ($wr \gtrdot rb$, $ud \gtrdot rb$, $sv \doteq rb$, $rb \gtrdot a$ for all $a \in \Sigma$). A user can perform an *ud* operation only if preceded by a *wr* ($M_{a,ud} \cap \{\lessdot, \doteq\} = \emptyset$ for all $a \neq wr$), which occurred after the last *sv* not discarded by a *rb*.

A more specialized model of this system might impose that the user regularly backs her work up, so that no more than N changes that are not undone (denoted *wr* as before) can occur between any two consecutive checkpoints *sv* (without any rollback *rb* between them). Figure 10 shows the corresponding OPA with $N = 2$, with the same OPM M.
States $0, 1$ and 2 denote respectively the presence of zero, one and two unmatched changes between two symbols *sv*.

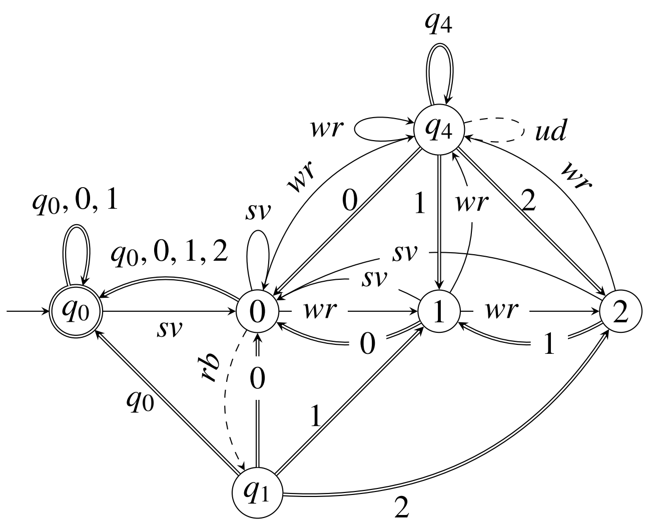


Figure 10.: OPA of Example 2.5, with $N = 2$.

An example of computation on the string *sv wr ud rb sv wr wr ud sv wr rb wr sv* is shown in Figure 11.

2.2.2 *Determinism vs nondeterminism*

An important property of OPAs is the equivalence between the deterministic and the nondeterministic version thereof. This result also implies the closure of OPLs under complementation, yielding an alternative proof to the traditional one presented in [35].

The deterministic version of OPAs is defined along the usual lines:

Definition 2.9 (Deterministic OPA)**.** An OPA is deterministic if I is a singleton, and the ranges of δ_{push}, δ_{shift} and δ_{pop} are Q rather than $\wp(Q)$.

It is well-known that the equivalence between nondeterministic and deterministic machines usually does not extend from finite state to pushdown ones. VPAs are however a noticeable exception. The construction described in [11] can be extended to

stack	state	current input
$\perp$	q_0	*sv wr ud rb sv wr wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0]$	0	*wr ud rb sv wr wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0]$	q_4	*ud rb sv wr wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0][ud,\ 0]$	q_4	*rb sv wr wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0]$	0	*rb sv wr wr ud sv wr rb wr sv #*
$\perp[rb,\ q_0]$	q_1	*sv wr wr ud sv wr rb wr sv #*
$\perp$	q_0	*sv wr wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0]$	0	*wr wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0]$	1	*wr ud sv wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0][wr,\ 1]$	q_4	*ud sv wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0][ud,\ 1]$	q_4	*sv wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0]$	1	*sv wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0][sv,\ 1]$	0	*wr rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0][sv,\ 1][wr,\ 0]$	1	*rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0][sv,\ 1]$	0	*rb wr sv #*
$\perp[sv,\ q_0][wr,\ 0][rb,\ 1]$	q_1	*wr sv #*
$\perp[sv,\ q_0][wr,\ 0]$	1	*wr sv #*
$\perp[sv,\ q_0][wr,\ 0][wr,\ 1]$	2	*sv #*
$\perp[sv,\ q_0][wr,\ 0][wr,\ 1][sv,\ 2]$	0	#
$\perp[sv,\ q_0][wr,\ 0][wr,\ 1]$	q_0	#
$\perp[sv,\ q_0][wr,\ 0]$	q_0	#
$\perp[sv,\ q_0]$	q_0	#
$\perp$	q_0	#

Figure 11.: Example of computation for the specialized system of Example 2.5

cover OPAs too. The construction for OPAs ensures that two different pop moves of two different runs of the nondeterministic automaton never "mix up" their initial and final states in the deterministic one by keeping track of the path of the automaton since the push move that marks the origin of the chain to be reduced by the next pop move. Precisely, the states of the deterministic automaton $\tilde{\mathcal{A}}$ are sets of pairs of states, instead of sets of single states, of the nondeterministic automaton $\mathcal{A}$: $\tilde{\mathcal{A}}$ simulates $\mathcal{A}$ along the first component of the pair, whereas the second component stores the state that gave origin to a push transition and it is propagated through shift moves. The deterministic pop operations will simulate only the nondeterministic ones defined on the states corresponding to the first component of the current state and the state reached before the last push move, which corresponds to the state on the top of the stack in an actual run of the nondeterministic automaton.

The following theorem formalizes the above informal reasoning.

Theorem 2.1 ([69]). *Given a nondeterministic OPA $\mathcal{A}$ with s states, an equivalent deterministic OPA $\tilde{\mathcal{A}}$ can effectively be built with $2^{O(s^2)}$ states.*

Proof. Let $\mathcal{A}$ be $\langle \Sigma, M, Q, I, F, \delta \rangle$; $\tilde{\mathcal{A}} = \langle \Sigma, M, \tilde{Q}, \tilde{I}, \tilde{F}, \tilde{\delta} \rangle$ is defined as follows:

- $\tilde{Q} = \wp(Q \times (Q \cup \{\top\}))$, where $Q \cap \{\top\} = \emptyset$ and $\top$ is a symbol that stands for the baseline of the computations; we will use $K, K_i, \bar{K}, K', \ldots$ to denote states in $\tilde{Q}$,
- $\tilde{I} = I \times \{\top\}$ is the initial state of $\tilde{\mathcal{A}}$,
- $\tilde{F} = \{K \mid K \cap (F \times \{\top\}) \neq \emptyset\}$,
- $\tilde{\delta} : \tilde{Q} \times (\Sigma \cup \tilde{Q}) \to \tilde{Q}$ is the transition function defined as follows.

  The push transition $\tilde{\delta}_{\text{push}} : \tilde{Q} \times \Sigma \to \tilde{Q}$ is defined by

  $$\tilde{\delta}_{\text{push}}(K, a) = \bigcup_{(q,p) \in K} \Big\{(h, q) \mid h \in \delta_{\text{push}}(q, a)\Big\}$$

  The shift transition $\tilde{\delta}_{\text{shift}} : \tilde{Q} \times \Sigma \to \tilde{Q}$ is defined by

  $$\tilde{\delta}_{\text{shift}}(K, a) = \bigcup_{(q,p) \in K} \{(h, p) \mid h \in \delta_{\text{shift}}(q, a)\}$$

  The pop transition $\tilde{\delta}_{\text{pop}} : \tilde{Q} \times \tilde{Q} \to \tilde{Q}$ is defined as follows:

  $$\tilde{\delta}_{\text{pop}}(K_1, K_2) = \bigcup_{(r,q) \in K_1, (q,p) \in K_2} \Big\{(h, p) \mid h \in \delta_{\text{pop}}(r, q)\Big\}.$$

Notice that, if $|Q| = s$ is the number of states of the nondeterministic OPA $\mathcal{A}$, the deterministic OPA $\tilde{\mathcal{A}}$ that is obtained in this way has a set of states whose size is exponential in s^2, i.e. $|\tilde{Q}| = 2^{|Q|\cdot|Q\cup\{\perp\}|}$ which is $2^{O(s^2)}$. Also, this bound is asymptotically optimal (the reasoning to prove succinctness of nondeterminism for VPAs discussed in [11] can be applied also to OPAs).

The proof of equivalence between the two automata is by induction and is based on lemmata 2.1 and 2.2.

Lemma 2.1. *Let y be the body of a chain with support $q \overset{y}{\leadsto} q'$ in $\mathcal{A}$. Then, for every $p \in Q$ and $K \in \tilde{Q}$, if $K \ni (q, p)$, there exists a support $K \overset{y}{\leadsto} K'$ in $\tilde{\mathcal{A}}$ with $K' \ni (q', p)$.*

Proof. We argue by induction on the depth h of y. If $h = 1$ then $y = a_1 a_2 \ldots a_n$ and the support can be rewritten as in (1) with $q_0 = q$ and $q_{n+1} = q'$. Set $K_0 = K$ and

$$\begin{aligned} K_1 &= \tilde{\delta}_{\text{push}}(K_0, a_1) \\ K_i &= \tilde{\delta}_{\text{shift}}(K_{i-1}, a_i), \text{ for every } i = 2, \ldots, n \\ K' &= \tilde{\delta}_{\text{pop}}(K_n, K) \end{aligned}$$

Then

$$K \xrightarrow{a_1} K_1 \overset{a_2}{\dashrightarrow} \ldots \overset{a_{n-1}}{\dashrightarrow} K_{n-1} \overset{a_n}{\dashrightarrow} K_n \overset{K}{\Longrightarrow} K' \tag{3}$$

is a support for C in $\tilde{\mathcal{A}}$. Moreover, since $K \ni (q, p)$, by the definition of $\tilde{\delta}$ we have:

$$\begin{aligned} &K_1 \ni (q_1, q) && \text{since } \delta_{\text{push}}(q, a_1) \ni q_1, \\ &K_i \ni (q_i, q) && \text{since } \delta_{\text{shift}}(q_{i-1}, a_i) \ni q_i, \\ &K' \ni (q', p) && \text{since } \delta_{\text{pop}}(q_n, q) \ni q'. \end{aligned}$$

Now assume that the statement holds for supports with depth lower than h and let $y = x_0 a_1 x_1 a_2 \ldots a_n x_n$ have depth h. The support can be rewritten as in (2) with $q_0 = q$ and $q_{n+1} = q'$, where $q'_i = q_i$ whenever x_i is the empty word, and every non-empty x_i has depth lower than h.

Then, by the inductive hypothesis and the definition of $\tilde{\delta}$, we can build a support

$$K \overset{x_0}{\leadsto} K'_0 \xrightarrow{a_1} K_1 \overset{x_1}{\leadsto} K'_1 \overset{a_2}{\dashrightarrow} \ldots \overset{a_n}{\dashrightarrow} K_n \overset{x_n}{\leadsto} K'_n \overset{K'_0}{\Longrightarrow} K' \tag{4}$$

where, (q, p) being in K, we have:

$$\begin{array}{ll}
K'_0 \ni (q'_0, p) & \text{by inductive hypothesis on the support } q = q_0 \overset{x_0}{\rightsquigarrow} q'_0, \\
K_1 \ni (q_1, q'_0) & \text{since } \delta_{\text{push}}(q'_0, a_1) \ni q_1, \\
K'_1 \ni (q'_1, q'_0) & \text{by inductive hypothesis on the support } q_1 \overset{x_1}{\rightsquigarrow} q'_1, \\
K_i \ni (q_i, q'_0) & \text{since } \delta_{\text{shift}}(q'_{i-1}, a_i) \ni q_i, \text{ for every } i = 2, \dots, n, \\
K'_i \ni (q'_i, q'_0) & \text{by inductive hypothesis on the support } q_i \overset{x_i}{\rightsquigarrow} q'_i, \\
K' \ni (q', p) & \text{since } \delta_{\text{pop}}(q_n, q'_0) \ni q',
\end{array}$$

and this concludes the proof. □

Lemma 2.2. *Let y be the body of a chain with support $K \overset{y}{\rightsquigarrow} K'$ in $\tilde{\mathcal{A}}$. Then, for every $p, q' \in Q$, if $K' \ni (q', p)$ there exists a support $q \overset{y}{\rightsquigarrow} q'$ in $\mathcal{A}$ with $(q, p) \in K$.*

Proof. First we present some remarks we will use in the proof.

i) By the definition of δ_{push}, if $\bar{K} \xrightarrow{a} K$ in $\tilde{\mathcal{A}}$, $(\bar{q}, q) \in K$, $(q, p) \in \bar{K}$, then $q \xrightarrow{a} \bar{q}$ in $\mathcal{A}$.

ii) By the definition of δ_{shift}, if $\bar{K} \overset{a}{\dashrightarrow} K$ in $\tilde{\mathcal{A}}$, $(r, q) \in K$, then there exists a state $\bar{q} \in Q$ such that $\bar{q} \overset{a}{\dashrightarrow} r$ in $\mathcal{A}$ and $(\bar{q}, q) \in \bar{K}$.

iii) By the definition of δ_{pop}, if $\bar{K} \overset{K}{\Longrightarrow} K'$ in $\tilde{\mathcal{A}}$ and $(q', p) \in K'$, then there exists a pair $(r, q) \in \bar{K}$ such that $(q, p) \in K$ and $r \overset{q}{\Longrightarrow} q'$ in $\mathcal{A}$.

We argue by induction on the depth h of y. If $h = 1$, then $y = a_1 a_2 \dots a_n$ and the support can be rewritten as in (3). Let $K' \ni (q', p)$; then, by Remark (iii) there exists a pair $(q_n, q) \in K_n$ such that $(q, p) \in K$ and $q_n \overset{q}{\Longrightarrow} q'$ in $\tilde{\mathcal{A}}$. Moreover, $(q_n, q) \in K_n$ and $K_{n-1} \overset{a_n}{\dashrightarrow} K_n$ imply by Remark (ii) the existence of a state $q_{n-1} \in Q$ such that $(q_{n-1}, q) \in K_{n-1}$ and $q_{n-1} \overset{a_n}{\dashrightarrow} q_n$. Similarly one can verify that for every $i = n-2, \dots 1$ there exists $q_i \in Q$ such that $(q_i, q) \in K_i$ and $q_i \overset{a_{i+1}}{\dashrightarrow} q_{i+1}$. Finally, $K \xrightarrow{a_1} K_1$, $(q_1, q) \in K_1$ and $(q, p) \in K$ imply by Remark (i) that $q \xrightarrow{a_1} q_1$ in $\mathcal{A}$. Thus, we built backward a path as in (1) with $q_0 = q$, $q_{n+1} = q'$, $(q, p) \in K$, and this concludes the proof of induction basis.

Now assume that the statement holds for chains with depth lower than h. Let $y = x_0a_1x_1a_2\dots a_nx_n$ have depth h and consider a support as in (4) where $K'_i = K_i$ whenever x_i is the empty word, and every non-empty x_i has depth lower than h. Let $(q', p) \in K'$. Since $K'_n \overset{K'_0}{\Longrightarrow} K'$, by Remark (iii) there exists a pair $(q'_n, q'_0) \in K'_n$ with $(q'_0, p) \in K'_0$ and $q'_n \overset{q'_0}{\Longrightarrow} q'$ in $\tilde{\mathcal{A}}$. If $x_n \neq \varepsilon$, by the inductive hypothesis, since $(q'_n, q'_0) \in K'_n$ there exists a support $q_n \overset{x_n}{\rightsquigarrow} q'_n$ with $(q_n, q'_0) \in K_n$.

Similarly one can see that, for all $i = n-1, \dots 2, 1$, there exist q'_i and q_i ($q'_i = q_i$ whenever x_i is empty) such that

$$q_i \overset{x_i}{\rightsquigarrow} q'_i \overset{a_{i+1}}{\dashrightarrow} q_{i+1}$$

with $(q'_i, q'_0) \in K'_i$ by Remark (ii) (since $K'_i \overset{a_{i+1}}{\dashrightarrow} K_{i+1}$ in $\tilde{\mathcal{A}}$ and $(q_{i+1}, q'_0) \in K_{i+1}$), and $(q_i, q'_0) \in K_i$ by the inductive hypothesis (since $K_i \overset{x_i}{\rightsquigarrow} K'_i$ in $\tilde{\mathcal{A}}$ and $(q'_i, q'_0) \in K'_i$).

In particular $q_1 \overset{x_1}{\rightsquigarrow} q'_1$ with $(q_1, q'_0) \in K_1$. Then, since also $K'_0 \overset{a_1}{\longrightarrow} K_1$ and $(q'_0, p) \in K'_0$, by Remark (i) we get $q'_0 \overset{a_1}{\longrightarrow} q_1$. Finally, since $(q'_0, p) \in K'_0$ and $K \overset{x_0}{\rightsquigarrow} K'_0$, if $x_0 \neq \varepsilon$ the inductive hypothesis implies the existence of a state $q \in Q$ such that $q \overset{x_0}{\rightsquigarrow} q'_0$ in $\tilde{\mathcal{A}}$ with $(q, p) \in K$. Hence we built a support as in (2) with $q_0 = q$, $q_{n+1} = q'$ and $(q, p) \in K$, and this concludes the proof. □

To complete the proof of Theorem 2.1, we prove that there exists an accepting computation for y in $\mathcal{A}$ if and only if there exists an accepting computation for y in $\tilde{\mathcal{A}}$.

Let y be in $L(\mathcal{A})$. Then it admits a support $q \overset{y}{\rightsquigarrow} q'$ with $q \in I$ and $q' \in F$. Then for $K = I \times \{\top\} \ni (q_0, \top)$, Lemma 2.1 implies the existence of a support $K \overset{y}{\rightsquigarrow} K'$ in $\tilde{\mathcal{A}}$ with $K' \ni (q', \top)$. $q' \in F$ implies $K' \in \tilde{F}$, hence y is accepted by $\tilde{\mathcal{A}}$.

Conversely, let y be in $L(\tilde{\mathcal{A}})$. Then y admits a support $\tilde{K} \overset{y}{\rightsquigarrow} K'$ in $\tilde{\mathcal{A}}$, with $K' \in \tilde{F}$. This means that there exists $q' \in F$ such that $(q', \top) \in K'$. Hence, by Lemma 2.2, there exists a support $q \overset{y}{\rightsquigarrow} q'$ in $\mathcal{A}$ with $(q', \top) \in \tilde{K}$, and this implies $q \in I$. Thus the support $q \overset{y}{\rightsquigarrow} q'$ defines an accepting computation for y in $\mathcal{A}$. □

2.2.3 *Complexity of OPL decision problems*

The basic decision problems for OPLs have the same order of complexity as those for VPLs; precisely:

- the emptiness problem is in PTIME, OPLs and VPLs being a subclass of context-free languages;
- the containment problem for deterministic OPAs is in PTIME too since it is reduced to the intersection, complement and emptiness problems which are all in PTIME in the deterministic case;
- the containment problem in the nondeterministic case is instead EXPTIME-complete: the same arguments used in [11] for VPLs apply identically for OPLs.

2.3 EQUIVALENCE BETWEEN OPERATOR PRECEDENCE GRAMMARS AND AUTOMATA

The next results show the equivalence between OPGs and OPAs.

2.3.1 *From OPGs to OPAs*

Theorem 2.2. *Let $G = \langle N, \Sigma, P, S \rangle$ be an OPG; then an OPA $\mathcal{A}$ such that $L(\mathcal{A}) = L(G)$ can effectively be built. Furthermore, let m be the sum of the lengths of the r.h.s.s of G; then $\mathcal{A}$ has $O(m^2)$ states.*

Proof. First, we describe a procedure to build a nondeterministic OPA $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ from a given OPG G with the same precedence matrix M as G. Then we prove the equivalence between $\mathcal{A}$ and G.

The construction sharply differs from the traditional one involving CF grammars and general pushdown automata, which is instead quite straightforward. This is due to the remarkable peculiarities of OPAs –among them the locality principle– which make them, in turn, significantly different from the more powerful general pushdown automata and from the less powerful VPAs. To keep the construction as simple as possible, we avoid introducing any optimization. Also, without loss of generality, we assume that the grammar G has no empty nor renaming rules.

$\mathcal{A}$ is built in such a way that a successful computation thereof corresponds to building bottom-up a derivation tree in G: the automaton performs a push transition when it reads the first terminal of a new r.h.s. It performs a shift transition when it reads a terminal symbol inside a r.h.s., i.e., a leaf with some left sibling leaf. It performs a pop transition when it completes the recognition of a r.h.s., then guesses (nondeterministically) the nonterminal at the l.h.s. Each state contains two pieces of information: the first component represents the prefix of the r.h.s. under construction, whereas the second component is used to recover the r.h.s. *previously under construction* (see Figure 12) whenever all r.h.s.s nested below have been completed.

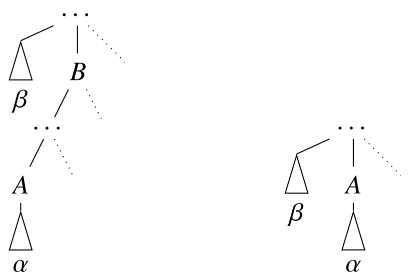


Figure 12.: When parsing α, the prefix previously under construction is β.

Precisely, the construction of $\mathcal{A}$ is defined as follows. Let

$$\mathbb{P} = \{\alpha \in (N \cup \Sigma)^*\Sigma \mid \exists A \to \alpha\beta \in P\}$$

be the set of prefixes, ending with a terminal symbol, of r.h.s. of G; define $\mathbb{Q} = \{\varepsilon\} \cup \mathbb{P} \cup N$, $Q = \mathbb{Q} \times (\{\varepsilon\} \cup \mathbb{P})$, $I = \{\langle\varepsilon,\varepsilon\rangle\}$, and $F = S \times \{\varepsilon\} \cup \{\langle\varepsilon,\varepsilon\rangle \mid \varepsilon \in L(G)\}$. Note that $|\mathbb{Q}| = 1 + |\mathbb{P}| + |N|$ is $O(m)$; therefore $|Q|$ is $O(m^2)$.

The transition functions are defined as follows, for $a \in \Sigma$ and $\alpha, \alpha_1, \alpha_2 \in \mathbb{Q}$, $\beta, \beta_1, \beta_2 \in \{\varepsilon\} \cup \mathbb{P}$:

- $\delta_{\text{shift}}(\langle\alpha,\beta\rangle, a) \ni \begin{cases} \langle\alpha a,\beta\rangle & \text{if } \alpha \notin N \\ \langle\beta\alpha a,\beta\rangle & \text{if } \alpha \in N \end{cases}$

- $\delta_{\text{push}}(\langle\alpha,\beta\rangle, a) \ni \begin{cases} \langle a,\alpha\rangle & \text{if } \alpha \notin N \\ \langle\alpha a,\beta\rangle & \text{if } \alpha \in N \end{cases}$

- $\delta_{\text{pop}}(\langle\alpha_1,\beta_1\rangle, \langle\alpha_2,\beta_2\rangle) \ni \langle A,\gamma\rangle$ for every A such that $\left[\begin{array}{ll} A \to \alpha_1 \in P, & \text{if } \alpha_1 \notin N \\ A \to \beta_1\alpha_1 \in P, & \text{if } \alpha_1 \in N \end{array}\right.$

  and $\gamma = \begin{cases} \alpha_2, & \text{if } \alpha_2 \notin N \\ \beta_2, & \text{if } \alpha_2 \in N. \end{cases}$

Notice that the result of δ_{shift} and δ_{push} is a singleton, whereas δ_{pop} may produce several states, in case of repeated r.h.s.s.

The states reached by push and shift transitions have the first component in $\mathbb{P}$. If state $\langle\alpha,\beta\rangle$ is reached after a push transition, then α is the prefix of the r.h.s. that is currently under construction and β is the prefix previously under construction; in this case α is either a terminal symbol or a nonterminal followed by a terminal one. If the state is reached after a shift transition, then α is the concatenation of the first component of the previous state with the read character, and β is not changed from the previous state. The states reached by a pop transition have the first component in N: if $\langle A,\gamma\rangle$ is such a state, then A is the corresponding l.h.s, and γ is the prefix previously under construction.

The equivalence between G and $\mathcal{A}$ derives from the following Lemmata 2.3 and 2.4, when $\beta = \gamma = \varepsilon$, $\Pi = \perp$ and A is an axiom. ◻

Example 2.6. Let G be the grammar introduced in Example 2.1. To apply the construction of Theorem 2.2 first we need to transform G in such a way that there are no renaming rules. The new grammar has the following productions

$$\begin{array}{lcl} E & \to & E+T \mid T\times F \mid n \mid (\!|E|\!) \\ T & \to & T\times F \mid n \mid (\!|E|\!) \\ F & \to & n \mid (\!|E|\!) \end{array}$$

where E, T, and F are axioms.

Figure 13 shows an accepting computation of the equivalent automaton, together with the corresponding derivation tree. Notice that the computation shown in Figure 13 is equal to that of Figure 8 up to a renaming of the states; in fact the shape of syntax trees and consequently the sequence of push, shift and pop moves in OPLs depends only on the OPM, not on the visited states.

Lemma 2.3. *Let x be the body of a chain and $\beta, \gamma \in \mathbb{P} \cup \{\varepsilon\}$. Then, for all $h \geq 1$, $\langle \beta, \gamma \rangle \overset{x}{\leadsto} q$ implies the existence of $A \in N$ such that $A \overset{*}{\Rightarrow} x$ in G and $q = \langle A, \beta \rangle$.*

Proof. We reason by induction on the depth h of x.

If $h = 1$, then $x = a_1 a_2 \dots a_n$ is the body of a simple chain, and the support is as in (1) with $q_0 = \langle \beta, \gamma \rangle$ and $q_{n+1} = q$. Then by the definition of push and shift transition functions we have $q_i = \langle a_1 \dots a_i, \beta \rangle$ for every $i = 1, 2, \dots n$, and by the definition of pop transition function (recall that $\beta \notin N$ by hypothesis) it is $q = \langle A, \beta \rangle$ for some A such that $A \to a_1 \dots a_n = x$ is in P. Hence $A \overset{*}{\Rightarrow} x$ and the statement is proved.

If $h > 1$, then as usual let $x = x_0 a_1 x_1 \dots a_n x_n$ and let its support be decomposed as in (2) with $q_0 = \langle \beta, \gamma \rangle$ and $q_{n+1} = q$. Also set $q_i = \langle \beta_i, \gamma_i \rangle$ for $i = 0, 1, \dots, n$ (in particular $\beta_0 = \beta$ and $\gamma_0 = \gamma$). Each non empty x_i being the body of a chain with depth lower than h, the inductive hypothesis implies that there exists $X_i \in N$ such that $X_i \overset{*}{\Rightarrow} x_i$ in G, and $q_i = \langle X_i, \beta_i \rangle$. Thus, the support can be rewritten as

$$\langle \beta, \gamma \rangle \overset{x_0}{\rightsquigarrow} q_0' \xrightarrow{a_1} \langle \beta_1, \gamma_1 \rangle \overset{x_1}{\rightsquigarrow} q_1' \overset{a_2}{\dashrightarrow} \dots \overset{a_n}{\dashrightarrow} \langle \beta_n, \gamma_n \rangle \overset{x_n}{\rightsquigarrow} q_n' \overset{q_0'}{\Longrightarrow} q$$

stack	state	current input
$\perp$	$\langle \varepsilon, \varepsilon \rangle$	$n + n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[n,\ \langle \varepsilon, \varepsilon \rangle]$	$\langle n, \varepsilon \rangle$	$+n \times \llparenthesis n + n \rrparenthesis \#$
$\perp$	$\langle E, \varepsilon \rangle$	$+n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle]$	$\langle E+, \varepsilon \rangle$	$n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][n,\ \langle E+, \varepsilon \rangle]$	$\langle n, \varepsilon \rangle$	$\times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle]$	$\langle T, E+ \rangle$	$\times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle]$	$\langle T\times, E+ \rangle$	$\llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle]$	$\langle \llparenthesis, E+ \rangle$	$n + n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle][n,\ \langle \llparenthesis, E+ \rangle]$	$\langle n, E+ \rangle$	$+n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle]$	$\langle E, \llparenthesis \rangle$	$+n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle][+,\ \langle E, \llparenthesis \rangle]$	$\langle E+, \llparenthesis \rangle$	$n \rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle][+,\ \langle E, \llparenthesis \rangle][n,\ \langle E+, \llparenthesis \rangle]$	$\langle n, \llparenthesis \rangle$	$\rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle][+,\ \langle E, \llparenthesis \rangle]$	$\langle T, E+ \rangle$	$\rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\llparenthesis,\ \langle T\times, E+ \rangle]$	$\langle E, \llparenthesis \rangle$	$\rrparenthesis \#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle][\rrparenthesis,\ \langle T\times, E+ \rangle]$	$\langle \llparenthesis E \rrparenthesis, \llparenthesis \rangle$	$\#$
$\perp[+,\ \langle E, \varepsilon \rangle][\times,\ \langle T, E+ \rangle]$	$\langle F, T\times \rangle$	$\#$
$\perp[+,\ \langle E, \varepsilon \rangle]$	$\langle T, E+ \rangle$	$\#$
$\perp$	$\langle E, \varepsilon \rangle$	$\#$

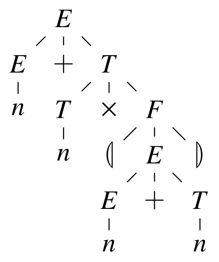


Figure 13.: Accepting computation of the automaton built in Theorem 2.2.

where

$$q'_i = \begin{cases} \langle \beta_i, \gamma_i \rangle & \text{if } x_i = \varepsilon \\ \langle X_i, \beta_i \rangle & \text{otherwise} \end{cases}$$

for every i. Now, by the definition of push and shift transition functions, one can see that, for $i \neq 0$, $\beta_i = X_0 a_1 \dots X_{i-1} a_i$ holds regardless of whether x_i is empty or not (setting $X_i = \varepsilon$ if $x_i = \varepsilon$). Thus, to compute the state q reached with the final pop transition $\delta_{\text{pop}}(q'_n, q'_0)$, we have to consider four cases depending on whether x_0 and x_n are empty or not, which are exactly the four combinations considered in the definition of δ_{pop}. In any case, q has the form $\langle A, \beta \rangle$, where A is a nonterminal of G such that $A \to X_0 a_1 X_1 \dots X_{n_1} a_n X_n$. □

Lemma 2.4. *Let x be the body of a chain and $A \in N$. Then, $A \overset{*}{\Rightarrow} x$ in G implies $\langle \beta, \gamma \rangle \overset{x}{\rightsquigarrow} \langle A, \beta \rangle$ for every $\beta, \gamma \in \mathbb{P} \cup \{\varepsilon\}$.*

Proof. We reason by induction on the depth h of the chain. If $h = 1$, then x is the body of a simple chain, hence $A \overset{*}{\Rightarrow} x$ means that $A \to x$ is a production. Thus, by the definition of δ (recall that $\beta \notin N$ by hypothesis), we obtain a support as in (1) with $q_0 = \langle \beta, \gamma \rangle$, $q_{n+1} = q$, and $q_i = \langle a_1 \dots a_i, \beta \rangle$ for every $i = 1, 2, \dots n$.

If $h > 1$, then x is the body of a composed chain with $x = x_0 a_1 x_1 \dots a_n x_n$. Hence $A \overset{*}{\Rightarrow} x$ in G implies that there exist $X_0, X_1, \dots, X_n \in \{\varepsilon\} \cup N$ (more precisely: $X_i = \varepsilon$ if $x_i = \varepsilon$) such that $A \to X_0 a_1 X_1 \dots a_n X_n$ and $X_i \overset{*}{\Rightarrow} x_i$. The first step of the computation is different depending on whether x_0 is empty or not. In any case, we have

$$\langle \beta, \gamma \rangle \overset{x_0}{\rightsquigarrow} q'_0 \xrightarrow{a_1} \langle X_0 a_1, \beta \rangle, \qquad \text{where} \qquad q'_0 = \begin{cases} \langle \beta, \gamma \rangle & \text{if } x_0 = \varepsilon \\ \langle X_0, \beta \rangle & \text{otherwise} \end{cases}$$

The computation goes on differently depending on whether $x_1, x_2, \dots, x_{n-1}$ are empty or not. However, by the inductive hypothesis and the definition of δ_{shift}, after reading

a_i the automaton reaches state $\langle X_0 a_1 \dots X_{i-1} a_i, \beta\rangle$ for every $i = 1, \dots, n$, i.e., we have the path

$$\langle \beta, \gamma\rangle \overset{x_0}{\leadsto} q_0' \xrightarrow{a_1} \langle X_0 a_1, \beta\rangle \overset{x_1}{\leadsto} q_1' \xrightarrow{a_2} \langle X_0 a_1 X_1 a_2, \beta\rangle \overset{x_2}{\leadsto} q_2' \xrightarrow{a_3} \dots$$
$$\xrightarrow{a_n} \langle X_0 a_1 \dots X_{n-1} a_n, \beta\rangle.$$

If $x_n \neq \varepsilon$, the computation proceeds with the last inductive step

$$\langle X_0 a_1 \dots X_{n-1} a_n \ , \ \beta_n\rangle \overset{x_n}{\leadsto} \langle X_n \ , \ X_0 a_1 \dots X_{n-1} a_n\rangle.$$

Finally, the computation ends with a pop transition. There are four cases depending on whether x_0 and x_n are empty or not, which are exactly the four combinations considered in the definition of δ_{pop}. In any case, we build a support ending with state $\langle A, \beta\rangle$, and this concludes the proof. □

Corollary 2.1. *If the source grammar is in FNF, then the corresponding automaton is deterministic.*

The thesis follows immediately by observing that the construction defined in Theorem 2.2 is such that the values defined by δ_{push} and δ_{shift} are always singleton, whereas δ_{pop} produces as many states as many l.h.s.s have the same r.h.s. Thus, since the initial state is a singleton and grammars in FNF have no repeated r.h.s.s, the automaton resulting from the construction is already deterministic. This corollary has an interesting effect in terms of size of the produced automata as pointed out below.

Remark 2.1. Given a grammar G with $|N|$ nonterminals the construction of Theorem 2.2 produces an automaton with $O(m^2)$ states, where m is defined as Theorem 2.2; thus, if we build a deterministic OPA from a generic OPG G by first building a nondeterministic automaton and then transforming it in deterministic version, we obtain an automaton with $2^{O(m^4)}$ states; instead, if we first transform the original G in FNF we obtain an equivalent grammar $\tilde{G}$ with $O(2^{|N|})$ nonterminals and $\tilde{m} = O(2^{m|N|^2})$; then, by applying the construction of Theorem 2.2 we directly obtain a deterministic automaton with $O(\tilde{m}^2) = O(2^{2m|N|^2})$ states.

The size of the complete automaton is clearly hardly manageable by human execution, but a prototype (non-optimized) tool is available to perform the construction[4].

2.3.2 *From OPAs to OPGs*

The construction of an OPG equivalent to a given OPA is far simpler than the converse one, thanks to the explicit structure associated to words by the precedence matrix.

Theorem 2.3 ([70])**.** *Let $\mathcal{A}$ be an OPA; then an OPG G such that $L(G) = L(\mathcal{A})$ can effectively be built.*

Proof. Given an OPA $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, an equivalent OPG G having operator precedence matrix M can be built as follows.

G's nonterminals are the 4-tuples $(a, q, p, b) \in \Sigma \times Q \times Q \times \Sigma$, written as $\langle {}^{a}p, q^{b} \rangle$. G's rules are built as follows:

- for every support of type (1) of a simple chain, the rule

$$\langle {}^{a_0}q_0, {q_{n+1}}^{a_{n+1}} \rangle \longrightarrow a_1 a_2 \ldots a_n \ ;$$

  is in P; furthermore, if $a_0 = a_{n+1} = \#$, q_0 is initial, and q_{n+1} is final, then $\langle {}^{\#}q_0, {q_{n+1}}^{\#} \rangle$ is in S;

- for every support of type (2) of a composed chain, add the rule

$$\langle {}^{a_0}q_0, {q_{n+1}}^{a_{n+1}} \rangle \longrightarrow \Lambda_0 a_1 \Lambda_1 a_2 \ldots a_n \Lambda_n \ ;$$

  where, for every $i = 0, 1, \ldots, n$, $\Lambda_i = \langle {}^{a_i}q_i, {q'_i}^{a_{i+1}} \rangle$ if $x_i \neq \varepsilon$ and $\Lambda_i = \varepsilon$ otherwise; furthermore, if $a_0 = a_{n+1} = \#$, q_0 is initial, and q_{n+1} is final, then add $\langle {}^{\#}q_0, {q_{n+1}}^{\#} \rangle$ to S, and, if ε is accepted by $\mathcal{A}$, add $A \to \varepsilon$, A being a new axiom not otherwise occurring in any other rule.

4 The tool is called *Flup*, available at [1].

Notice that the above construction is effective thanks to the hypothesis of $\doteq$-acyclicity of the OPM (remind that, as discussed in Section 2.1, this hypothesis could be replaced by weaker ones). This implies that the length of the r.h.s. is bounded; on the other hand, the cardinality of the nonterminal alphabet is finite (precisely it is $O(|\Sigma|^2 \cdot |Q|^2)$. Hence there is only a finite number of possible productions for G and only a limited number of chains to be considered. □

2.4 MONADIC SECOND-ORDER LOGIC CHARACTERIZATION OF OPLS

In his seminal paper [24] Büchi provided a logic characterization of regular languages: he defined a MSO syntax on the integers representing the position of characters within a string and he gave algorithms to build a finite state machine (FSM) recognizing exactly the strings satisfying a given formula and, conversely, to build a formula satisfied by all and only the strings accepted by a given FSM. Subsequently, a rich literature considerably extended his work to more powerful language families –typically, context-free [25]– and different logic formalisms, e.g., first-order or tree logics [5, 20, 29]. To the best of our knowledge, however, MSO logic characterizations of CF languages refer to "visible structure languages", i.e., to languages whose strings make their syntactic structure immediately visible in their external appearance, such as "tree-languages" [92][5] and Visibly Pushdown Languages [10] which explicitly refer to this peculiar property in their name. In this section we report a result of [71] that provides a complete MSO logic characterization of OPLs, which, instead, include also invisible-structure languages, whose syntax trees associated with external strings must be built by means of suitable parsing algorithms, in which the OPM plays a major role.

The MSO logic characterization of OPLs shows some similarities with the approach followed for VPLs, which has been presented in [11]. In this work they introduce a MSO logic, which exactly defines the class of VPLs, whose syntax consists of unary predicates over positions, first and second order quantifiers, the successor relation and a further, suitable binary predicate $\leadsto$ on string positions. Given an al-

5 It is not coincidence if tree automata [92] have been defined by extending the original finite state ones.

phabet Σ (partitioned into call, return and internal symbols), a countable set FV of first-order variables, a countable set SV of monadic second-order (set) variables and denoting by $x, y, x', \ldots$ elements in FV and by $X, Y, X', \ldots$ elements of SV, the MSO logic syntax is:

$$\varphi := c(x) \mid X(x) \mid call(x) \mid ret(x) \mid x = y + 1 \mid x \leadsto y \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

where c is a symbol in Σ, $x, y \in FV$ and $X \in SV$. The semantics is defined over a word so that the first-order variables are interpreted over positions of the string, while set variables are interpreted over sets of positions. The unary predicate $c(x)$ holds if the symbol at the position interpreted for x is c, and the two other unary predicates $call(x)$ and $ret(x)$ hold if the position interpreted for x is respectively a call or return symbol of the alphabet. $x = y + 1$ holds true as usual if x and y are interpreted as successive positions. As regards the binary predicate $\leadsto$, intuitively, if one interprets call and return symbols of the alphabet as respectively open and closed parentheses, the $\leadsto$ relation holds true on two positions either if they correspond to a pair of matching open and closed parentheses, or if the first position corresponds to an open parenthesis that is never matched in the string by a corresponding closed one and the second one is conventionally denoted by ∞, or if the second position corresponds to a closed parenthesis that was not preceded by a corresponding matching open one and the first position is conventionally denoted by ∞. As an example, the formula $\forall x.(call(x) \Rightarrow \exists x.x \leadsto y)$, where the quantifier $\forall$ and the implication $\Rightarrow$ are defined in the usual way in terms of the other basic syntax elements, holds in a word iff it has no unmatched calls symbols.

Analogously to VPLs, which in fact are a subclass of OPLs, the MSO logic characterization of OPLs is based on the definition of a suitable binary predicate on the string positions. The original definition of the $\leadsto$ relation in [11], however, cannot be naturally extended to the more general case of OPL strings. In fact the $\leadsto$ relation between two matching parentheses, which are extremes of the frontier of a sub-tree, is typically one-to-one (with the exclusion of the particular case of unmatched parentheses) whereas in general the relation between leftmost and rightmost leaves of an OPL sub-tree can be many-to-one or one-to-many or both. A further consequence of the more general structure of OPL trees is that, unlike FSMs, tree automata, and Visibly Pushdown Automata (for which every input symbol induces exactly a move, that is push, pop or neutral, depending on the class of the symbol itself), OPAs are not real-time automata as they may have to perform a series of pop moves without advancing their running head; this in turn produces the effect that, whereas in regular and VPLs each position is associated with a unique state visited by the machine during its behavior, for OPLs the same position may refer to several states –i.e., to several subsets of positions according to Büchi's approach.

Consequently, the approach to characterize OPLs departs from previous ones along two main directions:

- The binary relations between positions referring to a pop operation are attached to the look-back and look-ahead positions which in OP parsing embrace the r.h.s. to be reduced; thus, the formal definition of the relation will be based on the notion of chain.
- The sets of positions associated with the different automaton states are subdivided into three, not necessarily disjoint, subsets: one describing the state reached after a push or shift operation, and two to delimit the positions corresponding to each pop operation; in such a way a unique identification thereof is obtained.

2.4.1 *A Monadic Second-Order Logic over Operator Precedence Alphabets*

Let (Σ, M) be an OP alphabet. Consider a countable infinite set of first-order variables $\boldsymbol{x}, \boldsymbol{y}, \ldots$ and a countable infinite set of monadic second-order (set) variables $\boldsymbol{X}, \boldsymbol{Y}, \ldots$. As a convention, first and second-order variables will be denoted in boldface italic font.

Definition 2.10 (Monadic Second-order Logic over (Σ, M) [71])**.** 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 $\text{MSO}_{\Sigma,M}$

(*monadic second-order logic* over (Σ, M)) is defined by the following syntax (symbols Σ, M will be omitted unless necessary to prevent confusion):

$$\varphi := c(\boldsymbol{x}) \mid \boldsymbol{x} \in \boldsymbol{X} \mid \boldsymbol{x} \leq \boldsymbol{y} \mid \boldsymbol{x} \curvearrowright \boldsymbol{y} \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists \boldsymbol{x}.\varphi \mid \exists \boldsymbol{X}.\varphi$$

where $c \in \Sigma \cup \{\#\}$, $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}_1$, and $\boldsymbol{X} \in \mathcal{V}_2$.[6]

A MSO formula is interpreted over a (Σ, M) string w, with respect to assignments $\nu_1 : \mathcal{V}_1 \to \{0, 1, \ldots |w| + 1\}$ and $\nu_2 : \mathcal{V}_2 \to \wp(\{0, 1, \ldots |w| + 1\})$, in the following way.

- $\#w\#, M, \nu_1, \nu_2 \models c(\boldsymbol{x})$ iff $\#w\# = w_1 c w_2$ and $|w_1| = \nu_1(\boldsymbol{x})$.
- $\#w\#, M, \nu_1, \nu_2 \models \boldsymbol{x} \in \boldsymbol{X}$ iff $\nu_1(\boldsymbol{x}) \in \nu_2(\boldsymbol{X})$.
- $\#w\#, M, \nu_1, \nu_2 \models \boldsymbol{x} \leq \boldsymbol{y}$ iff $\nu_1(\boldsymbol{x}) \leq \nu_1(\boldsymbol{y})$.
- $\#w\#, M, \nu_1, \nu_2 \models \boldsymbol{x} \curvearrowright \boldsymbol{y}$ iff $\#w\# = w_1 a w_2 b w_3$, $|w_1| = \nu_1(\boldsymbol{x})$, $|w_1 a w_2| = \nu_1(\boldsymbol{y})$, and $a w_2 b$ is a chain ${}^a[w_2]^b$.
- $\#w\#, M, \nu_1, \nu_2 \models \neg\varphi$ iff $\#w\#, M, \nu_1, \nu_2 \not\models \varphi$.
- $\#w\#, M, \nu_1, \nu_2 \models \varphi_1 \vee \varphi_2$ iff $\#w\#, M, \nu_1, \nu_2 \models \varphi_1$ or $\#w\#, M, \nu_1, \nu_2 \models \varphi_2$.
- $\#w\#, M, \nu_1, \nu_2 \models \exists \boldsymbol{x}.\varphi$ iff $\#w\#, M, \nu_1', \nu_2 \models \varphi$, for some ν_1' with $\nu_1'(\boldsymbol{y}) = \nu_1(\boldsymbol{y})$ for all $\boldsymbol{y} \in \mathcal{V}_1 \setminus \{\boldsymbol{x}\}$.
- $\#w\#, M, \nu_1, \nu_2 \models \exists \boldsymbol{X}.\varphi$ iff $\#w\#, M, \nu_1, \nu_2' \models \varphi$, for some ν_2' with $\nu_2'(\boldsymbol{Y}) = \nu_2(\boldsymbol{Y})$ for all $\boldsymbol{Y} \in \mathcal{V}_2 \setminus \{\boldsymbol{X}\}$.

To improve readability, M, ν_1, ν_2 and the delimiters # are dropped from the notation whenever there is no risk of ambiguity; furthermore some standard abbreviations such as $\boldsymbol{x} + 1$, $\boldsymbol{x} - 1$, $\boldsymbol{x} = \boldsymbol{y}$, $\boldsymbol{x} \neq \boldsymbol{y}$, $\boldsymbol{x} < \boldsymbol{y}$, are used in formulae.

A *sentence* is a formula without free variables. The language of all strings $w \in \Sigma^*$ such that $w \models \varphi$ is denoted by $L(\varphi)$:

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

6 This is the usual MSO over strings, augmented with the $\curvearrowright$ predicate.

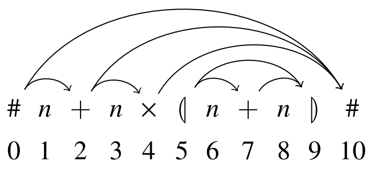


Figure 14.: The string of Figure 1, with positions and relation $\curvearrowright$.

Figure 14 illustrates the meaning of the $\curvearrowright$ relation with reference to the string of Figure 1: we have $0 \curvearrowright 2$, $2 \curvearrowright 4$, $5 \curvearrowright 7$, $7 \curvearrowright 9$, $5 \curvearrowright 9$, $4 \curvearrowright 10$, $2 \curvearrowright 10$, and $0 \curvearrowright 10$. 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 $\boldsymbol{x} \curvearrowright \boldsymbol{y}$ implies $\boldsymbol{y} > \boldsymbol{x} + 1$, and a position $\boldsymbol{x}$ may be in such a relation with more than one position and vice versa. Moreover, if w is compatible with M, then $0 \curvearrowright |w| + 1$.

Example 2.7. Consider the language of Example 2.1. The following sentence states that all parentheses are well-matched:

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \left(\boldsymbol{x} \curvearrowright \boldsymbol{y} \Rightarrow \left(\llparenthesis(\boldsymbol{x}+1) \Rightarrow \begin{array}{c} \rrparenthesis(\boldsymbol{y}-1) \land \\ \neg \exists \boldsymbol{z} (\boldsymbol{z} < \boldsymbol{y} \land \boldsymbol{x} \curvearrowright \boldsymbol{z}) \land \\ \neg \exists \boldsymbol{v} (\boldsymbol{x} < \boldsymbol{v} \land \boldsymbol{v} \curvearrowright \boldsymbol{y}) \end{array} \right) \right).$$

Note that this property is guaranteed a priori by the structure of the OPM.

The following sentence instead defines the language where parentheses are used only when they are needed (i.e. to give precedence of $+$ over $\times$).

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \left(\begin{array}{c} \boldsymbol{x} \curvearrowright \boldsymbol{y} \land \\ \llparenthesis(\boldsymbol{x}+1) \land \rrparenthesis(\boldsymbol{y}-1) \end{array} \Rightarrow (\times(\boldsymbol{x}) \lor \times(\boldsymbol{y})) \land \exists \boldsymbol{z} \left(\begin{array}{c} \boldsymbol{x}+1 < \boldsymbol{z} < \boldsymbol{y}-1 \land +(\boldsymbol{z}) \land \\ \neg \exists \boldsymbol{u} \exists \boldsymbol{v} \left(\begin{array}{c} \boldsymbol{x}+1 < \boldsymbol{u} < \boldsymbol{z} \land \llparenthesis(\boldsymbol{u}) \land \\ \boldsymbol{z} < \boldsymbol{v} < \boldsymbol{y}-1 \land \rrparenthesis(\boldsymbol{v}) \land \\ \boldsymbol{u}-1 \curvearrowright \boldsymbol{v}+1 \end{array} \right) \end{array} \right) \right)$$

The following main result holds.

Theorem 2.4 ([71])**.** *A language L over (Σ, M) is an OPL if and only if there exists a MSO sentence φ such that $L = L(\varphi)$.*

We report here the proof of the theorem presented in [71], which will provide the basis for the MSO logic characterization of ωOPLs in Chapter 3. The proof is constructive and structured in the following two subsections.

2.4.2 *From MSO to OPAs*

Statement 2.4 ([71])**.** *Let (Σ, M) be an operator precedence alphabet and φ be a MSO sentence. Then $L(\varphi)$ can be recognized by an OPA over (Σ, M).*

Proof. The proof follows the one by Thomas [96] and is composed of two steps: first the formula is rewritten so that no predicate symbols nor first order variables are used; then an equivalent OPA is built inductively.

Let Σ be $\{a_1, a_2, \ldots, a_n\}$. For each predicate symbol a_i, a fresh set variable $\boldsymbol{X}_i$ is introduced; therefore formula $a_i(\boldsymbol{x})$ will be translated into $\boldsymbol{x} \in \boldsymbol{X}_i$. Following the standard construction of [96], every first order variable is translated into a fresh second order variable with the additional constraint that the set it represents contains exactly one position. The only difference is that formulae like $\boldsymbol{x} \curvearrowright \boldsymbol{y}$ will be translated into formulae $\boldsymbol{X}_i \curvearrowright \boldsymbol{X}_j$, where $\boldsymbol{X}_i, \boldsymbol{X}_j$ are singleton sets. In this case, the semantics of $\curvearrowright$ is naturally extended to second order variables that are singletons.

Let φ' be the formula obtained from φ by such a translation, and consider any subformula ψ of φ': let $m(\psi)$ be the number of (second order) free variables appearing in ψ different from $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_n$, and denote them by $\boldsymbol{X}_{n+1}, \ldots \boldsymbol{X}_{n+m(\psi)}$. Recall that $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ represent symbols in Σ, hence they are never quantified.

As usual formulae are interpreted over strings; in this case the alphabet is:

$$\Lambda(\psi) = \left\{\alpha \in \{0,1\}^{n+m(\psi)} \mid \exists! i \text{ s.t. } 1 \le i \le n,\ \alpha_i = 1\right\}$$

A string $w \in \Lambda(\psi)^*$, with $|w| = \ell$, is used to interpret ψ in the following way: the projection over the j-th component of $\Lambda(\psi)$ gives a valuation $\{1, 2, \ldots, \ell\} \to \{0, 1\}$ of $\boldsymbol{X}_j$, for every $1 \le j \le n + m(\psi)$.

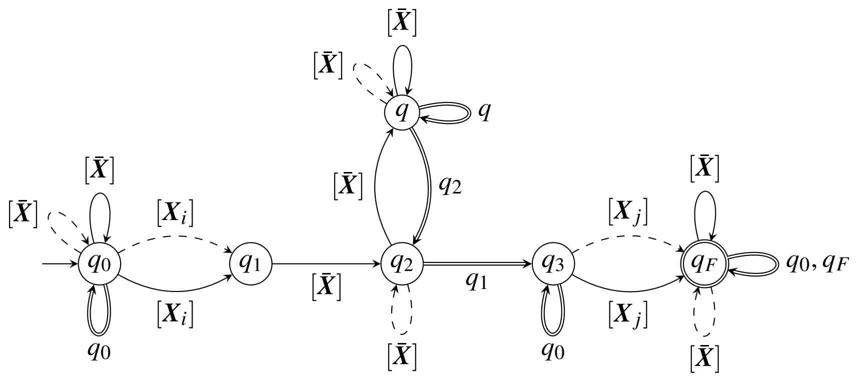


Figure 15.: OPA for atomic formula $\psi = X_i \curvearrowright X_j$

For any $\alpha \in \Lambda(\psi)$, the projection of α over the first n components encodes a symbol in Σ, denoted as $symb(\alpha)$. The matrix M over Σ can be naturally extended to the OPM $M(\psi)$ over $\Lambda(\psi)$ by defining $M(\psi)_{\alpha,\beta} = M_{symb(\alpha),symb(\beta)}$ for any $\alpha, \beta \in \Lambda(\psi)$.

Now one can build an OPA $\mathcal{A}$ equivalent to φ'. The construction is inductive on the structure of the formula: first one can define the OPA for all atomic formulae. We give here only the construction for $\curvearrowright$, since for the other ones the construction is standard and is the same as in [96].

Figure 15 represents the OPA for atomic formula $\psi = X_i \curvearrowright X_j$ (notice that $i, j > n$, and that both X_i and X_j are singleton sets). For the sake of brevity, notation $[X_i]$ is used to represent the set of all tuples $\Lambda(\psi)$ having the i-th component equal to 1; notation $[\bar{X}]$ represents the set of all tuples in $\Lambda(\psi)$ having both i-th and j-th components equal to 0.

The semantics of $\curvearrowright$ requires for $X_i \curvearrowright X_j$ that there must be a chain ${}^a[w_2]^b$ in the input word, where a is the symbol at the only position in X_i, and b is the symbol at the only position in X_j. By definition of chain, this means that a must be read, hence in the position represented by X_i the automaton performs either a push or a shift move (see Figure 15, 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 $\boldsymbol{X}_j$ – 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 $\boldsymbol{X}_i \curvearrowright \boldsymbol{X}_j$) portion of the syntax tree, when reading the symbol at position $\boldsymbol{X}_i$ performs either a push or a shift move, depending on whether $\boldsymbol{X}_i$ 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 $\boldsymbol{X}_j$, it accepts anything else that follows the examined fragment.

Then, a natural inductive path leads to the construction of the automaton associated with a generic MSO logic formula: the disjunction of two subformulae can be obtained by building the union automaton of the two corresponding automata; similarly for negation. The existential quantification of $\boldsymbol{X}_i$ is obtained by projection erasing the i-th component; since OPLs are closed under alphabetical homomorphisms preserving the OPM (see Statement 2.2), and since the OPM is determined only by the first n components of the alphabet's elements which are never erased by quantification such a projection produces a well defined automaton for any ψ. Finally, the alphabet of the automaton equivalent to φ' is $\Lambda(\varphi') = \{0, 1\}^n$, which is in bijection with Σ. □

2.4.3 *From OPAs to MSO*

When considering a chain ${}^{a}[w]^{b}$, assume that $w = w_0 a_1 w_1 \dots a_\ell w_\ell$, with ${}^{a}[a_1 a_2 \dots a_\ell]^{b}$ is a simple chain (any w_g may be empty). Also, denote by s_g the position of symbol a_g, for $g = 1, 2, \dots, \ell$ and set $a_0 = a$, $s_0 = 0$, $a_{\ell+1} = b$, and $s_{\ell+1} = |w| + 1$. The following shortcut notations is defined in [71]:

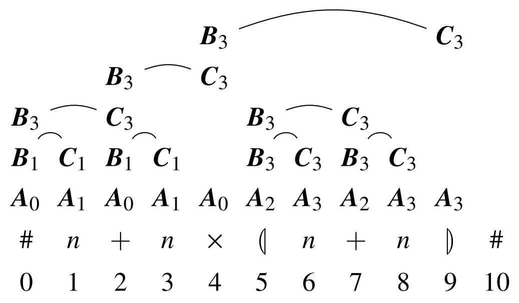
$$\boldsymbol{x} \circ \boldsymbol{y} \quad := \quad \bigvee_{M_{a,b} = \circ} a(\boldsymbol{x}) \wedge b(\boldsymbol{y}), \text{ for } \circ \in \{\lessdot, \doteq, \gtrdot\}$$

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

If $\boldsymbol{x} \curvearrowright \boldsymbol{y}$ then there exist (unique) $\boldsymbol{z}$ and $\boldsymbol{v}$ such that $\text{Tree}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y})$ is satisfied. In particular, if w is the body of a simple chain, then $0 \curvearrowright \ell + 1$ and $\text{Tree}(0, 1, \ell, \ell + 1)$ are satisfied; if it is the body of a composed chain, then $0 \curvearrowright |w| + 1$ and $\text{Tree}(0, s_1, s_\ell, s_{\ell+1})$ are satisfied. If $w_0 = \varepsilon$ then $s_1 = 1$, and if $w_\ell = \varepsilon$ then $s_\ell = |w|$. In the example of Figure 14 relations $\text{Tree}(2, 3, 3, 4)$, $\text{Tree}(2, 4, 4, 10)$, $\text{Tree}(4, 5, 9, 10)$, $\text{Tree}(5, 7, 7, 9)$ are satisfied, among others.

Statement 2.5 ([71])**.** *Let (Σ, M) be an operator precedence alphabet and $\mathcal{A}$ be an OPA over (Σ, M). Then there exists an MSO sentence φ such that $L(\mathcal{A}) = L(\varphi)$.*

Proof. Let $\mathcal{A} = \langle \Sigma, M, Q, q_0, F, \delta \rangle$ be *deterministic* (this simplifying assumption does not cause loss of generality, since nondeterministic OPAs are equivalent to deterministic ones by Theorem 2.1). W.l.o.g. assume also that the transition function of $\mathcal{A}$ is total. One can build a MSO sentence φ such that $L(\mathcal{A}) = L(\varphi)$. The main idea for encoding the behavior of the OPA is based on assigning the states visited during its run to positions along the same lines stated by Büchi [96] and extended for VPLs [11]. Unlike finite state automata and VPAs, however, OPAs do not work on-line. Hence, it is not possible to assign a single state to every position. Let $Q = \{q_0, q_1, \dots, q_N\}$ be the states of $\mathcal{A}$ with q_0 initial; as usual, they are encoded by second-order variables. Three different sets of second-order variables will be used, namely $\boldsymbol{A}_0, \boldsymbol{A}_1, \dots, \boldsymbol{A}_N$, $\boldsymbol{B}_0, \boldsymbol{B}_1, \dots, \boldsymbol{B}_N$ and $\boldsymbol{C}_0, \boldsymbol{C}_1, \dots, \boldsymbol{C}_N$. 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 $w_0 a_1 w_1 \dots a_l w_l$ of a chain whose support ends 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. Figure 16 presents such sets for the example automaton of Figure 8, with the same input as in Figure 14. 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 $\boldsymbol{C}_i$.



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
# n $+$ n $\times$ $\llparenthesis$ n $+$ n $\rrparenthesis$ #
0 1 2 3 4 5 6 7 8 9 10

Figure 16.: The string of Figure 1 with $\boldsymbol{B}_i, \boldsymbol{A}_i$, and $\boldsymbol{C}_i$ evidenced for the automaton of Figure 8. Pop moves of the automaton are represented by linked pairs $\boldsymbol{B}_i$, $\boldsymbol{C}_i$.

Then, sentence φ is defined as follows

$$\varphi := \exists \boldsymbol{e} \begin{array}{l} \exists \boldsymbol{A}_0, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_N \\ \exists \boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_N \\ \exists \boldsymbol{C}_0, \boldsymbol{C}_1, \ldots, \boldsymbol{C}_N \end{array} \left(\mathrm{Start}_0 \ \wedge \ \varphi_\delta \ \wedge \ \bigvee_{q_f \in F} \mathrm{End}_f \right), \tag{5}$$

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.

To complete the definition of φ, the following additional notations are introduced.

$$\begin{array}{rcl} \mathrm{Succ}_k(\boldsymbol{x}, \boldsymbol{y}) & := & \boldsymbol{x} + 1 = \boldsymbol{y} \wedge \boldsymbol{x} \in \boldsymbol{A}_k \\ \mathrm{Next}_k(\boldsymbol{x}, \boldsymbol{y}) & := & \boldsymbol{x} \curvearrowright \boldsymbol{y} \wedge \boldsymbol{x} \in \boldsymbol{B}_k \wedge \exists \boldsymbol{z}, \boldsymbol{v} \ (\mathrm{Tree}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y}) \wedge \boldsymbol{v} \in \boldsymbol{C}_k) \\ Q_i(\boldsymbol{x}, \boldsymbol{y}) & := & \mathrm{Succ}_i(\boldsymbol{x}, \boldsymbol{y}) \vee \mathrm{Next}_i(\boldsymbol{x}, \boldsymbol{y}) \end{array}$$

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, it is necessary to distinguish between push and shift moves (case

$\mathrm{Succ}_i(\boldsymbol{x},\boldsymbol{y})$), and pop moves (case $\mathrm{Next}_i(\boldsymbol{x},\boldsymbol{y})$). For instance, with reference to Figures 14 and 16; $\mathrm{Succ}_2(5,6)$, $\mathrm{Next}_3(5,9)$, and $\mathrm{Next}_3(5,7)$ hold.

The shortcuts representing the initial and final states are defined as follows.

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

$\varphi_{\delta_{\text{push}}}$ is the conjunction of the following two formulae. The former one states the sufficient condition for a position to be in a set $\boldsymbol{A}_i$, when performing a push move.

$$\varphi_{push\_fw} := \forall \boldsymbol{x}, \boldsymbol{y} \bigwedge_{i=0}^{N} \bigwedge_{k=0}^{N} \Big(\boldsymbol{x} \lessdot \boldsymbol{y} \wedge c(\boldsymbol{y}) \wedge Q_i(\boldsymbol{x},\boldsymbol{y}) \wedge \delta_{\text{push}}(q_i, c) = q_k \;\Rightarrow \boldsymbol{y} \in \boldsymbol{A}_k \Big)$$

The latter formula states the symmetric necessary condition

$$\varphi_{push\_bw} := \forall \boldsymbol{x}, \boldsymbol{y} \bigwedge_{k=0}^{N} \left(\begin{array}{c} \boldsymbol{x} \lessdot \boldsymbol{y} \;\wedge\; c(\boldsymbol{y}) \;\wedge\; \boldsymbol{y} \in \boldsymbol{A}_k \\ \wedge \\ (\boldsymbol{x}+1 = \boldsymbol{y} \;\vee\; \boldsymbol{x} \curvearrowright \boldsymbol{y}) \end{array} \Rightarrow \bigvee_{i=0}^{N} \Big(Q_i(\boldsymbol{x},\boldsymbol{y}) \wedge \delta_{\text{push}}(q_i, c) = q_k \Big) \right)$$

$\varphi_{\delta_{\text{shift}}}$ is defined analogously, with respect to shift moves instead of push moves.

$$\begin{aligned}\varphi_{shift\_fw} &:= \forall \boldsymbol{x}, \boldsymbol{y} \bigwedge_{i=0}^{N} \bigwedge_{k=0}^{N} (\; \boldsymbol{x} \doteq \boldsymbol{y} \;\wedge\; c(\boldsymbol{y}) \wedge Q_i(\boldsymbol{x},\boldsymbol{y}) \;\wedge\; \delta_{\text{shift}}(q_i, c) = q_k \;\Rightarrow \boldsymbol{y} \in \boldsymbol{A}_k) \\ \varphi_{shift\_bw} &:= \forall \boldsymbol{x}, \boldsymbol{y} \bigwedge_{k=0}^{N} \left(\begin{array}{c} \boldsymbol{x} \doteq \boldsymbol{y} \;\wedge\; c(\boldsymbol{y}) \;\wedge\; \boldsymbol{y} \in \boldsymbol{A}_k \\ \wedge \\ (\boldsymbol{x}+1 = \boldsymbol{y} \;\vee\; \boldsymbol{x} \curvearrowright \boldsymbol{y}) \end{array} \Rightarrow \bigvee_{i=0}^{N} (Q_i(\boldsymbol{x},\boldsymbol{y}) \wedge \delta_{\text{shift}}(q_i, c) = q_k) \right)\end{aligned}$$

Finally, to define $\varphi_{\delta_{\text{pop}}}$, introduce the shortcut $\mathrm{Tree}_{i,j}(\boldsymbol{x},\boldsymbol{z},\boldsymbol{v},\boldsymbol{y})$, 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).

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

Formula $\varphi_{\delta_{\text{pop}}}$ is thus defined as the conjunction of three formulae. As before, the forward formula gives 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 formulae state symmetric necessary conditions.

$$\begin{aligned}
\varphi_{pop\_fw} \quad &:= \quad \forall \boldsymbol{x},z,\boldsymbol{v},\boldsymbol{y} \bigwedge_{i=0}^{N}\bigwedge_{j=0}^{N}\bigwedge_{k=0}^{N}\left(\begin{array}{c}\mathrm{Tree}_{i,j}(\boldsymbol{x},z,\boldsymbol{v},\boldsymbol{y})\\ \wedge\\ \delta_{\text{pop}}(q_i,q_j)=q_k\end{array} \Rightarrow \boldsymbol{x}\in\boldsymbol{B}_k \wedge \boldsymbol{v}\in\boldsymbol{C}_k\right)\\
\varphi_{pop\_bwB} \quad &:= \quad \forall \boldsymbol{x} \bigwedge_{k=0}^{N}\left(\boldsymbol{x}\in\boldsymbol{B}_k \Rightarrow \exists \boldsymbol{y},z,\boldsymbol{v}\bigvee_{i=0}^{N}\bigvee_{j=0}^{N}\mathrm{Tree}_{i,j}(\boldsymbol{x},z,\boldsymbol{v},\boldsymbol{y}) \wedge \delta_{\text{pop}}(q_i,q_j)=q_k\right)\\
\varphi_{pop\_bwC} \quad &:= \quad \forall \boldsymbol{v} \bigwedge_{k=0}^{N}\left(\boldsymbol{v}\in\boldsymbol{C}_k \Rightarrow \exists \boldsymbol{x},\boldsymbol{y},z\bigvee_{i=0}^{N}\bigvee_{j=0}^{N}\mathrm{Tree}_{i,j}(\boldsymbol{x},z,\boldsymbol{v},\boldsymbol{y}) \wedge \delta_{\text{pop}}(q_i,q_j)=q_k\right)
\end{aligned}$$

Now notice that $\varphi = \bigvee_{q_f\in F}\psi_{0,f}$, where

$$\psi_{i,k} \ := \ \exists \boldsymbol{e} \ \begin{array}{l}\exists \boldsymbol{A}_0,\boldsymbol{A}_1,\ldots,\boldsymbol{A}_N\\ \exists \boldsymbol{B}_0,\boldsymbol{B}_1,\ldots,\boldsymbol{B}_N\\ \exists \boldsymbol{C}_0,\boldsymbol{C}_1,\ldots,\boldsymbol{C}_N\end{array} \ (\mathrm{Start}_i \ \wedge \ \varphi_\delta \ \wedge \ \mathrm{End}_k)$$

Hence, the proof that $L(\mathcal{A}) = L(\varphi)$ is direct consequence of the following Lemmata 2.5 and 2.6, stating that $w \models \psi_{i,k}$ if and only if $q_i \overset{w}{\rightsquigarrow} q_k$ in $\mathcal{A}$, for every word w compatible with (Σ, M). □

Lemma 2.5 ([71])**.** *Let w be the body of a chain ${}^{\#}[w]^{\#}$. If $q_i \overset{w}{\rightsquigarrow} q_k$ in $\mathcal{A}$, then $w \models \psi_{i,k}$.*

Proof. The proof of the lemma is by induction on the depth of chains. Note that, even if $\mathcal{A}$ is deterministic, some chains could have different supports; however, it is shown that every support produces exactly one assignment that satisfies $\psi_{i,k}$.

Let w be the body of a simple chain with support

$$q_i = q_{t_0} \xrightarrow{a_1} q_{t_1} \dashrightarrow^{a_2} \dots \dashrightarrow^{a_\ell} q_{t_\ell} \overset{q_{t_0}}{\Longrightarrow} q_k \tag{6}$$

One can prove that $w \models \psi_{i,k}$ for $\boldsymbol{e}, \boldsymbol{A}_0, \boldsymbol{A}_1, \dots, \boldsymbol{A}_N, \boldsymbol{B}_0, \dots, \boldsymbol{B}_N, \boldsymbol{C}_0, \dots, \boldsymbol{C}_N$ defined as follows. First-order variable $\boldsymbol{e}$ equals $|w|$, $\boldsymbol{B}_h$ is empty except for $\boldsymbol{B}_k = \{0\}$; $\boldsymbol{C}_h$ is empty except for $\boldsymbol{C}_k = \{\ell\}$; for every $0 \leq \boldsymbol{x} \leq \ell$, let $\boldsymbol{A}_h$ contain $\boldsymbol{x}$ iff $t_{\boldsymbol{x}} = h$ (i.e., $\boldsymbol{x} \in \boldsymbol{A}_{t_{\boldsymbol{x}}}$), and this also implies $Q_{t_{\boldsymbol{x}}}(\boldsymbol{x}, \boldsymbol{x}+1)$. Then Start$_i$ and End$_k$ are satisfied trivially since Tree$(0, 1, \ell, \ell+1)$ holds. One can now prove that also $\varphi_{\delta_{push}}$, $\varphi_{\delta_{shift}}$, and $\varphi_{\delta_{pop}}$ are satisfied; we omit to consider all cases where the antecedents are false.

- φ_{push} is satisfied for $\boldsymbol{x} = 0$ and $\boldsymbol{y} = 1$ since we have $a_1(1)$, $\# \lessdot a_1$, $Q_i(0,1)$, $1 \in \boldsymbol{A}_{t_1}$, and $\delta_{\text{push}}(q_i, a_1) = q_{t_1}$.
- φ_{shift} is satisfied $\forall 1 \leq \boldsymbol{x} < \ell$ and $\boldsymbol{y} = \boldsymbol{x}+1$ since we have $a_{\boldsymbol{y}}(\boldsymbol{y})$, $a_{\boldsymbol{x}} \doteq a_{\boldsymbol{y}}$, $Q_{t_{\boldsymbol{x}}}(\boldsymbol{x}, \boldsymbol{y})$, $\boldsymbol{y} \in \boldsymbol{A}_{t_{\boldsymbol{y}}}$, and $\delta_{\text{shift}}(q_{t_x}, a_y) = q_{t_y}$.
- φ_{pop} is satisfied for $\boldsymbol{x} = 0$ and $\boldsymbol{y} = |w|+1 = \ell+1$ since we have Tree$_{t_\ell,i}(0, 1, \ell, \ell+1)$, $0 \in \boldsymbol{B}_k$, $\ell \in \boldsymbol{C}_k$, and $\delta_{\text{pop}}(q_{t_\ell}, q_i) = q_k$.

Let now w be the body of a composed chain with support

$$q_i = q_{t_0} \overset{w_0}{\rightsquigarrow} q_{f_0} \xrightarrow{a_1} q_{t_1} \overset{w_1}{\rightsquigarrow} q_{f_1} \dashrightarrow^{a_2} \dots \dashrightarrow^{a_g} q_{t_g} \overset{w_g}{\rightsquigarrow} q_{f_g} \dots \dashrightarrow^{a_\ell} q_{t_\ell} \overset{w_\ell}{\rightsquigarrow} q_{f_\ell} \overset{q_{f_0}}{\Longrightarrow} q_k \tag{7}$$

One can prove that $w \models \psi_{i,k}$ for a suitable assignment. By the inductive hypothesis, for every $g = 0, 1, \dots, \ell$ such that $w_g \neq \varepsilon$ we have $w_g \models \psi_{t_g, f_g}$. Let $\boldsymbol{A}_0{}^g, \dots, \boldsymbol{A}_N{}^g, \boldsymbol{B}_0{}^g$, $\dots$, $\boldsymbol{B}_N{}^g$, $\boldsymbol{C}_0{}^g, \dots, \boldsymbol{C}_N{}^g$ be (the naturally shifted versions of) an assignment that satisfies ψ_{t_g, f_g}. In particular this implies $s_g \in \boldsymbol{A}_{t_g}$, Next$_{f_g}(s_g, s_{g+1})$, and $s_g \in \boldsymbol{A}_{t_g} \cup \boldsymbol{B}_{f_g}$, for each g such that $w_g \neq \varepsilon$. Then define $\boldsymbol{A}_h, \boldsymbol{B}_h, \boldsymbol{C}_h$ as follows. Let $\boldsymbol{A}_h$ include all $\boldsymbol{A}_h{}^g$, $\boldsymbol{B}_h$ include all $\boldsymbol{B}_h{}^g$, $\boldsymbol{C}_h$ include all $\boldsymbol{C}_h{}^g$. Also let $\boldsymbol{B}_k$ contain s_0, $\boldsymbol{C}_k$ contain s_ℓ, and $\boldsymbol{A}_{t_g}$

contain s_g whenever w_g is empty; in particular this implies $Q_{f_g}(s_g, s_{g+1})$ for every $0 \leq g < \ell$. Finally, $\boldsymbol{e}$ is defined as the length of w.

Then one can show that $\psi_{i,k}$ is satisfied by checking every subformula. Start_i and End_k are satisfied trivially since $\text{Tree}(0, s_1, s_\ell, |w| + 1)$ holds. By the inductive hypothesis, all other axioms are satisfied within every w_g. Thus, one has only to prove that they are satisfied in positions s_g, for $0 \leq g \leq \ell$. We omit to consider all cases where the antecedents are false.

- φ_{push} is satisfied for $\boldsymbol{x} = 0$ and $\boldsymbol{y} = s_1$ since we have $a_1(s_1)$, $\# \lessdot a_1$ $Q_{f_0}(0, s_1)$, $s_1 \in \boldsymbol{A}_{t_1}$, and $\delta_{\text{push}}(q_{f_0}, a_1) = q_{t_1}$.
- φ_{shift} is satisfied for all $\boldsymbol{x} = s_g$ and $\boldsymbol{y} = s_{g+1}$ with $1 \leq g < \ell$ since we have $a_g(s_g)$, $a_{s_g} \doteq a_{s_{g+1}}$, $Q_{f_g}(s_g, s_{g+1})$, $s_g \in \boldsymbol{A}_{t_g}$, and $\delta_{\text{shift}}(q_{f_g}, a_g) = q_{t_g}$.
- φ_{pop} is satisfied for $\boldsymbol{x} = 0$ and $\boldsymbol{y} = |w| + 1$ since we have $\text{Tree}_{f_\ell, f_0}(0, s_1, s_\ell, |w| + 1)$, $0 \in \boldsymbol{B}_k$, $\ell \in \boldsymbol{C}_k$, and $\delta_{\text{pop}}(q_{t_\ell}, q_i) = q_k$.

Hence $w \models \psi_{i,k}$ for every w with a suitable support, and this concludes the proof. □

Lemma 2.6 ([71])**.** *Let w be the body of a chain ${}^{\#}[w]^{\#}$. If $w \models \psi_{i,k}$ then $q_i \overset{w}{\rightsquigarrow} q_k$ in $\mathcal{A}$.*

Proof. Let $\boldsymbol{e} = |w|, \boldsymbol{A}_0, \ldots, \boldsymbol{A}_N, \boldsymbol{B}_0, \ldots, \boldsymbol{B}_N, \boldsymbol{C}_0, \ldots, \boldsymbol{C}_N$ be an assignment that satisfies $\psi_{i,k}$. In particular this implies $0 \in \boldsymbol{A}_i \wedge \text{Next}_k(0, |w| + 1)$, and such i, k are unique by definition of Start_i and End_k. Then the following properties hold.

(i) For each $0 \leq \boldsymbol{x} \leq |w|$, there exists a unique index i such that $\text{Succ}_i(\boldsymbol{x}, \boldsymbol{x} + 1)$ holds true. This can be proved by induction on $\boldsymbol{x}$ by applying the formulae for δ_{push} and δ_{shift}.

(ii) For each $\boldsymbol{x}, \boldsymbol{y}$ such that $\boldsymbol{x} \curvearrowright \boldsymbol{y}$, let $\boldsymbol{z}, \boldsymbol{v}$ such that $\text{Tree}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y})$ holds, then there exists a unique pair of indices i, j such that $\text{Tree}_{i,j}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y})$ holds, and there exists a unique index k such that $\text{Next}_k(\boldsymbol{x}, \boldsymbol{y})$. This can be proved by induction on the depth of the chain between positions $\boldsymbol{x}$ and $\boldsymbol{y}$, by applying the formulae for δ_{pop} and property (i).

Moreover, if $\text{Tree}_{i,j}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y})$ holds, then $\text{Next}_k(\boldsymbol{x}, \boldsymbol{y})$ holds if and only if $\delta_{\text{pop}}(q_i, q_j) = q_k$.

Hence, by properties (i) and (ii), for each $\boldsymbol{x}, \boldsymbol{y}$ such that $\boldsymbol{x} + 1 = \boldsymbol{y}$ or $\boldsymbol{x} \curvearrowright \boldsymbol{y}$, there exists a unique i such that $Q_i(\boldsymbol{x}, \boldsymbol{y})$ holds true.

Now, for every g let t_g be the index such that $g \in \boldsymbol{A}_{t_g}$. t_g is unique by property (i) and in particular $t_0 = i$.

The proof proceeds by induction on the depth h of w. Let $h = 1$ and $w = a_1 a_2 \dots a_\ell$ be the body of a simple chain. In this case t_g is the unique index such that $\mathrm{Succ}_{t_g}(g, g+1)$. Then, by φ_{push_bw} with $\boldsymbol{y} = 1$, we have $\delta(q_{t_0}, a_1) = q_{t_1}$; and by φ_{shift_bw} with $1 \leq g < \ell$, we have $\delta_{\text{shift}}(q_{t_g}, a_{g+1}) = q_{t_{g+1}}$. Moreover, since $\mathrm{Tree}_{t_\ell, t_0}(0, 1, \ell, \ell + 1) \wedge \mathrm{Next}_k(0, \ell + 1)$, we get $\delta(q_{t_\ell}, q_{t_0}) = q_k$ by property (ii). Hence we have built a support of the type (6).

Let now be $h > 1$ and $w = w_0 a_1 w_1 \dots a_\ell w_\ell$. For $0 \leq g \leq \ell$, since $s_g \curvearrowright s_{g+1} \vee s_{g+1} = s_g + 1$, by properties (i) and (ii) above there exists a unique index f_g such that $Q_{f_g}(s_g, s_{g+1})$ holds. Notice that $w_g = \varepsilon$ implies $f_g = t_g$, otherwise we have $w_g \models \psi_{t_g, f_g}$ and, by the inductive hypothesis, there exists a support $q_{t_g} \overset{s_g}{\rightsquigarrow} q_{f_g}$ in $\mathcal{A}$. Thus, for every $0 \leq g < \ell$, by applying φ_{push_bw} with $\boldsymbol{y} = s_{g+1}$ we get $\delta(q_{f_g}, a_{g+1}) = q_{t_{g+1}}$. Moreover, since $\mathrm{Tree}_{f_\ell, f_0}(0, s_1, s_\ell, |w| + 1) \wedge \mathrm{Next}_k(0, |w| + 1)$, by property (ii) above we get $\delta(q_{t_\ell}, q_i) = q_k$. Hence a support of type (7) has been built and this concludes the proof. □

3 AUTOMATA AND LOGIC-BASED CHARACTERIZATION OF ωOPLS

Languages of infinite-length strings, called ω-languages, have been introduced to model nonterminating processes; thus they are becoming more and more relevant nowadays when most applications are "ever-running", often in a distributed environment. The foundations of the theory of ω-languages are due to the pioneering work by Büchi [24] and others [78, 74, 86, 19]. Büchi, in particular, investigated their main algebraic properties in the context of finite state machines, pointing out commonalities and differences w.r.t. the finite length counterpart [24, 96]. His work has then been extended to larger classes of languages, among them, noticeably, the class of VPLs, for which both an automata and logic-based characterization has been provided.

In this chapter we follow the same path for the class of OPLs. OPLs, in fact, are not only useful to model programs, which are typically of finite length, but are also well-suited to formalize possibly never-ending sequences of events for systems in various contexts (operating systems, databases,...). Herein, we introduce the basic definitions of OPLs of infinite length strings and we characterize them in terms of a class of automata and a MSO logic.

More precisely, this chapter is organized as follows. In Section 3.1 we first extend to ω-languages a few basic notions given in Chapter 2 for finite-length languages and generalize to OPAs the classical accepting criteria for ω-languages; then we show by means of an example the usefulness of ωOPAs to model and analyze various system types. Section 3.2 shows the relations between the various classes of ωOPLs classified according to the acceptance criteria defined in the previous section; Section 3.3 shows which closure properties are preserved and which ones are lost when moving from finite length languages to the various classes of ω-languages; finally, Section 3.4 extends to ω-languages the characterization in terms of MSO logic that has been presented for OPLs in Chapter 2.

3.1 BASIC DEFINITIONS OF ω-LANGUAGES

We introduce some definitions that tailor the model of OPLs to its extension to ω-languages. Preliminarily we state some properties related to chains that are relevant when they occur within infinite words.

Definition 3.1. Let (Σ, M) be a precedence alphabet and w a word on Σ compatible with M:

- A chain in w is *maximal* if it does not belong to a larger composed chain. In a finite word w preceded and ended by #, only the outmost chain ${}^{\#}[w]^{\#}$ is maximal.
- An *open chain* is a sequence of symbols $b_0 \lessdot a_1 \doteq a_2 \doteq \ldots \doteq a_n$, for $n \geq 1$.
- A letter $a \in \Sigma$ in a word $\#w$ with $w \in \Sigma^*$ compatible with M, is *pending* if it does not belong to the body of a chain. In a word w preceded and ended by #, there are no pending letters.

Furthermore, we generalize in a natural way to the infinite case the notion of string compatible with an OPM: given a precedence alphabet (Σ, M), we say that an ω-word w is *compatible* with the OPM M if every prefix of w is compatible with M. We denote by $L_M \subseteq \Sigma^\omega$ the ω-language comprising all infinite words $x \in \Sigma^\omega$ compatible with M.

We adopt for OPAs operating on infinite strings the same acceptance criteria that have been adopted in the literature for regular and other classes of languages.

Definition 3.2 (Büchi operator precedence ω-automaton)**.** A *nondeterministic Büchi operator precedence ω-automaton* (ωOPBA) is given by a tuple $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$, where Σ, Q, I, F, δ are defined as for OPAs; the operator precedence matrix M is restricted to be a $|\Sigma \cup \{\#\}| \times |\Sigma|$ array, since ω-words are not terminated by the delimiter #.

Configurations and *(infinite) runs* are defined as for operator precedence automata on finite-length words. Let "$\exists^\omega i$" be a shorthand for "there exist infinitely many i" and let ρ be a run of the automaton on a given word $x \in \Sigma^\omega$. Define $Inf(\rho) = \{q \in Q \mid \exists^\omega i \ \langle \beta_i, \ q_i, \ x_i \rangle \in \rho, \ q_i = q\}$ as the set of states that occur infinitely often in configurations in ρ. A run ρ of an ωOPBA on an infinite word $x \in \Sigma^\omega$ is *successful* iff there exists a state $q_f \in F$ such that $q_f \in Inf(\rho)$. $\mathcal{A}$ *accepts* $x \in \Sigma^\omega$ iff there is a successful run of $\mathcal{A}$ on x. The ω-language *recognized* by $\mathcal{A}$ is $L(\mathcal{A}) = \{x \in \Sigma^\omega \mid \mathcal{A} \text{ accepts } x\}$.

The classical notion of acceptance for Muller automata can be likewise defined for OPAs.

Definition 3.3 (Muller operator precedence ω-automaton)**.** A *nondeterministic Muller operator precedence automaton* (ωOPMA) is a tuple $\langle \Sigma, M, Q, I, \mathcal{T}, \delta \rangle$ where Σ, M, Q, I, δ are defined as for ωOPBAs and $\mathcal{T}$ is a collection of subsets of Q, $\mathcal{T} \subseteq \wp(Q)$, called the *table* of the automaton.
A run ρ of an ωOPMA on an infinite word $x \in \Sigma^\omega$ is *successful* iff $Inf(\rho) \in \mathcal{T}$, i.e. the set of states occurring infinitely often in the configurations of ρ is a set in the table $\mathcal{T}$.

Definition 3.4. A *nondeterministic Büchi operator precedence automaton accepting with empty stack* (ωOPBEA) is a variant of ωOPBA where a run ρ is successful iff there exists a state $q_f \in \mathcal{F}$ such that configurations with stack $\perp$ and state q_f occur infinitely often in ρ.

Thus, a run of an ωOPBEA is successful iff the automaton traverses final states with an empty stack infinitely often. We will use the following simple normal form for ωOPBEA.

Definition 3.5. An ωOPBEA is in *normal form* if the set of states is partitioned into states that are always visited with empty stack and states that are never visited with empty stack.

For all above classes of automata, say, ω-XXX, their deterministic counterpart ω-DXXX is defined as usual.

Example 3.1 (Managing interrupts)**.** Consider as an example a software system that is designed to work forever and must serve requests issued by different users but subject to interrupts. Precisely, assume that the system manages two types of "normal operations" a and b, and two types of interrupts, with different levels of priority.

We model its behavior by introducing an alphabet with two pairs of calls and returns, $call_a$, $call_b$, ret_a, ret_b, for operations a and b and symbols int_1, $serve_1$ denoting the lower level interrupt and its serving, respectively, and int_2, $serve_2$ denoting the higher level ones. Not only both interrupts discard possible pending calls not already matched by corresponding returns, but also the serving of a higher priority interrupt erases possible pending requests for lower priority ones, but not those that occurred before the higher priority interrupt just served: thus, a sequence such as $int_1 int_2\ int_1\ int_1\ serve_2$ should produce popping the second and third int_1 without matching them, to match immediately int_2 with $serve_2$, but would leave the first occurrence of int_1 still pending; the next $serve_1$, if any, would match it, whereas possible further $serve_1$ would remain unmatched. Furthermore neither calls to, nor returns from, operations a and b can occur while any interrupt is pending.

Figure 17 shows an OPM that assigns to sequences on the above alphabet a structure compatible with the described priorities. Then, a suitable ω-automaton can specify further constraints on such sequences; for instance the ωOPBA of Figure 18 restricts the set of ω-sequences compatible with the matrix by imposing that all int_2 are eventually served by a corresponding $serve_2$; furthermore lower priority interrupts are not just discarded when a higher priority one is pending but they are simply disabled, i.e. they are not accepted as a correct system behavior.

For instance, the ω-word $call_a\ int_1\ int_2\ int_1 \ldots$ is not accepted by the ωOPBA because int_1 is not accepted from state q_2 reached after reading int_2; similarly, $call_a\ int_1\ int_2\ serve_2\ call_a$ is rejected since, after serving int_2 the automaton would be back in state q_1 with int_1 pending (the prefix $call_a\ int_1\ int_2\ serve_2$ is compatible with the OPM and int_1 is pending therein) but no $call_a$ is admitted in q_1 since there is no precedence relation between int_1 and $call_a$. On the contrary the ω-word $call_a\ int_1 (int_2\ serve_2\ serve_1\ call_a\ call_a\ ret_a)^\omega$ is accepted: in fact the automaton reaches q_1 after reading $call_a$ (and popping it) followed by int_1; then, after receiving and serving the higher priority interrupt, it would serve the pending instance of int_1 returning to q_0; from this point on it would enter an infinite loop during which it would process the input string

$(call_a\ call_a\ ret_a\ int_2\ serve_2\ serve_1)^\omega$ traversing the states $q_0 \xrightarrow{call_a} q_0 \xrightarrow{call_a} q_0 \overset{ret_a}{\dashrightarrow} q_0 \overset{q_0}{\Longrightarrow} q_0 \xrightarrow{int_2} q_2 \overset{serve_2}{\dashrightarrow} q_2 \overset{q_2}{\Longrightarrow} q_0 \overset{q_0}{\Longrightarrow} q_0 \xrightarrow{serve_1} q_1 \overset{q_0}{\Longrightarrow} q_0$ leaving the first $call_a$ and $serve_1$ unmatched. Notice that all finite prefixes $call_a\ int_1(int_2\ serve_2\ serve_1\ call_a\ call_a\ ret_a)^n\ int_2\ serve_2\ serve_1\ call_a\ call_a$, with $n > 0$, end with the open chain $call_a \lessdot call_a$. Finally, observe that the automaton would accept some strings beginning with $serve_1$ which might appear somewhat counterintuitive but is consistent with the general philosophy of admitting unmatched elements; it would be easy, however, to forbid such a string beginning.

We call $L_{\text{interrupt}}$ the language recognized by this ωOPBA.

	$call_a$	ret_a	$call_b$	ret_b	int_1	int_2	$serve_1$	$serve_2$
$call_a$	$\lessdot$	$\doteq$	$\lessdot$		$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
ret_a	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$call_b$	$\lessdot$		$\lessdot$	$\doteq$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
ret_b	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
int_1					$\lessdot$	$\lessdot$	$\doteq$	$\gtrdot$
int_2					$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$
$serve_1$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$serve_2$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
#	$\lessdot$		$\lessdot$		$\lessdot$	$\lessdot$	$\lessdot$	

Figure 17.

A more sophisticated policy that could easily be formalized by means of a suitable ω-automaton is a "weak fairness requirement" imposing that, after a first $call_a$ not matched by ret_a but interrupted by a int_1 or int_2, a second $call_a$ cannot be interrupted by a new lower priority interrupt int_1 (but can still be interrupted at any time by higher priority ones).

This example too retains some typical features of VPLs, namely the possibility of having unmatched calls or returns, but it strongly generalizes them in that unmatched elements can occur in various places of the whole string, e.g., due to the occurrence of interrupts or other exceptional events.

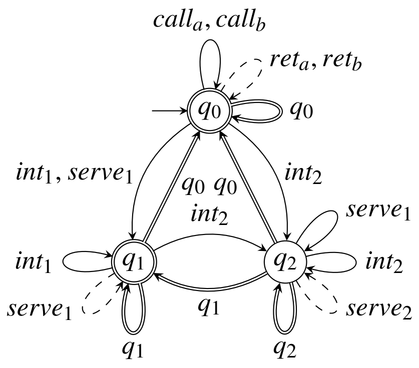


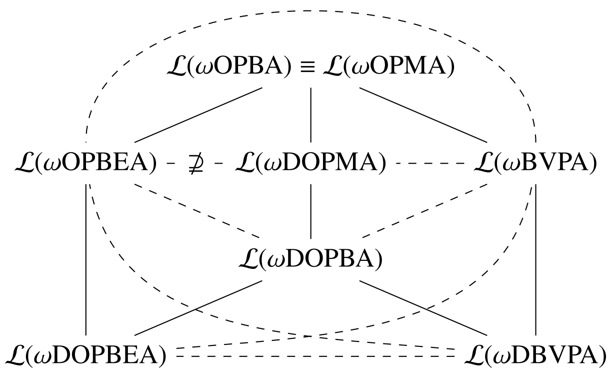
Figure 18.: ωOPBA recognizing the language of Example 3.1.

Further examples illustrating the modeling capabilities of OPLs both on finite and infinite strings are reported in [82].

3.2 RELATIONSHIPS AMONG CLASSES OF ωOPLS

In this section we describe the relationships among languages recognized by the different classes of operator precedence ω-automata and visibly pushdown ω-automata (with Büchi acceptance criterion), denoted as ωBVPA. Such relations are summarized by the diagram in Figure 19, where solid lines denote strict inclusion and dashed lines link classes that are not comparable.

In the following, we first present the proofs of the weak containment relations holding among the various classes: most of them follow trivially from the definitions, except for the equality between $\mathcal{L}$(ωOPBA) and $\mathcal{L}$(ωOPMA). Then we will prove strict inclusions and incomparability relations by means of a suitable set of examples that separate the various classes.



$\mathcal{L}(\omega\text{OPBA}) \equiv \mathcal{L}(\omega\text{OPMA})$
$\mathcal{L}(\omega\text{OPBEA})$
$\not\supseteq$
$\mathcal{L}(\omega\text{DOPMA})$
$\mathcal{L}(\omega\text{BVPA})$
$\mathcal{L}(\omega\text{DOPBA})$
$\mathcal{L}(\omega\text{DOPBEA})$
$\mathcal{L}(\omega\text{DBVPA})$

Figure 19.: Containment relations for ωOPLs. Solid lines denote strict inclusion of the lower class in the upper one; dashed lines link classes which are not comparable. It is still open whether $\mathcal{L}(\omega\text{OPBEA}) \subseteq \mathcal{L}(\omega\text{DOPMA})$ or not.

3.2.1 *Weak inclusion results*

Theorem 3.1. *The following inclusion relations hold:*

$$\mathcal{L}(\omega BVPA) \subseteq \mathcal{L}(\omega OPBA), \qquad \mathcal{L}(\omega DBVPA) \subseteq \mathcal{L}(\omega DOPBA).$$

Proof. Let $\mathcal{A} = \langle Q_A, I_A, \Gamma_A, \delta_A, F_A \rangle$ be an ωBVPA[7] over a partitioned alphabet $\Sigma = (\Sigma_c, \Sigma_r, \Sigma_i)$. An ωOPBA $\mathcal{B}$ that recognizes the same language as $\mathcal{A}$ is defined in a straightforward way as follows: $\mathcal{B} = \langle \Sigma, M, Q_B, I_B, \delta_B, F_B \rangle$ where

- $Q_B = Q_A \times \Gamma_A$,
- $I_B = I_A \times \{\top\}$,
- $F_B = F_A \times \Gamma_A$,

7 Among the many equivalent definitions for VPAs we adopt here the original one in [10].

- M is the precedence matrix induced by the partition on Σ:

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

- the transition function $\delta : Q_B \times (\Sigma \cup Q_B) \to \wp(Q_B)$ is defined as follows, where $q_1, q_2 \in Q_A$.
  The push transition $\delta_{\text{Bpush}} : Q_B \times \Sigma \to \wp(Q_B)$ is defined by:
  - for $a \in \Sigma_c$, $\delta_{\text{Bpush}}(\langle q_1, \gamma_1 \rangle, a) = \{\langle q_2, \gamma_2 \rangle \mid (q_1, a, q_2, \gamma_2) \in \delta_A\}$
  - for $a \in \Sigma_i$, $\delta_{\text{Bpush}}(\langle q_1, \gamma \rangle, a) = \{\langle q_2, \gamma \rangle \mid (q_1, a, q_2) \in \delta_A\}$
  - for $a \in \Sigma_r$, $\delta_{\text{Bpush}}(\langle q_1, \top \rangle, a) = \{\langle q_2, \top \rangle \mid (q_1, a, \top, q_2) \in \delta_A\}$.

  The shift transition $\delta_{\text{Bshift}} : Q_B \times \Sigma \to \wp(Q_B)$ is defined by
  - for $a \in \Sigma_r$, $\delta_{\text{Bshift}}(\langle q_1, \gamma \rangle, a) = \{\langle q_2, \gamma \rangle \mid (q_1, a, \gamma, q_2) \in \delta_A\}$, i.e., the ωOPBA simulates the pop move of the ωBVPA by setting, as state q_2, a state reached by the ωBVPA while reading the return symbol a.

  The pop transition $\delta_{\text{pop}} : Q_B \times Q_B \to \wp(Q_B)$ is defined as follows:
  - $\delta_{\text{Bpop}}(\langle q_1, \gamma_1 \rangle, \langle q_2, \gamma_2 \rangle) = \{\langle q_1, \gamma_2 \rangle\}$, i.e., restores the state reached by the ωBVPA after its pop move.

If the original ωBVPA is deterministic, so is the ωOPBA obtained with the above construction, and this yields the second relation. □

Proposition 3.1. *The following inclusion relations hold:*

$$\mathcal{L}(\omega OPBEA) \subseteq \mathcal{L}(\omega OPBA),$$

$$\mathcal{L}(\omega DOPBEA) \subseteq \mathcal{L}(\omega DOPBA) \subseteq \mathcal{L}(\omega DOPMA) \subseteq \mathcal{L}(\omega OPMA).$$

Proof. The first inclusion follows from the definition of ωOPBA and ωOPBEA in normal form: given an ωOPBEA whose set of states is partitioned into states that are always visited with empty stack and states that are never visited with empty stack, we can define an equivalent ωOPBA that has as final states the final states of the ωOPBEA that are always visited with empty stack.

The inclusion follows similarly for the deterministic counterparts of these classes of ωOPAs, since this ωOPBA is deterministic if the ωOPBEA is deterministic.

About the relations involving Muller automata, $\mathcal{L}(\omega\text{DOPBA}) \subseteq \mathcal{L}(\omega\text{DOPMA})$ derives form the fact that any ωDOPBA $\mathcal{B} = \langle \Sigma, M, Q, q_0, F, \delta \rangle$ is equivalent to an ωDOPMA $\mathcal{A} = \langle \Sigma, M, Q, q_0, \mathcal{T}, \delta \rangle$ whose acceptance component $\mathcal{T}$ consists of all subsets of Q including some final state of $\mathcal{B}$, namely $\mathcal{T} = \{P \subseteq Q \mid P \cap F \neq \emptyset\}$; the last relation is obvious. □

In the case of classical finite-state automata on infinite words, nondeterministic Büchi automata and nondeterministic Muller automata are equivalent and define the class of ω-regular languages. Traditionally, Muller automata have been introduced to provide an adequate acceptance mode for deterministic automata on ω-words. In fact, deterministic Büchi automata cannot recognize all ω-regular languages, whereas deterministic Muller automata are equivalent to nondeterministic Büchi ones [96].

For VPAs on infinite words, instead, the paper [11] showed that the classical determinization algorithm of Büchi automata into deterministic Muller automata is no longer valid, and deterministic Muller ωVPAs are strictly less powerful than nondeterministic Büchi ωVPAs. A similar relationship holds for ωOPAs too.

Theorem 3.2. *$\mathcal{L}(\omega OPBA) = \mathcal{L}(\omega OPMA)$.*

Proof. Each ωOPBA is equivalent to an ωOPMA having the same underlying OPA and acceptance component $\mathcal{T}$ consisting of all subsets of states including some final state of B (as for their deterministic counterpart, see proof of Proposition 3.1).

Conversely, any ω-language recognized by an ωOPMA $\mathcal{A} = \langle \Sigma, M, Q, I, \mathcal{T}, \delta \rangle$ can be recognized by an ωOPBA $\mathcal{B}$ with the same precedence matrix and with $O(s2^s)$

states, where s is the number of states of $\mathcal{A}$. We can assume that $\mathcal{T}$ is a singleton. Indeed, $L(\mathcal{A})$ can be expressed as

$$L(\mathcal{A}) = \bigcup_{T \in \mathcal{T}} L(\mathcal{A}_T), \text{ where } \mathcal{A}_T = \langle \Sigma, M, Q, I, \{T\}, \delta \rangle.$$

Since $\mathcal{L}(\omega\text{OPBA})$ is closed under union (a property that will be proved later, with Theorem 3.4), if each language $L(\mathcal{A}_T)$ is accepted by an ωOPBA, then $L(\mathcal{A})$ too is accepted by an ωOPBA.

Thus, let $\mathcal{T}$ be the singleton $\{T\}$. Let us build an ωOPBA $\mathcal{B} = \langle \Sigma, M, \tilde{Q}, I, F, \tilde{\delta} \rangle$ that accepts the same language as $\mathcal{A}$ as follows. $\tilde{Q}$ includes elements of two types: states of $\mathcal{A}$, and states (q, R) where $q \in Q$ and $R \subseteq Q$ is a set (that we informally call "box"), which will be used to test whether the run of $\mathcal{A}$ is successful.

Intuitively, the automaton $\mathcal{B}$ simulates $\mathcal{A}$, reading the input string x, along a sequence of states q, and then guesses nondeterministically the point after which a successful run ρ of $\mathcal{A}$ on x stops visiting the states that occur only finitely often in the run, and ρ begins to visit all and only the states in the set T. After this point $\mathcal{B}$ switches to the states of the form (q, R) and collects in R the states visited by $\mathcal{A}$ during the run, "emptying the box" as soon as it contains exactly the set T. Every time it empties the box, $\mathcal{B}$ resumes collecting the states that $\mathcal{A}$ will visit from that point onwards. If the final states of $\mathcal{B}$ are defined as those ones when it collects exactly the set T, then $\mathcal{B}$ will visit infinitely often these final states iff $\mathcal{A}$ visits all and only the states in T infinitely often.

More formally, $\mathcal{B}$ is defined by:

- $\tilde{Q} = Q \cup (Q \times \wp(Q))$,
- $F = \{(q, T) \mid q \in T\}$,
- $\tilde{\delta} : \tilde{Q} \times (\Sigma \cup \tilde{Q}) \to \wp(\tilde{Q})$, where the push function is defined by:
  - $\tilde{\delta}_{\text{push}}(q, a) = \delta_{\text{push}}(q, a) \cup \{\langle p, \{p\} \rangle \mid p \in \delta_{\text{push}}(q, a)\} \quad \forall q \in Q, a \in \Sigma$
  - $\tilde{\delta}_{\text{push}}(\langle q, R \rangle, a) = \begin{cases} \{\langle p, R \cup \{p\} \rangle \mid p \in \delta_{\text{push}}(q, a)\} & \text{if } R \neq T \\ \{\langle p, \{p\} \rangle \mid p \in \delta_{\text{push}}(q, a)\} & \text{if } R = T \end{cases}$
    $\forall q \in Q, R \subseteq Q, a \in \Sigma$.

The shift function is defined analogously.

The pop function $\tilde{\delta}_{\text{pop}} : \tilde{Q} \times \tilde{Q} \to \wp(\tilde{Q})$ is defined by:

- $\tilde{\delta}_{\text{pop}}(q_1, q_2) = \delta_{\text{pop}}(q_1, q_2) \cup \{\langle p, \{p\}\rangle \mid p \in \delta_{\text{pop}}(q_1, q_2)\}$, $\forall q_1, q_2 \in Q$
- $\tilde{\delta}_{\text{pop}}(\langle q_1, R\rangle, q_2) = \begin{cases} \{\langle p, R \cup \{p\}\rangle \mid p \in \delta_{\text{pop}}(q_1, q_2)\} & \text{if } R \neq T \\ \{\langle p, \{p\}\rangle \mid p \in \delta_{\text{pop}}(q_1, q_2)\} & \text{if } R = T \end{cases}$
- $\tilde{\delta}_{\text{pop}}(\langle q_1, R_1\rangle, \langle q_2, R_2\rangle) = \begin{cases} \{\langle p, R_1 \cup \{p\}\rangle \mid p \in \delta_{\text{pop}}(q_1, q_2)\} & \text{if } R_1 \neq T \\ \{\langle p, \{p\}\rangle \mid p \in \delta_{\text{pop}}(q_1, q_2)\} & \text{if } R_1 = T \end{cases}$
  $\forall q_1, q_2 \in Q, R, R_1, R_2 \subseteq Q$.

First, we show that $L(\mathcal{A}) \subseteq L(\mathcal{B})$. Let $x \in L(\mathcal{A})$, and let ρ be a successful run on x. There exists a finite prefix $v \in \Sigma^*$ of $x = vu_1u_2\ldots$ such that the infinite path followed by $\mathcal{A}$ after reading v (i.e., on the infinite word $u_1u_2\ldots$) visits all and only states in T infinitely often. Thus, the run ρ can be written as:

$$\rho = \langle \alpha_0 = \bot,\ q_0,\ x = vu_1u_2\ldots\rangle \overset{*}{\vdash} \langle \alpha_{|v|},\ q_{|v|},\ u_1u_2\ldots\rangle \overset{+}{\vdash} \ldots \overset{+}{\vdash} \langle \alpha_i,\ q_i,\ u_i\ldots\rangle \overset{+}{\vdash} \ldots$$

where $\{q_i \mid i > |v|\} = T$ and $q_0 \in I$. Then, there is a successful run $\tilde{\rho}$ of $\mathcal{B}$ on the same word, which follows singleton states of $\mathcal{A}$ while it reads v

$$\tilde{\rho} = \langle \beta_0 = \alpha_0 = \bot,\ q_0,\ x = vu_1u_2\ldots\rangle \overset{*}{\vdash} \langle \beta_{|v|} = \alpha_{|v|},\ q_{|v|},\ u_1u_2\ldots\rangle$$

and then switches to states augmented with a box: $\langle \beta_{|v|} = \alpha_{|v|},\ q_{|v|},\ u_1u_2\ldots\rangle \vdash \langle \beta_{|v|+1} = \alpha_{|v|+1},\ \langle p, \{p\}\rangle,\ \tilde{u}_1u_2\ldots\rangle$, where $\langle \alpha_{|v|},\ q_{|v|},\ u_1u_2\ldots\rangle \vdash \langle \alpha_{|v|+1},\ p,\ \tilde{u}_1u_2\ldots\rangle$ and $u_1 = a\tilde{u}_1$.

Since after this point $\mathcal{A}$ visits each state in T and only these states infinitely often, $\mathcal{B}$ will reach infinitely often final states $(q, T) \in F$, emptying infinitely often its box as soon as it gets full, and resuming the collection of states therein with the subsequent state in the run.

Conversely, we show that $L(\mathcal{B}) \subseteq L(\mathcal{A})$. Let $x \in \Sigma^\omega$ be an infinite word in $L(\mathcal{B})$. Define the projection $\pi : Q \cup (Q \times \wp(Q)) \to Q$ as $\pi(q) = q$ and $\pi(\langle q, R\rangle) = q, \forall q \in$

$Q, R \subseteq Q$. Given a run $\tilde{\rho}$ of the automaton $\mathcal{B}$, let $\pi(\tilde{\rho})$ be the natural extension of π on a run.

By construction, if $\tilde{\rho}$ is a run of $\mathcal{B}$ on an ω-word, then $\pi(\tilde{\rho}) = \rho$ is a run for $\mathcal{A}$ on the same word.
Now, let $\tilde{\rho}$ be a successful run for $\mathcal{B}$ on x; $\rho = \pi(\tilde{\rho})$ is a run for $\mathcal{A}$ on x. Since only the states augmented with a box are final states, then after a sequence (possibly empty) of singleton states initially traversed by $\mathcal{B}$, the automaton will definitively visit only states of the form (q, R) (in fact, no singleton state is reachable again from these states).

By induction on the number of final states reached by $\mathcal{B}$ along its run, it can be proved that, for each pair of final states consecutively reached by $\mathcal{B}$, say (q_{F_i}, R_i) and $(q_{F_{i+1}}, R_{i+1})$, the portion of the run visited between them, say $\tilde{\rho}_i$, is such that the set of states reached along $\pi(\tilde{\rho}_i)$ equals exactly T. Finally, since final states in $\tilde{\rho}$ are visited infinitely often, the run $\pi(\tilde{\rho})$ is successful for $\mathcal{A}$. □

3.2.2 *Strict inclusion and incomparability results*

To prove the strict inclusion and incomparability relations summarized in Figure 19, we introduce some simple examples of ω-languages, whose membership properties are summarized in Table 1.

1. For $\Sigma = \{a, b\}$, $L_{a\infty} = \{x \in \Sigma^\omega$: x contains an infinite number of occurrences of letter $a\}$ is recognized by the ωDOPBEA depicted in Figure 20.

	a	b
a	$\gtrdot$	$\gtrdot$
b	$\gtrdot$	$\gtrdot$
$\#$	$\lessdot$	$\lessdot$

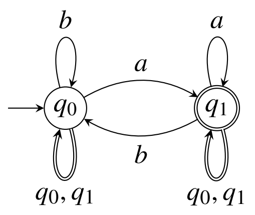


Figure 20.: ωDOPBEA, with its OPM, for $L_{a\infty}$.

	$\mathcal{L}(\omega\text{DOPBEA})$	$\mathcal{L}(\omega\text{OPBEA})$	$\mathcal{L}(\omega\text{DOPBA})$	$\mathcal{L}(\omega\text{DOPMA})$	$\mathcal{L}(\omega\text{OPBA})$	$\mathcal{L}(\omega\text{DBVPA})$	$\mathcal{L}(\omega\text{BVPA})$
$L_{a-\text{finite}}$	$\notin$	$\in$	$\notin$	$\in$		$\notin$	$\in$
$L_{a\infty}$	$\in$		$\in$				
$L_{\omega\text{Dyck-pr}(c,r)}$	$\in$	$\in$		$\in$			
L_{repbsd}	$\notin$	$\notin$		$\notin$	$\in$		$\in$
$L_{a2abseq}$	$\notin$	$\notin$	$\in$	$\in$	$\in$		
L_{abseq}^{ω}		$\in$					
$L_{\text{interrupt}}$	$\in$	$\in$	$\in$	$\in$	$\in$	$\notin$	$\notin$

Table 1.: Membership properties of some ω-languages, proved in Section 3.2.2 or consequences of inclusion relations proved in previous sections. The table displays only the relations needed to prove the results in this and the following section.

2. $L_{a-\text{finite}} = \{x \in \Sigma^\omega : x \text{ contains a finite number of occurrences of } a\}$, i.e., the complement of $L_{a\infty}$, is clearly recognized by an ωDOPMA and by an ωOPBEA, but cannot be recognized by any ωDOPBA. The proof of this latter fact resembles the classical proof (see [96]) that deterministic Büchi automata are strictly weaker than nondeterministic Büchi ones.

3. For $\Sigma = \{c, r\}$, let $L_{\omega\text{Dyck-pr}(c,r)}$ be the language of ω-words composed by an infinite sequence of finite-length words belonging to the Dyck language with pair c, r with possibly *pending returns*, i.e. letters r not matched by any previous corresponding letter c. $L_{\omega\text{Dyck-pr}(c,r)}$ is recognized by the ωDOPMA and the ωDOPBEA whose state graph is depicted in Figure 21 and with acceptance component defined, respectively, by the table $\mathcal{T} = \{\{q_0\}, \{q_0, q_1\}\}$ and the set of final states $F = \{q_0\}$.

4. For $\Sigma = \{c, r\}$, let L_{repbsd} be the language (studied in [11]) consisting of ω-words x on Σ such that x has only finitely many *pending calls*, i.e. occurrences of letter c not matched by any subsequent corresponding letter r (*repbsd* stands

$$\begin{array}{c|cc} & c & r \\ \hline c & \lessdot & \doteq \\ r & \gtrdot & \gtrdot \\ \# & \lessdot & \lessdot \end{array}$$

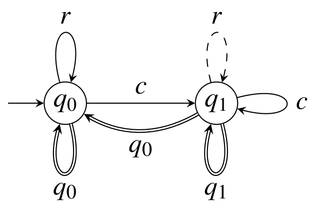


Figure 21.: ωDOPMA and ωDOPBEA recognizing $L_{\omega\text{Dyck-pr}(c,r)}$.

for repeatedly bounded stack depth). L_{repbsd} is accepted by an ωOPBA, but cannot be accepted by any ωOPBEA.

Intuitively, an ωOPBEA accepts a word iff it reaches infinitely often a final configuration with empty stack reading the input string; however, the automaton is never able to remove all the input symbols piled on the stack since it cannot pop the pending calls interspersed among the correctly nested letters c, otherwise it would either introduce conflicts in the OPM or it would not be able to verify that they are in finite number.

More formally, assume by contradiction that there is an ωOPBEA $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ recognizing L_{repbsd}. M must satisfy the following constraints: since

- $r^\omega \in L_{repbsd}$, then $M_{\# r} = \{\lessdot\}$ and $M_{rr} = \{\gtrdot\}$,
- $cr^\omega \in L_{repbsd}$, then $M_{\# c} = \{\lessdot\}$, and either $M_{cr} = \{\gtrdot\}$ or $M_{cr} = \{\doteq\}$,
- $r(cr)^\omega \in L_{repbsd}$, thus if $c \doteq r$, $M_{rc} = \{\gtrdot\}$,
- $c(cr)^\omega \in L_{repbsd}$, thus if $c \doteq r$, $M_{cc} \neq \{\lessdot\}$

Hence, M must comply with one of the matrices M_1 or M_2 shown in Figure 22.

Let $w = crc^2r^2c^3r^3 \dots c^n r^n \dots \in L_{repbsd}$ and let ρ be an accepting run of $\mathcal{A}$ on w starting from a state $q_0 \in I$. The proof that $L_{repbsd} \notin \mathcal{L}(\omega\text{OPBEA})$ is based on the two straightforward remarks:

M_1	c	r
c	$\gtrdot$	$\doteq$
r	$\gtrdot$	$\gtrdot$
$\#$	$\lessdot$	$\lessdot$

M_2	c	r
c	$\circ$	$\gtrdot$
r	$\circ$	$\gtrdot$
$\#$	$\lessdot$	$\lessdot$

Figure 22.: Matrices for L_{repbsd}, where $\circ \in \{\lessdot, \gtrdot, \doteq\}$.

- If, along a run, an ωOPA (or also an OPA) reaches a state with an empty stack, the subsequent suffix of the run does not depend on the transitions performed until that state.
- Since Q is finite, there exist $p, q \in Q$, and an infinite set of indexes $E \subseteq \mathbb{N} \setminus \{0, 1, 2\}$ such that, for each $i \in E$, ρ has a prefix: $q_0 \overset{v_i}{\rightsquigarrow} p \overset{w_i}{\rightsquigarrow} q$, where $v_i = c^1 r^1 \ldots c^{i-2} r^{i-2} c^{i-1} r^{i-2}$ and $w_i = rc^i r$ and, given the precedence relations in M_1 and M_2, both p and q are reached with an empty stack, just before performing a push move while reading the letter r in w that follows, respectively, v_i and w_i. For each $i \in E$, let ρ_i be the finite factor of ρ given by $p \overset{w_i}{\rightsquigarrow} q$.

Let $J \subseteq E$ be the set of indexes in E such that, $\forall i \in J$, ρ_i visits a final state with empty stack. We can build a run ρ', which differs from ρ in that

- for every $i \in E \setminus J$, the factor ρ_i is replaced by a ρ_j for some $j \in E$, with $j > i$,
- for every $i \in J$, the factor ρ_i is replaced by a ρ_j with $i < j \in J$ if $|J| = \infty$, or $i < j \in E$ if $|J| < \infty$.

ρ' is an accepting run in $\mathcal{A}$, along which the automaton reads a word with infinitely many pending calls, which does not belong to L_{repbsd}, and this is a contradiction.

Furthermore, L_{repbsd} is not recognizable by any ωDOPMA. The proof of this fact resembles the analogous proof in [11]; indeed, that proof is essentially based on topological properties of the state-graph of the automata and it is general enough to adapt to both ωVPAs and ωOPAs.

5. For $\Sigma = \{a, b\}$, let $L_{abseq} = \{a^k b^k \mid k \geq 1\}$ and $L_{a2abseq} = \{x \in \Sigma^\omega \mid x = a^2 L^\omega_{abseq}\}$. Language $L_{a2abseq}$ is recognized by an ωDOPBA, but it is not recognized by any ωOPBEA (nor a fortiori by any ωDOPBEA).

   Indeed, words in L_{abseq} can be recognized only with the OPM M depicted in Figure 23: any other OPM will prevent verifying that the number of as equals that of bs in subwords belonging to L_{abseq}. Since $a \lessdot a$, an ωOPBEA piles up on the stack the first sequence a^2 of a word and cannot remove it afterwards; hence it cannot empty the stack infinitely often to accept a string in $L_{a2abseq}$. There is, however, an ωDOPBA (and thus an ωDOPMA) that recognizes such a language: it is shown in Figure 23. Notice also that L^ω_{abseq} can be recognized by an ωOPBEA, with OPM M and with state graph depicted in Figure 23 but with state q_2 instead of q_0 as initial state.

	a	b
a	$\lessdot$	$\doteq$
b	$\gtrdot$	$\gtrdot$
$\#$	$\lessdot$	

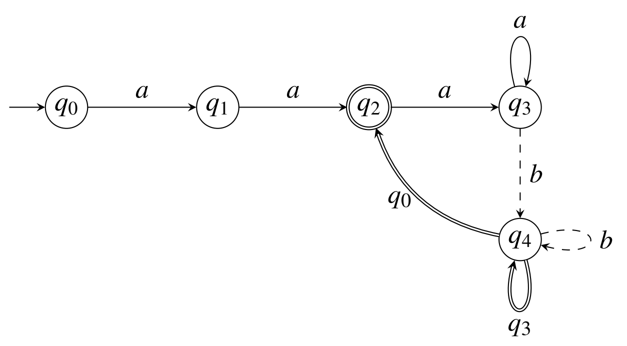


Figure 23.: An ωDOPBA recognizing language $L_{a2abseq}$.

3.3 CLOSURE PROPERTIES

Table 2 displays the closure properties of the various families of ω-languages. In order to prove them, we first introduce some preliminary constructions in Section 3.3.1. Then in Section 3.3.2 we present the proofs for $\mathcal{L}(\omega\text{OPBA})$; in particular closure under complement and concatenation are the cases that require novel investigation techniques w.r.t. previous literature. In Section 3.3.3 we prove the closure properties for other classes of ωOPA.

	$\mathcal{L}(\omega\text{DOPBEA})$	$\mathcal{L}(\omega\text{OPBEA})$	$\mathcal{L}(\omega\text{DOPBA})$	$\mathcal{L}(\omega\text{DOPMA})$	$\mathcal{L}(\omega\text{OPBA})$	$\mathcal{L}(\omega\text{BVPA})$
Intersection	Yes	Yes	Yes	Yes	Yes	Yes
Union	Yes	Yes	Yes	Yes	Yes	Yes
Complement	No	No	No	Yes	Yes	Yes
$L_1 \cdot L_2$	No	No	No	No	Yes	Yes

Table 2.: Closure properties of families of ω-languages. ($L_1 \cdot L_2$ denotes the concatenation of a language of finite-length words L_1 with an ω-language L_2).

3.3.1 *Preliminary properties and constructions*

The following constructions will be exploited to prove several closure properties. Indeed, they would be useful even to provide alternative proofs of the same properties in the case of finite length languages w.r.t. those that have been introduced in literature [34, 35] by referring to OPGs rather than OPAs.

We begin by introducing the deterministic product of transition functions, defined by extending the usual construction for finite state automata. Such a definition is meaningful when applied to automata that share the same precedence matrix, because they perform the same type of move (push/shift/pop) while reading the input word.

Definition 3.6. Let Q_1 and Q_2 be two disjoint sets of states of two deterministic automata sharing the same OP alphabet and let δ_1 and δ_2 be their transition functions.

Their product state Q is defined as $Q = Q_1 \times Q_2$ and their product transition function $\delta : Q \times (\Sigma \cup Q) \to Q$ is defined as follows, where $q_1, q_2, p_1, p_2 \in Q, a \in \Sigma$:

$$\begin{aligned}
\delta_{\text{push}}((q_1, q_2), a) &= (\delta_{1\text{push}}(q_1, a), \delta_{2\text{push}}(q_2, a)) \\
\delta_{\text{shift}}((q_1, q_2), a) &= (\delta_{1\text{shift}}(q_1, a), \delta_{2\text{shift}}(q_2, a)) \\
\delta_{\text{pop}}((q_1, q_2), (p_1, p_2)) &= (\delta_{1\text{pop}}(q_1, p_1), \delta_{2\text{pop}}(q_2, p_2))
\end{aligned}$$

Clearly $|Q| = |Q_1| \cdot |Q_2|$.

Next, we introduce the definition of OP Büchi ω-transducers, which will be needed in some technical steps; other types of ω-transducers could be defined similarly.

Definition 3.7 (Operator precedence (Büchi) ω-transducer)**.** An *operator precedence ω-transducer* is defined in the usual way as a tuple $T = \langle \Sigma, M, Q, I, F, O, \delta, \eta \rangle$ where Σ, M, Q, I, F are defined as in Definition 2.7, O is a finite set of output symbols, the transition function δ and the output function η are defined by $\langle \delta, \eta \rangle : Q \times (\Sigma \cup Q) \to \wp_F(Q \times O^*)$, where $\wp_F(Q \times O^*)$ denotes the set of finite subsets of $Q \times O^*$, and $\langle \delta, \eta \rangle$ can be seen as the union of three functions, $\langle \delta_{\text{shift}}, \eta_{\text{shift}} \rangle : Q \times \Sigma \to \wp_F(Q \times O^*)$, $\langle \delta_{\text{push}}, \eta_{\text{push}} \rangle : Q \times \Sigma \to \wp_F(Q \times O^*)$ and $\langle \delta_{\text{pop}}, \eta_{\text{pop}} \rangle : Q \times Q \to \wp_F(Q \times O^*)$.

A *configuration* of the ω-transducer is denoted $\langle \beta,\ q,\ w \rangle \downarrow z$, where $C = \langle \beta,\ q,\ w \rangle$ is the configuration of the underlying ωOPBA and the string after $\downarrow$ represents the output of the automaton in the configuration. The transition relation $\vdash$ is naturally extended from ωOPBAs, by concatenating the output symbols produced at each move with those generated in the previous moves. Runs and acceptance by the transducer are defined as in the corresponding ωOPBA.

The *transduction* $\tau(x)$, $x \in \Sigma^\omega$, generated by T is the set of ω-strings produced during its nondeterministic successful runs over x.

The next statement is propaedeutic to many constructive proofs of closure properties, where the operands are in general OPAs with compatible but not identical matrices, and the result's matrix must often be the union of the two original ones. If $\mathcal{A}$ is an OPA with precedence matrix M and $M' \supseteq M$, then clearly $\mathcal{A}$ works also over M' but the language recognized by $\mathcal{A}$ over M' is not necessarily the same, since the presence of precedence relations in M' that are not included in M may allow for successful runs on some words that are, instead, not successful in the original OPA. The next statement proves, however, that the precedence matrix of an OPA can always be extended (up to completion), provided that conflict-freedom is preserved, without affecting the recognized language.

Statement 3.1 (Extended matrix normal form)**.** *Let $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ be an OPA (over finite-length or omega words) with $|Q| = s$. For any conflict-free OPM $M' \supseteq M$, there exists an OPA with OPM M' that recognizes the same language as $\mathcal{A}$ and has $O(|\Sigma|^2 s)$ states.*

Proof. First consider finite-length words. The new OPA $\mathcal{A}' = \langle \Sigma, M', Q', I', F', \delta' \rangle$ is derived from $\mathcal{A}$ in the following way:

- $Q' = \hat{\Sigma} \times Q \times \hat{\Sigma}$, where $\hat{\Sigma} = (\Sigma \cup \{\#\})$, i.e. the first component of a state is the lookback symbol, the second component is a state of $\mathcal{A}$ and the third component is the lookahead symbol,
- $I' = \{\#\} \times I \times \{a \in \hat{\Sigma} \mid M_{\#a} \neq \emptyset\}$,
- $F' = \{\#\} \times F \times \{\#\}$,
- $\delta' : Q' \times (\Sigma \cup Q') \to \wp(Q')$ is the transition function defined as follows.

  Let $a \in \hat{\Sigma}, b \in \Sigma, q \in Q$. The push transition $\delta'_{\text{push}} : Q' \times \Sigma \to \wp(Q')$ is defined by:

  $$\delta'_{\text{push}}(\langle a, q, b \rangle, b) = \{\langle b, p, c \rangle \mid p \in \delta_{\text{push}}(q, b) \wedge M_{ab} = \{\lessdot\} \wedge M_{bc} \neq \emptyset\},$$

  The shift transition $\delta'_{\text{shift}} : Q' \times \Sigma \to \wp(Q')$ is defined analogously:

  $$\delta'_{\text{shift}}(\langle a, q, b \rangle, b) = \{\langle b, p, c \rangle \mid p \in \delta_{\text{shift}}(q, b) \wedge M_{ab} = \{\doteq\} \wedge M_{bc} \neq \emptyset\},$$

  The pop transition $\delta'_{\text{pop}} : Q' \times Q' \to \wp(Q')$ is defined by:

  $$\delta'_{\text{pop}}(\langle a_1, q_1, a_2 \rangle, \langle b_1, q_2, b_2 \rangle) = \left\{ \langle b_1, q_3, a_2 \rangle \;\middle|\; \begin{array}{c} q_3 \in \delta_{\text{pop}}(q_1, q_2) \wedge \\ M_{a_1 a_2} = \{\gtrdot\} \wedge M_{b_1 a_2} \neq \emptyset \end{array} \right\},$$

where $a_1, b_2 \in \Sigma, a_2, b_1 \in \hat{\Sigma}, q_1, q_2 \in Q$.

Clearly, the OPA $\mathcal{A}'$ has OPM M' and accepts the same language as $\mathcal{A}$.

This construction can be naturally extended to ωOPAs: in particular, for ωOPBA the set of final states of $\mathcal{A}'$ is $F' = \hat{\Sigma} \times F \times \Sigma$, i.e. a run of $\mathcal{A}'$ is accepting iff it visits infinitely often final states of $\mathcal{A}$, independently of the lookback and the lookahead symbols considered for these states. For ωOPBEA this acceptance component may be further refined as $F' = \{\#\} \times F \times \Sigma$. For ωOPMA, $\mathcal{T}' = \{t \mid t = A_1 \times S \times A_2$, $S \in \mathcal{T}, A_1 \subseteq \hat{\Sigma}, A_2 \subseteq \Sigma\}$ where $\mathcal{T} \subseteq \wp(Q)$ is the table of $\mathcal{A}$. Furthermore, the transformation preserves determinism. □

OPA's version without # as lookahead

In this section we illustrate a new version of OPAs that do not rely on the end-marker # for the recognition of a finite length word.

The new model is defined by slightly modifying the semantics of the transition relation and of the acceptance condition of original OPAs, in such a way that a string is accepted by an automaton if it reaches a final state right at the end of the parsing of the whole word, and does not perform any pop move determined by the ending delimiter # to empty the stack; thus the automaton stops just after having pushed on the stack (or updated the top of the stack symbol with) the last symbol of the string.

In this alternative characterization of OPAs, the semantics of the transition relation differs from the classical definition in that, once a configuration with the end-marker as lookahead is reached, the computation cannot evolve in any subsequent configuration, i.e., a pop move $C_1 \vdash_{\not\#} C_2$ with $C_1 = \langle \Pi[a,\ p],\ q,\ x\# \rangle$ is performed only if $x \neq \varepsilon$ (where symbol $\vdash_{\not\#}$ denotes a move according to this variation of the semantics of the transition relation). The language accepted by the automaton according to this new semantics (denoted as $L_{\not\#}$) is the set of words:

$$L_{\not\#}(\mathcal{A}) = \{x \mid \langle \bot,\ q_I,\ x\# \rangle \overset{*}{\vdash}_{\not\#} \langle \bot\gamma,\ q_F,\ \# \rangle, q_I \in I, q_F \in F, \gamma \in \Gamma^*\}$$

This new version of the automaton, called *no-#-look-aheadOPA ($\not\#$OPA)* is closer to the traditional acceptance criterion of general pushdown automata; we emphasize, however, that, unlike normal acceptance by final state of a pushdown automaton, which can perform a number of ε-moves after reaching the end of a string and accepts it if just one of the visited states is final, this type of automaton cannot perform any (pop, i.e., ε-) move when it reaches the end of the input string. The following lemmata (Lemma 3.1 and Lemma 3.2) prove the equivalence between the original version of OPAs and the new one.[8]

Lemma 3.1. *Let $\mathcal{A}_1$ be a nondeterministic OPA defined on an OP alphabet (Σ, M) with s states. Then there exists a nondeterministic $\not\#$OPA $\mathcal{A}_2$ on (Σ, M) and $O(s^2)$ states such that $L(\mathcal{A}_1) = L_{\not\#}(\mathcal{A}_2)$.*

We first explain informally the rationale of the simulation of $\mathcal{A}_1$ by $\mathcal{A}_2$, with the aid of an example; then we formally define its construction and prove their equivalence.

Consider a word of finite length w compatible with M: the string $\#w$ can be factored in a unique way as a sequence of bodies of chains and pending letters as

$$\# \, w = \# \, w_1 a_1 w_2 a_2 \dots w_n a_n$$

where ${}^{a_{i-1}}[w_i]^{a_i}$ are maximal chains and each w_i can be possibly missing, with $a_0 = \#$ and $\forall i : 1 \le i \le n-1$ $a_i \lessdot a_{i+1}$ or $a_i \doteq a_{i+1}$. Let $i_j \in \{1, 2, \dots, n\}$, $1 \le j \le k$, $k \ge 1$ be indexes such that

$$\# \lessdot a_{i_1} = a_1 \doteq \dots \doteq a_{i_2-1} \lessdot a_{i_2} \doteq \dots \doteq a_{i_3-1} \lessdot a_{i_3} \doteq \dots \doteq a_{i_k-1} \lessdot a_{i_k} \doteq a_{i_k+1} \dots \doteq a_n \tag{8}$$

When reading w, the symbols of the string are progressively put on the stack, either by a push move or by a shift move, and, whenever a chain w_i is recognized, the symbol on the top of the stack is popped. Hence, after reading w the stack contains only the symbols $\#\ a_{i_2-1}\ a_{i_3-1}\ \dots\ a_n$ that are the ending symbols of the open chains in the sequence (8).

When w is read by a standard OPA, the automaton performs a series of pop moves at the end of the string due to the presence of the end delimiter #. These moves progressively empty the stack. The run is accepting if it leads to a final state after all pop moves.

8 Only Lemma 3.1 will be used in Section 3.3.2 but we include both of them for completeness.

A nondeterministic automaton that, unlike standard OPAs, does not resort to the end delimiter # for the recognition of a string must guess nondeterministically the ending point of each open chain and guess how, in an accepting run, the states in these points would be updated if the final pop moves were progressively performed. The automaton must behave as if, at the same time, it simulates two snapshots of the accepting run of a standard OPA: a move during the reading of the input, and a step during the final pop transitions which will later on empty the stack, leading to a final state. To this aim, the states of a standard OPA are augmented with an additional component.

A $\not\#$OPA $\mathcal{A}_2$ equivalent to a given OPA $\mathcal{A}_1$ thus may be defined so that, after reading each prefix of a word, it reaches a final state whenever, if the word were completed in that point with #, $\mathcal{A}_1$ could reach an accepting state with a sequence of pop moves. In this way, $\mathcal{A}_2$ can guess in advance which words may eventually lead to an accepting state of $\mathcal{A}_1$, without having to wait until reading the delimiter # and to perform final pop moves. In other words, it simulates the possible look-ahead of the # delimiter. Before going into the details of the construction, the following example illustrates the above intuitive description.

Example 3.2. We refer to the computation of the OPA in Example 2.3. Consider the input word of this computation without the end-marker #. The sequence of pending letters in the input word corresponds to three open chains, according to sequence (8), with starting symbols $+$, $\times$, $\llparenthesis$, respectively.

Figure 24 shows the configuration just before looking ahead at the symbol #. The states depicted within a box are those placeholders that an equivalent $\not\#$OPA should fill up to guess in advance the last pop moves $q_3 = \boxed{q_3} \overset{q_0}{\Longrightarrow} \boxed{q_3} \overset{q_1}{\Longrightarrow} \boxed{q_3} \overset{q_1}{\Longrightarrow} \boxed{q_3 \in F_1}$ of the accepting run.

$$\begin{tikzcd}
\langle \bot & {[+, q_1]} & {[\times, q_1]} & {[\llparenthesis, q_0]} & {,\; q_3} & {,\ \#\rangle} \\
& \boxed{q_3 \in F_1} & \boxed{q_3} \arrow[l, bend right] & \boxed{q_3} \arrow[l, bend right] & \boxed{q_3} \arrow[l, bend right] &
\end{tikzcd}$$

Figure 24.: Configuration of the OPA of Example 2.3 just before looking ahead at #.

The corresponding configuration of the $\not\#$OPA is depicted in Figure 25.

$$\langle \perp \quad [+, \langle q_1, \boxed{q_3} \rangle] \quad [\times, \langle q_1, \boxed{q_3} \rangle] \quad [(\!|, \langle q_0, \boxed{q_3} \rangle] \quad , \quad \langle q_3, \boxed{q_3} \rangle \quad , \quad \# \rangle$$

Figure 25.: Configuration of the $\not\#$OPA described in Example 3.2.

We now proceed with the construction of $\mathcal{A}_2$ and the proof of its equivalence with $\mathcal{A}_1$.

Proof. of Lemma 3.1
Let $\mathcal{A}_1$ be $\langle \Sigma, M, Q_1, I_1, F_1, \delta_1 \rangle$ and define $\mathcal{A}_2 = \langle \Sigma, M, Q_2, I_2, F_2, \delta_2 \rangle$ as follows.

- $Q_2 = \{B, Z, U\} \times Q_1 \times Q_1$.
  Hence, a state $\langle x, q, p \rangle$ of $\mathcal{A}_2$ is a tuple whose first component denotes a nondeterministic guess for the next input symbol to be read, i.e., whether it is a pending letter which is the initial symbol of an open chain (Z), or a pending letter within an open chain other than the first one (U), or a symbol within a maximal chain (B). The second component of a state represents the current state q in $\mathcal{A}_1$. To illustrate the meaning of the last component, consider an accepting run of $\mathcal{A}_1$ and let q be its current state just before a push move to be performed when reading the first symbol of an open chain; also, let r be the state reached by such push move and s be the state of the automaton when the stack element pushed by this move (possibly updated by subsequent shifts) is going to be popped leading to a state p. Then, in the same position of the corresponding run of $\mathcal{A}_2$, the current state would be $\langle Z, q, p \rangle \in Q_2$ and state $\langle x, r, s \rangle \in Q_2$ will be reached by $\mathcal{A}_2$ (x being nondeterministically anyone of B, Z, U); in other words, the last component p represents a guess about the state that will be reached in $\mathcal{A}_1$ when the stack element pushed by this move will be popped. Hence we can consider only states $\langle Z, q, p \rangle \in Q_2$ such that $s \overset{q}{\Longrightarrow} p$ in $\mathcal{A}_1$ for some $s \in Q_1$. In all the other positions the last component is simply propagated.

  For instance, Figure 26 shows an accepting run on the word $n + n \times (\!| n + n |\!)$ of a $\not\#$OPA that is equivalent to the OPA of Example 2.3. Note that before reading

stack	state	current input
$\perp$	$\langle B, q_0, q_3\rangle$	$n + n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[n,\ \langle B, q_0, q_3\rangle]$	$\langle B, q_1, q_3\rangle$	$+ n \times \llparenthesis n + n \rrparenthesis \#$
$\perp$	$\langle Z, q_1, q_3\rangle$	$+ n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle]$	$\langle B, q_0, q_3\rangle$	$n \times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][n,\ \langle B, q_0, q_3\rangle]$	$\langle B, q_1, q_3\rangle$	$\times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle]$	$\langle Z, q_1, q_3\rangle$	$\times \llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle]$	$\langle Z, q_0, q_3\rangle$	$\llparenthesis n + n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle]$	$\langle B, q_2, q_3\rangle$	$n + n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle][n,\ \langle B, q_2, q_3\rangle]$	$\langle B, q_3, q_3\rangle$	$+ n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle]$	$\langle B, q_3, q_3\rangle$	$+ n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle][+,\ \langle B, q_3, q_3\rangle]$	$\langle B, q_2, q_3\rangle$	$n \rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle][+,\ \langle B, q_3, q_3\rangle][n,\ \langle B, q_2, q_3\rangle]$	$\langle B, q_3, q_3\rangle$	$\rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle][+,\ \langle B, q_3, q_3\rangle]$	$\langle B, q_3, q_3\rangle$	$\rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\llparenthesis,\ \langle Z, q_0, q_3\rangle]$	$\langle U, q_3, q_3\rangle$	$\rrparenthesis \#$
$\perp[+,\ \langle Z, q_1, q_3\rangle][\times,\ \langle Z, q_1, q_3\rangle][\rrparenthesis,\ \langle Z, q_0, q_3\rangle]$	$\langle Z, q_3, q_3\rangle$	$\#$

Figure 26.: Example of an accepting computation for the word $n + n \times \llparenthesis n + n \rrparenthesis$ of a $\not\#$OPA that is equivalent to the OPA of Example 2.3.

the $\llparenthesis$, which is the beginning of an open chain, the automaton is in the state $\langle Z, q_0, q_3\rangle$ and then moves to $\langle B, q_2, q_3\rangle$ guessing the state that is reached by the pop move that occurs in the corresponding run of the OPA after reading the $\rrparenthesis$. Before reading the second n, which is the body of a maximal chain, instead, the automaton is in state $\langle B, q_0, q_3\rangle$ and, after popping n from the stack, moves to $\langle Z, q_1, q_3\rangle$ since the following $\times$ is the beginning of an open chain.

- $I_2 = \{\langle x, q, q_F\rangle \mid x \in \{Z, B\}, q \in I_1, q_F \in F_1\}$
- $F_2 = \{\langle Z, q, q\rangle \mid q \in Q_1\}$
- The transition function is defined as the union of three functions.

  The push transition function $\delta_{2\text{push}} : Q_2 \times \Sigma \rightarrow \wp(Q_2)$ is defined as follows, where $p, q, r, s \in Q_1$, $a \in \Sigma$.

  – *Pending letter at the beginning of an open chain.*

$$\delta_{2\text{push}}(\langle Z, q, p\rangle, a) = \left\{\langle x, r, s\rangle \mid x \in \{B, Z, U\}, r \in \delta_{1\text{push}}(q, a), s \overset{q}{\Longrightarrow} p \text{ in } \mathcal{A}_1\right\}$$

– *Symbol of a maximal chain.*

$$\delta_{2\text{push}}\left(\langle B, q, p\rangle, a\right) = \left\{\langle B, r, p\rangle \mid r \in \delta_{1\text{push}}(q, a)\right\}$$

The shift transition function $\delta_{2\text{shift}} : Q_2 \times \Sigma \to \wp(Q_2)$ is defined as follows:

– *Pending letter within an open chain.*

$$\delta_{2\text{shift}}\left(\langle U, q, p\rangle, a\right) = \{\langle x, r, p\rangle \mid x \in \{B, Z, U\}, r \in \delta_{1\text{shift}}(q, a)\}$$

– *Symbol of a maximal chain.*

$$\delta_{2\text{shift}}\left(\langle B, q, p\rangle, a\right) = \{\langle B, r, p\rangle \mid r \in \delta_{1\text{shift}}(q, a)\}$$

Notice that the second component of the states computed by $\delta_{2\text{push}}$ and $\delta_{2\text{shift}}$ is independent of the first component of the starting state.

The pop transition function $\delta_{2\text{pop}} : Q_2 \times Q_2 \to \wp(Q_2)$ can be executed only within a maximal chain since there is no pop determined by the ending delimiter:

$$\delta_{2\text{pop}}\left(\langle B, q, s\rangle, \langle B, p, s\rangle\right) = \left\{\langle x, r, s\rangle \mid x \in \{B, Z, U\}, q \overset{p}{\Longrightarrow} r \text{ in } \mathcal{A}_1\right\}$$

All other moves lead to an error state.

Let us prove first $L(\mathcal{A}_1) \subseteq L_{\#}(\mathcal{A}_2)$. Consider a word $w \in L(\mathcal{A}_1)$. Then there exists a support $q \overset{w}{\rightsquigarrow} q'$ in $\mathcal{A}_1$ with $q \in I_1$ and $q' \in F_1$. If $w = w_1 a_1 w_2 a_2 \dots w_n a_n$ where a_i are pending letters and w_i are maximal chains, let k be the number of open chains determined by the sequence of pending letters in w according to the structure (8), and let $a_{i_1} = a_1, a_{i_2}, \dots, a_{i_k}$ be their initial symbols. Also, for every $i = 2, \dots, n$, let $t(i)$ be the greatest index t such that $i_t < i$, i.e., a_i is within the $t(i)$-th open chain beginning with $a_{i_{t(i)}}$. In particular, for $i = n$, if $a_{n-1} \lessdot a_n$ then $i_k = n$, otherwise $t(n) = k$. As a notational convention, denote by $\longmapsto$ a move that can be either a push or a shift.

Then the above support for w can be decomposed as

$$q = \widetilde{q}_0 \overset{w_1}{\leadsto} q_1 \xrightarrow{a_1} \widetilde{q}_1 \overset{w_2}{\leadsto} q_2 \overset{a_2}{\longmapsto} \dots \overset{w_n}{\leadsto} q_n \overset{a_n}{\longmapsto} \widetilde{q}_n = p_k \tag{9}$$

$$\widetilde{q}_n = p_k \overset{q_{i_k}}{\Longrightarrow} p_{k-1} \overset{q_{i_{k-1}}}{\Longrightarrow} p_{k-2} \Longrightarrow \dots \Longrightarrow p_2 \overset{q_{i_2}}{\Longrightarrow} p_1 \overset{q_{i_1}=q_1}{\Longrightarrow} p_0 = q',$$

where $q_i = \widetilde{q}_{i-1}$ if $w_i = \varepsilon$ for $i = 1, 2, \dots, n$. Notice that, for every t, q_{i_t} is the state reached in this path before the push move that pushes symbol a_{i_t} on the stack; moreover, when the last symbol in the open chain beginning with a_{i_t} is to be popped, the current state is p_t and then the symbol on the top of the stack (whose state component is q_{i_t}) is removed and $\mathcal{A}_1$ moves to state p_{t-1}.

Starting with state $\langle Z, q_1, p_0\rangle$ if $w_1 = \varepsilon$ or with $\langle B, \widetilde{q}_0, p_0\rangle \overset{w_1}{\leadsto} \langle Z, q_1, p_0\rangle$ if $w_1 \neq \varepsilon$, an accepting computation of $\mathcal{A}_2$ can be built on the basis of the following facts:

- Since $\mathcal{A}_1$ performs $q_1 \xrightarrow{a_1} \widetilde{q}_1$ and $p_1 \overset{q_1}{\Longrightarrow} p_0$, then $\delta_{2\text{push}}(\langle Z, q_1, p_0\rangle, a_1) \ni \langle x, \widetilde{q}_1, p_1\rangle$ in $\mathcal{A}_2$ for $x \in \{B, Z, U\}$. This is a push move that can be applied at the beginning of the first open chain, a_1, where p_1 is the guess about the state that will be reached before the stack symbol pushed on the stack by this move will be popped.

- In general, for every t, since $\mathcal{A}_1$ executes $q_{i_t} \xrightarrow{a_{i_t}} \widetilde{q}_{i_t}$ and $p_t \overset{q_{i_t}}{\Longrightarrow} p_{t-1}$, then $\delta_2(\langle Z, q_{i_t}, p_{t-1}\rangle, a_{i_t}) \ni \langle x, \widetilde{q}_{i_t}, p_t\rangle$ for $x \in \{B, Z, U\}$. This is a push move that can be applied at the beginning of the t-th open chain, i.e. when reading a_{i_t}, where p_t is the guess about the state that will be reached before the stack symbol with the last letter of the chain will be popped. In particular, if $i_k = n$, we can reach state $\langle Z, \widetilde{q}_n, p_k\rangle$ which is final in $\mathcal{A}_2$ since $q_n = p_k$.

- For every maximal chain w_i of w (with $i \geq 2$) consider its support $\widetilde{q}_{i-1} \overset{w_i}{\leadsto} q_i$ in the sequence (9). Then in $\mathcal{A}_2$ we have a sequence of moves starting from a state $\langle B, \widetilde{q}_{i-1}, p_{t(i)}\rangle$ and reading w_i, that ends in $\langle x, q_i, p_{t(i)}\rangle$, where $x \in \{U, Z\}$. Notice that the last component of the states does not change because we are within a maximal chain. During the reading of w_i, the last component is equal to $p_{t(i)}$, as guessed by the push move at the beginning of the current open chain.

- For every $i \notin \{i_1, i_2, \dots, i_k\}$, since $\delta_{1\text{shift}}(q_i, a_i) \ni \widetilde{q}_i$, then $\delta_{2\text{shift}}(\langle U, q_i, p_{t(i)}\rangle, a_i)$ contains $\langle x, \widetilde{q}_i, p_{t(i)}\rangle$, for $x \in \{B, Z, U\}$. In particular, if $n \neq i_k$, then $t(n) = k$ and for $i = n$ we can reach state $\langle Z, \widetilde{q}_n, p_k\rangle$ which is final in $\mathcal{A}_2$, since $\widetilde{q}_n = p_k$.

Thus, by composing in the right order the previous moves, one can obtain an accepting computation for w in $\mathcal{A}_2$.

Conversely, to prove that $L_{\#}(\mathcal{A}_2) \subseteq L(\mathcal{A}_1)$, consider a word $w \in L_{\#}(\mathcal{A}_2)$. This means that there exists a successful run of $\mathcal{A}_2$ on w. Let w be factorized as above; then the accepting run for w can be decomposed as

$$\pi_0 \overset{w_1}{\rightsquigarrow} \rho_1 \xrightarrow{a_1} \pi_1 \overset{w_2}{\rightsquigarrow} \rho_2 \dots \rho_i \xmapsto{a_i} \pi_i \overset{w_{i+1}}{\rightsquigarrow} \dots \overset{w_n}{\rightsquigarrow} \rho_n \xmapsto{a_n} \pi_n$$

where $\pi_i, \rho_i \in Q_2$, $\rho_i = \pi_{i-1}$ if $w_i = \varepsilon$, $\pi_0 \in I_2$ and $\pi_n \in F_2$. By projecting this path on the second component of states π_i and ρ_i (let them respectively be p_i and $r_i \in Q_1$), we obtain a path in $\mathcal{A}_1$ labelled by w. This path is not accepting because there are symbols left on the stack that need to be popped, but we can complete this path arguing by induction on the structure of maximal chains according to the definition of δ_2. Precisely, one can verify that Q_1 contains suitable states p_i (for $0 \leq i \leq n$), r_i (for $1 \leq i \leq n$), s_t (for $1 \leq t \leq k$), with $r_i = p_{i-1}$ whenever $w_i = \varepsilon$, such that the following facts hold.

- $\pi_0 \in I_2$, hence $\pi_0 = \langle x_0, p_0, s_0\rangle$, with $p_0 \in I_1$ and $s_0 \in F_1$; x_0 is B if $w_1 \neq \varepsilon$, otherwise $x_0 = Z$.
- $\pi_0 \overset{w_1}{\rightsquigarrow} \rho_1$ in $\mathcal{A}_2$ implies that the last component of state π_0 is propagated through chain w_1 without change; hence $\rho_1 = \langle Z, r_1, s_0\rangle$ with $p_0 \overset{w_1}{\rightsquigarrow} r_1$ in $\mathcal{A}_1$.
- $\rho_1 \xrightarrow{a_1} \pi_1$ is a push move of $\mathcal{A}_2$ at the beginning of an open chain, and this implies that the last component of π_1 is a guess on the state from which $\mathcal{A}_1$ would perform the corresponding pop, so that $\pi_1 = \langle x_1, p_1, s_1\rangle$ with $r_1 \xrightarrow{a_1} p_1$ and $s_1 \overset{r_1}{\Longrightarrow} s_0$ in $\mathcal{A}_1$; the first component is $x_1 = B$ if $w_2 \neq \varepsilon$ otherwise x_1 equals Z or U according to whether a_2 starts an open chains or not, respectively,
- The pop moves within $\pi_i \overset{w_{i+1}}{\rightsquigarrow} \rho_{i+1}$ for $1 \leq i < i_2$, and the shift moves within an open chain $\rho_i \overset{a_i}{\dashrightarrow} \pi_i$ for $1 < i < i_2$ propagate with no change the last

component. Hence $\rho_i = \langle U, r_i, s_1\rangle$ and $\pi_i = \langle x_i, p_i, s_1\rangle$ with $p_{i-1} \overset{w_i}{\rightsquigarrow} r_i \xrightarrow{a_i} p_i$ in $\mathcal{A}_1$. The first component is $x_i = B$ if $w_i \neq \varepsilon$, otherwise $x_i = Z$ for $i = i_2 - 1$, and $x_i = U$ in the other cases.

- $\rho_{i_2} \xrightarrow{a_{i_2}} \pi_{i_2}$ is a push move of $\mathcal{A}_2$ at the beginning of an open chain, and this implies that the last component of π_{i_2} is a guess on the state from which $\mathcal{A}_1$ would perform the corresponding pop, so that $\pi_{i_2} = \langle x_{i_2}, p_{i_2}, s_2\rangle$ with $r_{i_2} \xrightarrow{a_{i_2}} p_{i_2}$ and $s_2 \overset{r_{i_2}}{\Longrightarrow} s_1$ in $\mathcal{A}_1$. The first component is $x_{i_2} = B$ if $w_{i_2} \neq \varepsilon$ otherwise x_1 equals Z or U according to whether $a_{i_2} + 1$ begins an open chains or not, respectively.
- Similarly for the following moves in the run.

In general, we get

$$\begin{aligned}
\rho_i &= \langle y_i, r_i, s_{t(i)}\rangle && \text{for every } i = 1, 2, \ldots, n,\\
\pi_i &= \langle x_i, p_i, s_{t(i)}\rangle && \text{for every } i \notin \{i_1, i_2, \ldots, i_k\},\\
\pi_{i_t} &= \langle x_{i_t}, p_{i_t}, s_t\rangle && \text{for every } t = 1, 2, \ldots, k,\\
\text{with } r_i \overset{a_i}{\longmapsto} p_i,\ s_t &\overset{r_{i_t}}{\Longrightarrow} s_{t-1},\ p_{i-1} \overset{w_i}{\rightsquigarrow} r_i \text{ in } \mathcal{A}_1 &&\\
\text{and } y_i \in \{Z, U\}, x_i &\in \{B, Z, U\} && \text{for every } i \text{ and } t.
\end{aligned}$$

For $i = n$ we have $n = i_k$ or $t(n) = k$, hence $\pi_n = \langle x_n, p_n, s_k\rangle$, and $p_n = s_k$ and $x_n = Z$ since $\pi_n \in F_2$. Thus, in $\mathcal{A}_1$ there is an accepting run

$$I_1 \ni p_0 \overset{w_1}{\rightsquigarrow} r_1 \xrightarrow{a_1} p_1 \overset{w_2}{\rightsquigarrow} r_2 \ldots r_i \overset{a_i}{\longmapsto} p_i \overset{w_{i+1}}{\rightsquigarrow} \ldots \overset{w_n}{\rightsquigarrow} r_n \overset{a_n}{\longmapsto} p_n = s_k$$

$$p_n = s_k \overset{r_{i_k}}{\Longrightarrow} s_{k-1} \overset{r_{i_{k-1}}}{\Longrightarrow} s_{k-2} \Longrightarrow \ldots \Longrightarrow s_2 \overset{r_{i_2}}{\Longrightarrow} s_1 \overset{r_{i_1} = r_1}{\Longrightarrow} s_0 \in F_1.$$

□

The next lemma completes the proof of equivalence between OPAs and ⋕OPAs.

Lemma 3.2. *Let $\mathcal{A}_2$ be a nondeterministic $\not\#$OPA defined on an OP alphabet (Σ, M) with s states. Then there exists a nondeterministic OPA $\mathcal{A}_1$ on (Σ, M) and $O(|\Sigma|s)$ states, such that $L(\mathcal{A}_1) = L_{\not\#}(\mathcal{A}_2)$.*

Proof. Let $\mathcal{A}_2$ be $\langle \Sigma, M, Q, I, F, \delta \rangle$ and consider, first, an equivalent form of $\mathcal{A}_2$, where all states are enriched with a lookahead symbol and no final state is reached by a pop edge: $\tilde{\mathcal{A}}_2 = \langle \Sigma, M, Q_2, I_2, F_2, \delta_2 \rangle$, where

- $Q_2 = Q \times \hat{\Sigma}$, where $\hat{\Sigma} = (\Sigma \cup \{\#\})$, i.e. the first component of a state is a state of $\mathcal{A}_2$ and the second component of the state is the lookahead symbol,
- $I_2 = I \times \{a \in \hat{\Sigma} \mid M_{\#a} \neq \emptyset\}$ is the set of initial states of $\tilde{\mathcal{A}}_2$,
- $F_2 = F \times \{\#\}$
- the transition function $\delta_2 : Q_2 \times (\Sigma \cup Q_2) \to \wp(Q_2)$ is defined in the following natural way, where $a, b \in \Sigma$, $p, q, r \in Q$:
  - $\delta_{2\text{push}}(\langle p, a \rangle, a) = \{\langle q, b \rangle \mid q \in \delta_{\text{push}}(p, a) \wedge M_{ab} \neq \emptyset\}$,
  - $\delta_{2\text{shift}}(\langle p, a \rangle, a) = \{\langle q, b \rangle \mid q \in \delta_{\text{shift}}(p, a) \wedge M_{ab} \neq \emptyset\}$,
  - $\delta_{2\text{pop}}(\langle p, a \rangle, \langle q, b \rangle) = \{\langle r, a \rangle \mid r \in \delta_{\text{pop}}(p, q)\} \setminus F_2$.

It is easy too see that $L_{\not\#}(\mathcal{A}_2) = L_{\not\#}(\tilde{\mathcal{A}}_2)$. Furthermore, the final states of $\tilde{\mathcal{A}}_2$ cannot be reached by pop edges: in fact, these pop transitions cannot be performed by a $\not\#$OPA according to the semantics of the transition relation $\vdash_{\not\#}$, since it stops a computation right before reading the delimiter #, when the parsing of the word ends.

Thus, we build, without loss of generality, an OPA $\mathcal{A}_1$ equivalent to the $\not\#$OPA $\tilde{\mathcal{A}}_2$. $\mathcal{A}_1 = \langle \Sigma, M, Q_1, I_1, F_1, \delta_1 \rangle$ has only one final state, reachable through a pop edge by all final states of $\tilde{\mathcal{A}}_2$. Its role is to let $\mathcal{A}_1$ empty the stack after reading a word that is accepted by $\tilde{\mathcal{A}}_2$.

- $Q_1 = Q_2 \cup \{q_{\text{accept}}\}$
- $I_1 = I_2 \cup \{q_{\text{accept}}\}$ if $I_2 \cap F_2 \neq \emptyset$; $I_1 = I_2$ otherwise

- $F_1 = \{q_{\text{accept}}\}$
- The transition function δ_1 equals δ_2 on all states in Q_2; in addition $\mathcal{A}_1$ has departing pop edges from the final states in F_2 to q_{accept} and q_{accept} has no outgoing push/shift edge but only self-loops pop edges.
  The push transition function $\delta_{1\text{push}} : Q_1 \times \Sigma \to \wp(Q_1)$ is defined as $\delta_{1\text{push}}(q, c) = \delta_{2\text{push}}(q, c), \forall q \in Q_2, c \in \Sigma$. The shift function is defined analogously.
  The pop transition $\delta_{1\text{pop}} : Q_1 \times Q_1 \to \wp(Q_1)$ is defined by:
  $\delta_{1\text{pop}}(q, p) = \delta_{2\text{pop}}(q, p), \forall q, p \in Q_2$
  $\delta_{1\text{pop}}(q, p) = q_{\text{accept}}, \forall q \in (F_2 \cup \{q_{\text{accept}}\}), p \in Q_2,$

We now show that $L(\mathcal{A}_1) = L_{\natural}(\tilde{\mathcal{A}}_2)$.

$L(\mathcal{A}_1) \subseteq L_{\natural}(\tilde{\mathcal{A}}_2)$: in fact, if the OPA $\mathcal{A}_1$ recognizes a word, then it is either the empty word and thus $q_{\text{accept}} \in I_1$ and also $\tilde{\mathcal{A}}_2$ has a successful run on it, or $\mathcal{A}_1$ recognizes a word $w \neq \varepsilon$ and there exists a run σ of $\mathcal{A}_1$ which ends in the final state q_{accept} with empty stack. Notice that q_{accept} is reached by a pop move from a state in F_2, say $q_f \in F_2$:

$$\sigma : q_0 \in I_2 \overset{w}{\rightsquigarrow} q_f \Longrightarrow q_{\text{accept}} (\overset{p \in Q_1}{\Longrightarrow} q_{\text{accept}})^*$$

and q_f itself is reached exactly when the reading of w is finished, since, as said before, a state in F_2 cannot be reached by pop moves. This condition is necessary to avoid the presence of sequences of pop moves from non-accepting states toward final states. Then the path from q_0 to q_f, which traverses the same states and edges as σ, represents a run of $\tilde{\mathcal{A}}_2$ which ends in a final state q_f right after the reading of the whole word, thus accepting w. Conversely, the relation $L(\mathcal{A}_1) \supseteq L_{\natural}(\tilde{\mathcal{A}}_2)$ derives easily from the fact that, if $\tilde{\mathcal{A}}_2$ accepts a word along a successful run, then $\mathcal{A}_1$ recognizes the word along the same run, possibly emptying the stack in the final state q_{accept}. □

Remark 3.1. With some further effort –and a further exponential leap in the automaton's size– a deterministic version of this $\natural$OPA could also be built. We did not include it here, however, since the $\natural$OPA construction will be applied only in this chapter to prove the closure w.r.t. concatenation with finite length languages of ωOPLs: we will see that such a closure holds only for nondeterministic automata.

3.3.2 *Closure properties and emptiness problem for class $\mathcal{L}(\omega OPBA)$*

$\mathcal{L}(\omega$OPBA) enjoys all closure and decidability properties suitable for model checking. Precisely, the emptiness problem is decidable for OPAs in polynomial time because they can be interpreted as pushdown automata on infinite-length words: e.g., [25] shows an algorithm that decides the alternation-free modal μ-calculus for context-free processes, with linear complexity in the size of the system's representation.

The following theorems state that $\mathcal{L}(\omega$OPBA) is a Boolean algebra closed under concatenation.

Theorem 3.3 ($\mathcal{L}(\omega$OPBA) is closed under intersection)**.** *Let L_1 and L_2 be ω-languages recognized by two ωOPBA defined over the same alphabet Σ, with compatible precedence matrices M_1 and M_2 and with s_1 and s_2 states respectively. Then $L = L_1 \cap L_2$ is recognizable by an ωOPBA with OPM $M = M_1 \cap M_2$ and $O(s_1 s_2)$ states.*

Proof. Let $\mathcal{A}_1 = \langle \Sigma, M_1, Q_1, I_1, F_1, \delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, M_2, Q_2, I_2, F_2, \delta_2 \rangle$ be two ωOPBA with $L(\mathcal{A}_1) = L_1$ and $L(\mathcal{A}_2) = L_2$ and with compatible precedence matrices M_1 and M_2. Suppose, without loss of generality, that Q_1 and Q_2 are disjoint and do not contain $\{0, 1, 2\}$.

First, observe that, the two OPMs being compatible, at each move either the two automata perform the same type of move (push/shift/pop), or at least one of them stops without accepting since its transition function is not defined.

An ωOPBA that recognizes $L_1 \cap L_2$ is defined in a similar way as for classical finite-state Büchi automata; precisely, $\mathcal{A} = \langle \Sigma, M = M_1 \cap M_2, Q, I, F, \delta \rangle$ where:

- $Q = Q_1 \times Q_2 \times \{0, 1, 2\}$,
- $I = I_1 \times I_2 \times \{0\}$,
- $F = Q_1 \times Q_2 \times \{2\}$

- the transition function $\delta : Q \times (\Sigma \cup Q) \to \wp(Q)$ is defined as follows, where $p_1, q_1, p_2, q_2 \in Q, a \in \Sigma$:
  - $\delta_{\text{push}}(\langle p_1, p_2, x\rangle, a) = \{\langle r_1, r_2, y\rangle \mid r_1 \in \delta_{1\text{push}}(p_1, a) \land r_2 \in \delta_{2\text{push}}(p_2, a)\}$
  - $\delta_{\text{shift}}(\langle p_1, p_2, x\rangle, a) = \{\langle r_1, r_2, y\rangle \mid r_1 \in \delta_{1\text{shift}}(p_1, a) \land r_2 \in \delta_{2\text{shift}}(p_2, a)\}$
  - $\delta_{\text{pop}}(\langle p_1, p_2, x\rangle, \langle q_1, q_2, z\rangle) = \{\langle r_1, r_2, y\rangle \mid r_1 \in \delta_{1\text{pop}}(p_1, q_1) \land r_2 \in \delta_{2\text{pop}}(p_2, q_2)\}$

  and the third component of the states is computed as follows:
  - if $x = 0$ and $r_1 \in F_1$ then $y = 1$
  - if $x = 1$ and $r_2 \in F_2$ then $y = 2$
  - if $x = 2$ then $y = 0$
  - $y = x$ otherwise.

Reading an input string, the automaton $\mathcal{A}$ simulates $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively on the first two components of the states, whereas the third component keeps track of the succession of visits of the two automata to their final states: in particular its value is 0 at the beginning, then switches from 0 to 1, from 1 to 2 and then back to 0, whenever the first automaton reaches a final state and the other one visits a final state afterwards. This cycle is repeated infinitely often whenever both the automata reach their final states infinitely many times along their run.

Conversely, if an ω-word x does not belong to $L_1 \cap L_2$, then at least one of the runs of $\mathcal{A}_1$ and $\mathcal{A}_2$ must either stop because the transition function of the automaton is undefined for the given input or it does not visit infinitely often final states. Hence, $\mathcal{A}$ cannot have a successful run on x and the word is rejected by $\mathcal{A}$ too. □

Theorem 3.4 ($\mathcal{L}(\omega$OPBA) is closed under union)**.** *Let L_1 and L_2 be ω-languages recognized by two ωOPBA defined over the same alphabet Σ, with compatible precedence matrices M_1 and M_2 and with s_1 and s_2 states respectively. Then $L = L_1 \cup L_2$ is recognizable by an ωOPBA with OPM $M = M_1 \cup M_2$ and $O(|\Sigma|^2(s_1 + s_2))$ states.*

Proof. Let A_1 and $\mathcal{A}_2$ be ωOPBAs accepting L_1 and L_2 over OPMs M_1 and M_2, respectively. Without loss of generality we may assume $M = M_1 = M_2$ (otherwise one can apply Statement 3.1 increasing the number of states by a factor $|\Sigma|^2$). For $i = 1, 2$, let $\mathcal{A}_i = \langle \Sigma, M, Q_i, I_i, F_i, \delta_i \rangle$. Then the ω-language $L = L_1 \cup L_2$ is recognized by the ωOPBA $\mathcal{A} = \langle \Sigma, M, Q = Q_1 \cup Q_2, I = I_1 \cup I_2, F = F_1 \cup F_2, \delta \rangle$ whose transition function $\delta : Q \times (\Sigma \cup Q) \rightarrow \wp(Q)$ is the nondeterministic union of δ_1 and δ_2, defined by setting $\forall p, q \in Q, a \in \Sigma$:

$$\delta_{\text{push}}(q,a) = \begin{cases} \delta_{1\text{push}}(q,a) & \text{if } q \in Q_1 \\ \delta_{2\text{push}}(q,a) & \text{if } q \in Q_2 \end{cases}, \quad \delta_{\text{shift}}(q,a) = \begin{cases} \delta_{1\text{shift}}(q,a) & \text{if } q \in Q_1 \\ \delta_{2\text{shift}}(q,a) & \text{if } q \in Q_2 \end{cases},$$

$$\delta_{\text{pop}}(p,q) = \begin{cases} \delta_{1\text{pop}}(p,q) & \text{if } p,q \in Q_1 \\ \delta_{2\text{pop}}(p,q) & \text{if } p,q \in Q_2 \end{cases}.$$

The above definition is well-posed since it applies to automata that share the same precedence matrix, because they perform the same type of move (push/shift/pop) while reading the input word.

Since the sets of states of the two automata are disjoint and Q is their union, then for every $x \in \Sigma^\omega$ there exists a successful run in $\mathcal{A}$ iff there exists a successful run of $\mathcal{A}_1$ on x or a successful run of $\mathcal{A}_2$ on x.

Clearly, the number of states of A is $|Q| = |Q_1| + |Q_2|$ and this concludes the proof, recalling the possible factor $|\Sigma|^2$ implied by Statement 3.1. □

Theorem 3.5 (Closure of $\mathcal{L}$(ωOPBA) under complementation)**.** *Let M be a conflict-free precedence matrix on an alphabet* Σ*. Let L be an* ω*-language on* Σ *that is recognized by a nondeterministic* ω*OPBA with precedence matrix M and s states. Then the complement of L w.r.t.* L_M *(the language of all the words* $x \in \Sigma^\omega$ *compatible with M) is recognized by an* ω*OPBA with the same precedence matrix M and* $2^{O(s^2+|\Sigma|s\, log|\Sigma|s)}$ *states.*

Proof. The proof follows to some extent the structure of the corresponding proof for $\mathcal{L}$(ωBVPA) [11], but it exhibits some relevant technical aspects which distinctly characterize it; in particular, we need to introduce an ad-hoc factorization of ω-words due to the more complex management of the stack performed by OPAs.

Let $\mathcal{A} = \langle \Sigma, M, Q, I, F, \delta \rangle$ be a nondeterministic ωOPBA with $|Q| = s$. Without loss of generality $\mathcal{A}$ can be considered complete with respect to the transition function δ, i.e. there is a run of $\mathcal{A}$ on every ω-word on Σ compatible with M.

An ω-word $w \in \Sigma^\omega$ compatible with M can be factored as a sequence of chains and pending letters $w = w_1 w_2 w_3 \ldots$ where either $w_i = a_i \in \Sigma$ is a pending letter or $w_i = a_{i1} a_{i2} \ldots a_{in}$ is the body of the chain ${}^{l_i}[w_i]^{first_{i+1}}$, where l_i denotes the last pending letter preceding w_i in the word and $first_{i+1}$ denotes the first letter of word w_{i+1}. Let also, by convention, $a_0 = \#$ be the first pending letter.

Such factorization is not unique, since a string w_i can be nested into a larger chain having the same preceding pending letter. The factorization is unique, however, if we additionally require that the body w_i has no prefix (including itself) $u_i b$ such that ${}^{l_i}[u_i]^b$ is a chain; in fact, in this case, as soon as a chain body with its context is identified after a pending letter, it becomes part of the factorization and what follows is either the beginning of a new body or a new pending letter.

For instance, for the word $w = \underbrace{\lessdot a \lessdot c \gtrdot}\, b\, \underbrace{\lessdot a \gtrdot}\, \underbrace{\;\; d \gtrdot}\, b \ldots$, with precedence relations in the OPM $a \gtrdot b$ and $b \lessdot d$, two possible factorizations are $w = w_1 b w_3 b \ldots$ and $w = \overline{w}_1 b \overline{w}_3 \overline{w}_4 b \ldots$, where b is a pending letter and ${}^{\#}[w_1]^b = {}^{\#}[\overline{w}_1]^b = {}^{\#}[ac]^b$, ${}^{b}[w_3]^b = {}^{b}[\overline{w}_3 d]^b$, ${}^{b}[\overline{w}_3]^d = {}^{b}[a]^d$ and ${}^{b}[\overline{w}_4]^b = {}^{b}[d]^b$ are chains. The second factorization is the unique one where each word $\overline{w}_i$ has no prefix $u_i b$ such that ${}^{l_i}[u_i]^b$ is a chain.

Let $x \in \Sigma^*$ be the body of some chain ${}^{a}[x]^b$ and let $T(x)$ be the set of all triples $(q, p, f) \in Q \times Q \times \{0, 1\}$ such that there exists a support $q \overset{x}{\rightsquigarrow} p$ in $\mathcal{A}$, and $f = 1$ iff the support contains a state in F. Also let $\mathcal{T}$ be the set of all such $T(x)$, i.e., $\mathcal{T}$ contains sets of triples identifying all supports for some chain, and set PR to be the finite alphabet $\Sigma \cup \mathcal{T}$. A *pseudorun* for the word w in $\mathcal{A}$'s uniquely factorized as $w_1 w_2 w_3 \ldots$ as stated above, is the ω-word $w' = y_1 y_2 y_3 \ldots \in PR^\omega$ where $y_i = a_i$ if w_i is a pending letter, otherwise $y_i = T(w_i)$.

For the unique factorization in the example above, $w' = T(ac)\ b\ T(a)\ T(d)\ b \ldots$

The automaton recognizing the complement of $L = L(\mathcal{A})$ w.r.t. L_M can be built as an "online composition" of a transducer ωOPBA $\mathcal{B}$ that computes the pseudorun corresponding to an input word w, and a Büchi finite-state automaton $\mathcal{B}_R$ that recognizes all the pseudoruns of ω-words not accepted by $\mathcal{A}$: while reading w, $\mathcal{B}$ outputs the pseudorun w' of w online, and the states of $\mathcal{B}_R$ are updated accordingly. The automaton accepts if both $\mathcal{B}$ and $\mathcal{B}_R$ reach infinitely often final states.

In order to define $\mathcal{B}_R$ we first define a nondeterministic Büchi finite-state automaton $\mathcal{A}_R = \langle PR, Q_{A_R}, I_{A_R}, F_{A_R}, \delta_{A_R} \rangle$ over the alphabet PR whose language includes all pseudoruns w' of any words $w \in L(\mathcal{A})$.

The states of $\mathcal{A}_R$ correspond to the states of $\mathcal{A}$, but are extended with a lookback symbol that, in a correct pseudorun, represents the last pending letter of the input word read so far. $\mathcal{A}_R$ has all transitions corresponding to $\mathcal{A}$'s push and shift transitions but is devoid of pop edges (in fact it is a finite state automaton). In addition, for every $S \in \mathcal{T}$ it is endowed with arcs labeled S which link, for each triple (q, p, f) in S and $a \in \hat{\Sigma} = \Sigma \cup \{\#\}$, either the pair of states $\langle a, q \rangle$, $\langle a, p \rangle$ if $f = 0$, or $\langle a, q \rangle$, $\langle a, p' \rangle$ if $f = 1$, where $\langle a, p' \rangle$ is a new final state which takes into account the states in F met along the support $q \rightsquigarrow p$ and which has the same outgoing edges as $\langle a, p \rangle$.

Formally, $Q_{A_R} = \hat{\Sigma} \times (Q \cup Q')$, where $Q' = \{q' \mid q \in Q\}$, $I_{A_R} = \{\#\} \times I$, $F_{A_R} = \hat{\Sigma} \times (F \cup Q')$. The transition function of $\mathcal{A}_R$ is defined as follows, where $a \in \hat{\Sigma}, q \in Q, q' \in Q', S \in \mathcal{T}$ (δ_{push} and δ_{shift} are the transition functions of $\mathcal{A}$):

- $\delta(\langle a, q \rangle, b) = \begin{cases} \langle b, \delta_{\text{push}}(q, b) \rangle & \text{if } a \lessdot b \\ \langle b, \delta_{\text{shift}}(q, b) \rangle & \text{if } a \doteq b \end{cases}$
- $\delta(\langle a, q \rangle, S) = \{\langle a, p \rangle \mid \langle q, p, 0 \rangle \in S\} \cup \{\langle a, p' \rangle \mid \langle q, p, 1 \rangle \in S\}$
- $\delta(\langle a, q' \rangle, X) = \delta(\langle a, q \rangle, X), \forall X \in PR.$

Notice that, given a set $S \in \mathcal{T}$, the existence of an edge S between the pairs of states q, p in the triples in S can be decided in an effective way.

The automaton $\mathcal{A}_R$ built so far is able to parse all pseudoruns and recognizes all pseudoruns of ω-words recognized by $\mathcal{A}$. However, since its moves are no longer completely determined by the OPM M, it can also accept input words along the edges of the graph of $\mathcal{A}$ that are not pseudoruns since they do not correspond to a correct factorization on PR. This is irrelevant, however, since the aim of the proof is to devise an automaton recognizing the complement of $L(\mathcal{A})$, and all the words in $L_M \setminus L(\mathcal{A})$ are parsed along pseudoruns, which are not accepted by $\mathcal{A}_R$. If one gives as input words only pseudoruns (and not generic words on PR), then they will be accepted by

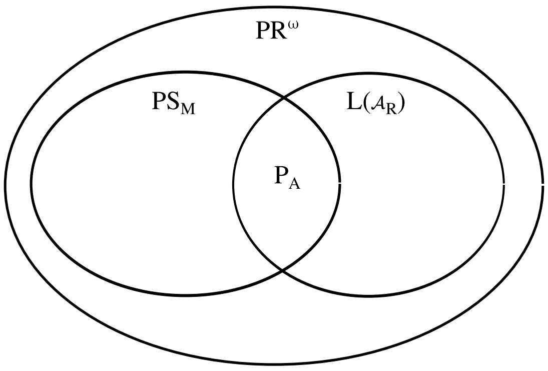


Figure 27.: Containment relations for languages, where $PS_M = \{w' \in PR^\omega \mid w'$ is the pseudorun in $\mathcal{A}$ for $w \in L_M\}$ and $P_A = \{w' \in PR^\omega \mid w'$ is the pseudorun in $\mathcal{A}$ for $w \in L(\mathcal{A})\}$.

$\mathcal{A}_R$ if the corresponding words on Σ belong to $L(\mathcal{A})$, and they will be rejected if the corresponding words do not belong to $L(\mathcal{A})$ (see Figure 27). Then we can construct a deterministic Büchi automaton $\mathcal{B}_R$ that accepts the complement of $L(\mathcal{A}_R)$, on the alphabet PR [84]. If $\mathcal{B}_R$ receives only input words on PR that are pseudoruns, then it will accept only words in $L_M \setminus L(\mathcal{A})$.

Now we define a nondeterministic transducer ωOPBA $\mathcal{B}$ which on reading w generates online the pseudorun w'. The transducer $\mathcal{B}$ nondeterministically guesses whether the next input symbol is a pending letter, the beginning of a chain appearing in the factorization of w, or a symbol within such a chain, and uses stack symbols Z, B, or elements in $\mathcal{T}$, respectively, to distinguish these three cases.

Whenever the automaton reads a pending letter it outputs the same letter, whereas when it completes the recognition of a chain of the factorization, performing a pop move that removes from the stack an element with state B, it outputs the set of all the pairs of states which define a support for the chain. Thus, the output w' produced by $\mathcal{B}$ is unique, despite the nondeterminism of the translator.

Formally, the transducer ωOPBA $\mathcal{B} = \langle \Sigma, M, Q_B, I_B, F_B, PR, \delta_B, \eta_B \rangle$ is defined as follows:

- $Q_B = \{Z, B\} \cup \mathcal{T}$, i.e., a state in Q_B represents the guess whether the next symbol to be read is a pending letter (Z), the beginning of a chain (B), or a letter within such a chain w_i ($T \in \mathcal{T}$). In the third case, T contains all information necessary to correctly simulate the moves of $\mathcal{A}$ during the parsing of the chain w_i of w, and compute the corresponding symbol y_i of w'. In particular, T is a set comprising all triples (r, q, v) where r represents the state reached before the last push move, q represents the current state reached by $\mathcal{A}$, and v is a bit that reminds whether, while reading the chain, a state in F has been encountered (as in the construction of a deterministic OPA on words of finite length, it is necessary to keep track of the state from which the parsing of a chain started, to avoid erroneous merges of runs on pop moves).

- $I_B = F_B = \{B, Z\}$.

- The transition function and the output function are defined as the union of three pairs of functions. Let $a \in \Sigma, T, S \in \mathcal{T}$.

  The push pair $\langle \delta_{\text{Bpush}}, \eta_{\text{Bpush}} \rangle : Q_B \times \Sigma \to \wp_F(Q_B \times PR^*)$ is defined as follows, where the symbols after $\downarrow$ denote the output.

  - *Push of a pending letter.*

$$\langle \delta_{\text{Bpush}}, \eta_{\text{Bpush}} \rangle \, (Z, a) = \{B \downarrow a, \ Z \downarrow a\}$$

  - *Push at the beginning of a chain of the factorization.*

$$\langle \delta_{\text{Bpush}}, \eta_{\text{Bpush}} \rangle \, (B, a) = \{T \downarrow \varepsilon\}$$

$$\text{where } T = \left\{ \langle q, p, v \rangle \mid q \in Q, p \in \delta_{\text{push}}(q, a), v = 1 \text{ iff } p \in F \right\}$$

  - *Push within a chain of the factorization.*

$$\langle \delta_{\text{Bpush}}, \eta_{\text{Bpush}} \rangle \, (T, a) = \{S \downarrow \varepsilon\} \quad \text{where}$$

$$S = \left\{ \langle q, p, v\rangle \mid \exists \langle r, q, \xi\rangle \in T \text{ s.t. } v = \begin{bmatrix} \xi & \text{if } p \notin F \\ 1 & \text{if } p \in F \end{bmatrix}, \; p \in \delta_{\text{push}}(q, a) \right\}$$

The shift pair $\langle \delta_{\text{Bshift}}, \eta_{\text{Bshift}} \rangle : Q_B \times \Sigma \to \wp_F(Q_B \times PR^*)$ is defined as follows:

- *Pending letter.*

$$\langle \delta_{\text{Bshift}}, \eta_{\text{Bshift}} \rangle \, (Z, a) = \{B \downarrow a, \; Z \downarrow a\}$$

- *Shift move within a chain of the factorization.*

$$\langle \delta_{\text{Bshift}}, \eta_{\text{Bshift}} \rangle \, (T, a) = \{S \downarrow \varepsilon\} \quad \text{where}$$

$$S = \left\{ \langle r, p, v\rangle \mid \exists \langle r, q, \xi\rangle \in T \text{ s.t. } v = \begin{bmatrix} \xi & \text{if } p \notin F \\ 1 & \text{if } p \in F \end{bmatrix}, \; p \in \delta_{\text{shift}}(q, a) \right\}$$

The pop pair $\langle \delta_{\text{Bpop}}, \eta_{\text{Bpop}} \rangle : Q_B \times Q_B \to \wp_F(Q_B \times PR^*)$ is defined as follows.

- *Pop at the end of a chain of the factorization.*

$$\langle \delta_{\text{Bpop}}, \eta_{\text{Bpop}} \rangle (T, B \rangle) = \{B \downarrow R, \; Z \downarrow R\} \quad \text{where}$$

$$R = \left\{ \langle r, p, v\rangle \mid \exists \langle r, q, \xi\rangle \in T \text{ s.t. } p \in \delta_{\text{pop}}(q, r), v = \begin{bmatrix} \xi & \text{if } p \notin F \\ 1 & \text{if } p \in F \end{bmatrix} \right\}$$

- *Pop within a chain of the factorization*[9].

$$\langle \delta_{\text{Bpop}}, \eta_{\text{Bpop}} \rangle (T, S \rangle) = \{R \downarrow \varepsilon\} \quad \text{where}$$

9 Remember that we consider only chains having no prefixes that are chains.

$$R = \Big\{ \langle t, p, \nu \rangle \mid \exists \langle r, q, \xi \rangle \in T, \exists \langle t, r, \zeta \rangle \in S \text{ s.t. } p \in \delta_{\text{pop}}(q, r),$$
$$\nu = \left[\begin{array}{ll} \xi & \text{if } p \notin F \\ 1 & \text{if } p \in F \end{array} \right\} \Big\}$$

An error state is reached in any other case.

We conclude the construction by computing the size of the resulting automaton, which is an "online composition" of $\mathcal{B}$ and $\mathcal{B}_R$. The Büchi finite-state automaton $\mathcal{A}_R$ has $O(|\Sigma|s)$ states and hence the automaton $\mathcal{B}_R$ has $2^{O(|\Sigma|s \log|\Sigma|s)}$ states [96, 84]; while the transducer $\mathcal{B}$ has $|Q_B| = 2^{O(s^2)}$ states. Thus the ωOPBA has $2^{O(s^2+|\Sigma|s \log|\Sigma|s)}$ states.

To prove that $\mathcal{B}$ produces all $\mathcal{A}$'s pseudoruns –whether accepting or not– observe, first, that its guess about reading a pending letter or the beginning of a chain belonging to the unique factorization defined above, or reading a symbol within such a chain, is essentially the same as the one described in the proof of Lemma 3.1, where the recognition of a maximal chain is replaced by the recognition of a chain with no prefixes that are chains; thus, wrong guesses are resolved at the time of a pop move (e.g., a pop move is not defined on a first state of type Z). Furthermore, pending letters, when correctly guessed as such, are output as soon as they are read (the incorrectly guessed ones belong to runs that will be aborted); elements of $\mathcal{T}$ are output only when a chain of the factorization is recognized, i.e., the transition is defined on a pair of states whose second component is B, which separates these moves from the pop ones occurring within a chain of the factorization; the set T output during the move records all pairs of states that can be the beginning and the end of a support of the recognized chain. Finally the input string is accepted iff infinitely many times either pending letters are read or chains of the factorization are recognized, or both facts occur, i.e., the string is compatible with the OPM, and the produced output is the pseudorun associated with the input by definition, independently on whether the original $\mathcal{A}$'s run was accepting, i.e., infinitely many times sets of triples with $\nu = 1$ have been output, or not. $\square$

Let us finally consider the case of concatenation between a finite length OPL and a language in $\mathcal{L}(\omega\text{OPBA})$. For classical families of automata (on finite or infinite length words) the closure with respect to concatenation is traditionally proved by building an automaton which simulates the moves of the first automaton while reading the first word of the concatenation and –whether deterministically or not– once it reaches some final state, it switches to the initial states of the second one. This natural approach has already been proved ineffective for OPLs in the case of finite-length words since the structure of two concatenated strings is not necessarily the concatenation of the two structures, so that the actions of the second automaton cannot be independent from those of the previous one ([34] provides a constructive proof of the closure of finite-length OPLs w.r.t. concatenation in terms of generating grammars); in fact the lack of the # delimiter between the two strings prevents the typical look-ahead mechanisms which drives the operator-based parsing; thus, the stack cannot be emptied by the normal sequence of pop moves before beginning the parse of the new string. In the case of ω-languages the difficulty is further exacerbated by the fact the automaton might never be able to empty the stack, as e.g., in the case of a language $L_1 \subseteq \{a, b\}^*$ with $a \lessdot a$, $b \lessdot a$, concatenated with $L_2 = \{a^\omega\}$. Notice also that, after reading the first finite word in the concatenation, it would not be possible to determine whether this word *might* be accepted by –possibly nondeterministically– guessing the position of a potential delimiter #, since this check would require to know the states already reached and piled on the stack, which are not visible without emptying the stack itself.

To overcome the above difficulties we follow this approach:

- We give up deterministic parsing. In fact the different computational power between deterministic and nondeterministic automata is a distinguishing property when moving from finite to infinite length languages. Thus, we nondeterministically guess the point of separation between the first finite word and the second infinite one.

- To afford the second major problem, i.e., the lack of enough knowledge to decide whether the guessed first word would be accepted by the corresponding automaton, we use $\not\#$OPAs introduced in Section 3.3.1.

The following theorem exploits the above approach. Its proof differs significantly from the non-trivial proof of closure under concatenation of OPLs of finite-length words [34], which, instead, can be recognized deterministically.

Theorem 3.6 ($\mathcal{L}(\omega$OPBA) is closed under concatenation)**.** *Let $L_1 \subseteq \Sigma^*$ be a language of finite words recognized by an OPA with OPM M_1 and s_1 states. Let $L_2 \subseteq \Sigma^\omega$ be an ω-language recognized by a nondeterministic ωOPBA with OPM M_2 compatible with M_1 and s_2 states. Then the concatenation $L_1 \cdot L_2$ is also recognized by an ωOPBA with OPM $M \supseteq M_1 \cup M_2$ and $O(s_1^2 + s_2^2)$ states.*

Proof. Let $\mathcal{A}_1$ be a nondeterministic OPA on (Σ, M_1) that recognizes L_1 and let $\mathcal{A}_2 = \langle \Sigma, M_2, Q_2, I_2, F_2, \delta_2 \rangle$ be a nondeterministic ωOPBA with OPM M_2 compatible with M_1 that accepts L_2. Suppose, without loss of generality, that the sets of states of $\mathcal{A}_1$ and $\mathcal{A}_2$ are disjoint.

To define an automaton ωOPBA $\mathcal{A}$ that accepts $L_1 \cdot L_2$, we first build a $\not\#$OPA $\mathcal{A}'_1 = \langle \Sigma, M_1, Q_1, I_1, F_1, \delta_1 \rangle$ such that $L_{\not\#}(\mathcal{A}'_1) = L(\mathcal{A}_1)$.

The automaton $\mathcal{A}$ can recognize the first finite words in the concatenation $L_1 \cdot L_2$ by simulating $\mathcal{A}'_1$: reading the input string, if $\mathcal{A}'_1$ reaches a final state at the end of a finite-length prefix, then it belongs to L_1 and $\mathcal{A}$ immediately starts the recognition of the second infinite string without the need to perform any pop move to empty the stack. From this point onwards, then, $\mathcal{A}$ checks that the remaining infinite portion of the input belongs to L_2, behaving as the ωOPBA $\mathcal{A}_2$.

The strings belonging to the concatenation of two OPLs, however, may contain new chains that span over the two concatenated words. Consider, for instance, the concatenation of $L_1 = \{a^m b^n \mid m \geq n \geq 1\}$ with $L_2 = \{c^+ b^\omega\}$; notice that any OPA recognizing L_1 must be defined on an OPM such that $a \lessdot a$, $a \doteq b$, $b \gtrdot b$ to be able to compare the occurrences of a with those of b; assume also the further precedence relations $a \lessdot c$, $c \lessdot c$, $c \gtrdot b$ (such relations could be mandated, e.g., by other components of either language not included here for simplicity). An automaton recognizing $L_1 \cdot L_2$ can deterministically find the borderline between words $x \in L_1$, and $y \in L_2$; after finishing reading x it will have on its stack $m - n$ remaining a's; however, since $a \lessdot c$ it cannot empty the stack and must push all c's on top of the a's. Only when receiving the first b, it will pop all c's until the top of the stack will store an a. Since $a \doteq b$, and $b \gtrdot b$ the next action must consist in shifting the b by replacing the topmost a and then popping it, thus consuming part of the stack left by the analysis of x; in other words, it must produce the support of a chain ${}^{a}[ab]^{b}$, whose left part belongs to L_1 and whose right part belongs to L_2.

Therefore, $\mathcal{A}$ cannot merely read the second infinite word performing the same transitions as $\mathcal{A}_2$, but it can still simulate this ωOPBA by keeping in the states some summary information about its runs. In this way, while reading the second word in the concatenation, whenever $\mathcal{A}$ has to reduce a chain that extends to the previous word in L_1 and, therefore, must perform a pop move of a symbol in the portion of the stack piled up during the parsing of the first word, it can restore the state that $\mathcal{A}_2$ would instead have reached, resuming therefrom as in a run of $\mathcal{A}_2$.

Precisely, $\mathcal{A}$ is defined as the tuple $\langle \Sigma, M, Q, I, F, \delta \rangle$ where:

- $M \supseteq M_1 \cup M_2$ and may be supposed to be a complete matrix, for instance assigning arbitrary precedence relations to the empty entries, so that the strings in the concatenation of languages L_1 and L_2 are compatible with M.

- $Q = Q_1 \cup Q_2 \cup Q_2 \times (Q_2 \cup \{-\})$, i.e. the set of states of $\mathcal{A}$ includes the states of $\mathcal{A}'_1$ and $\mathcal{A}_2$, along with the states of $\mathcal{A}_2$ extended with a second component. The first component is the state of Q_2 that $\mathcal{A}_2$ would reach in its corresponding computation on the second word of the concatenation, and the second one represents the state of the symbol that is on the top of the stack when the current input letter is read in this run of $\mathcal{A}_2$. Storing this component is necessary to guarantee that, whenever the automaton $\mathcal{A}$ has to perform a pop move that removes symbols that have been piled on the stack during the recognition of the first word in the concatenation, it is still possible to compute the state that $\mathcal{A}_2$ would have reached instead.

  This second component is denoted $'-'$ if all the preceding symbols in the stack have been piled up during the parsing of the first word of the concatenation (thus the stack of $\mathcal{A}_2$ is empty).

- $I = I_1 \cup \{\langle q_0, - \rangle \mid q_0 \in I_2\}$ if $\varepsilon \in L_1$; $I = I_1$ otherwise

- $F = F_2 \cup F_2 \times (Q_2 \cup \{-\})$

- The transition function $\delta : Q \times (\Sigma \cup Q) \to \wp(Q)$ is $\delta = \delta_1 \cup \delta_2 \cup \delta^{\text{join}}$ and is defined as the union of three functions: the transition functions of $\mathcal{A}'_1$ and $\mathcal{A}_2$ by which it simulates the first automaton on the first word of the concatenation and the second automaton on the second one, and a function δ^{join} that handles the nondeterministic transition from the simulation of the first automaton to the second one and the parsing of the suffix (within the second word of the concatenation) of the chains that span over the two words.

  Function δ^{join} is defined as follows: let $c \in \Sigma$, $p \in Q_1, q, q_1, q_2, q_3 \in Q_2$, $r \in (Q_2 \cup \{-\})$.

  The push transition function $\delta^{\text{join}}_{\text{push}} : Q \times \Sigma \to \wp(Q)$ is defined by:

  - $\delta^{\text{join}}_{\text{push}}(p, c) = \{\langle q_0, -\rangle \mid q_0 \in I_2,\ \text{if } \exists p_f \in F_1 \text{ s.t. } \delta_{1\text{push}}(p, c) \ni p_f\}$,

    i.e., $\mathcal{A}$ nondeterministically enters the initial states of $\mathcal{A}_2$ after the recognition of a word in L_1

  - $\delta^{\text{join}}_{\text{push}}(\langle q, r\rangle, c) = \delta_{2\text{push}}(q, c)$,

    i.e., $\mathcal{A}$ simulates a push move of $\mathcal{A}_2$, reaching a state in Q_2, whenever it starts to recognize a chain in the second word of the concatenation (which thus does not extend to the first word).

  The shift transition function $\delta^{\text{join}}_{\text{shift}} : Q \times \Sigma \to \wp(Q)$ is defined by:

  - $\delta^{\text{join}}_{\text{shift}}(p, c) = \{\langle q_0, -\rangle \mid q_0 \in I_2,\ \text{if } \exists p_f \in F_1 \text{ s.t. } \delta_{1\text{shift}}(p, c) \ni p_f\}$,

    i.e., $\mathcal{A}$ nondeterministically enters the initial states of $\mathcal{A}_2$ after the recognition of a word in L_1

  - $\delta^{\text{join}}_{\text{shift}}(\langle q_1, -\rangle, c) = \{\langle q_2, q_1\rangle \mid q_2 \in \delta_{2\text{push}}(q_1, c)\}$,

    i.e., $\mathcal{A}$ simulates the push move induced by the precedence relation $\# \lessdot c$ that, in the corresponding run of $\mathcal{A}_2$, starts the recognition of a chain that is a prefix of the second word of the concatenation

  - $\delta^{\text{join}}_{\text{shift}}(\langle q_1, q_2\rangle, c) = \{\langle q_3, q_2\rangle \mid q_3 \in \delta_{2\text{shift}}(q_1, c)\}$, i.e., $\mathcal{A}$ performs a shift move within a chain that spans over the two words of the concatenation.

The pop transition function $\delta_{\text{pop}}^{\text{join}} : Q \times Q \to \wp(Q)$ is defined by:

- $\delta_{\text{pop}}^{\text{join}}(\langle q, - \rangle, p) = \langle q, - \rangle$,

  i.e, $\mathcal{A}$ concludes to recognize a chain, at the end of the first word of the concatenation, induced by the precedence relations with the letters of the second string, and consumes the corresponding stack symbols piled while reading the first word

- $\delta_{\text{pop}}^{\text{join}}(\langle q_1, q_2 \rangle, p) = \{\langle q_3, - \rangle \mid q_3 \in \delta_{2\text{pop}}(q_1, q_2)\}$,

  i.e., whenever the precedence relations induce a merge of the chains of the words of the concatenation, $\mathcal{A}$ restores the state q_3 of $\mathcal{A}_2$ from which a run of $\mathcal{A}_2$ will continue

- $\delta_{\text{pop}}^{\text{join}}(q_1, \langle q_2, r \rangle) = \{\langle q_3, r \rangle \mid q_3 \in \delta_{2\text{pop}}(q_1, q_2)\}$,

  i.e., $\mathcal{A}$ completes the recognition of a chain that belongs to a composed chain spanning over the two words of the concatenation.

One can verify that, after having simulated $\mathcal{A}'_1$ and nondeterministically guessed the end of a word in L_1, $\mathcal{A}$ proceeds with the simulation of $\mathcal{A}_2$ and accepts the remaining ω-string iff it belongs to L_2. In fact, the projection on the first component of the states visited along $\mathcal{A}$'s run on the second word of the concatenation identifies a successful run of $\mathcal{A}_2$ on the same word. □

To summarize, Table 3 displays the complexities (upper bounds) of the various constructions to obtain the closure w.r.t. Boolean operations and concatenation; it also compares them with the corresponding complexities for VPLs showing that the only main difference occurs in the case of concatenation. We leave as a future work the study of the optimality of these bounds.

	$\mathcal{L}(\omega\text{OPBA})$	$\mathcal{L}(\omega\text{BVPA})$
$L_1 \cap L_2$	$O(s_1 s_2)$	$O(s_1 s_2)$
$L_1 \cup L_2$	$O(\|\Sigma\|^2(s_1 + s_2))$	$s_1 + s_2$
$\neg L_1$	$2^{O(s_1^2+\|\Sigma\|s_1 \log\|\Sigma\|s_1)}$	$2^{O(s_1^2)}$
$L_3 \cdot L_1$	$O(s_1^2 + s_3^2)$	$s_1 + s_3$

Table 3.: Size of state sets of languages recognizing $L_1 \cap L_2$, $L_1 \cup L_2$, $\neg L_1$ and $L_3 \cdot L_1$. The results on ωOPBAs have been proved, respectively, in Theorem 3.3, Theorem 3.4, Theorem 3.5 and Theorem 3.6. The complexity results on ωBVPAs derive from the constructions and proofs of their closure properties shown in [11].

3.3.3 *Closure properties of the other classes of ωOPLs*

The class of languages recognized by ωDOPMAs is a Boolean algebra. The other classes are closed only under union and intersection.

Theorem 3.7 ($\mathcal{L}(\omega\text{DOPMA})$ is a Boolean algebra)**.** *Let L_1 and L_2 be ω-languages that are recognized by two ωDOPMAs defined over the same alphabet Σ, with compatible precedence matrices M_1 and M_2 and s_1 and s_2 states respectively. Then $L_1 \cap L_2$ (resp. the complement of L_1 w.r.t. L_M, or $L_1 \cup L_2$) is recognized by an ωDOPMA with OPM $M = M_1 \cap M_2$ and $s_1 s_2$ (resp. s_1, or $|\Sigma|^4 s_1 s_2$) states.*

Proof. Let $\mathcal{A}_1 = \langle \Sigma, M_1, Q_1, q_{01}, \mathcal{T}_1, \delta_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, M_2, Q_2, q_{02}, \mathcal{T}_2, \delta_2 \rangle$ be ωDOPMAs recognizing languages L_1 and L_2. Assume without loss of generality that their transition function is total (otherwise, it can be naturally completed once the set of states is extended with an "error" state).

An ωDOPMA $\mathcal{A}$ with OPM $M = M_1 \cap M_2$ recognizing $L = L_1 \cap L_2$ may be defined adopting the usual product construction for ω-regular automata, requiring that a successful path in $\mathcal{A}$ corresponds to paths that visit infinitely often sets in the table $\mathcal{T}_1$ and $\mathcal{T}_2$. More precisely let $\mathcal{A} = \langle \Sigma, M, Q, q_0, \mathcal{T}, \delta \rangle$ where

- $Q = Q_1 \times Q_2$,
- $q_0 = (q_{01}, q_{02})$,

- Define π_i (i = 1, 2) as the projection from $Q_1 \times Q_2$ on Q_i, that can also be naturally extended to define projections on paths of the automata, and let
  $\mathcal{T} = \{P \subseteq Q_1 \times Q_2 \mid \pi_1(P) \in \mathcal{T}_1 \wedge \pi_2(P) \in \mathcal{T}_2\}$,
- The transition function δ is the product of δ_1 and δ_2 (see Definition 3.6).

Let ρ be a successful path of $\mathcal{A}$, starting in the initial state $q_0 = (q_{01}, q_{02})$: since it is accepting, the set $Inf(\rho) = P \in \mathcal{T}$. By definition of $\mathcal{T}$, the paths ρ_1 and ρ_2 that are the projection of ρ on the set of states of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, have $Inf(\rho_1) = \pi_1(P) \in \mathcal{T}_1$ and $Inf(\rho_2) = \pi_2(P) \in \mathcal{T}_2$: hence ρ_1 and ρ_2 are successful paths for the two automata, and x belongs to $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

Let now $x \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$; thus, x labels two successful paths ρ_1 and ρ_2 of the two automata, i.e., $Inf(\rho_1) \in \mathcal{T}_1$ and $Inf(\rho_2) \in \mathcal{T}_2$. The path ρ of $\mathcal{A}$ which visits the pairs of states of the two automata, performing the same type of move they perform for each input symbol, is defined so as $\pi_1(Inf(\rho)) = Inf(\rho_1) \in \mathcal{T}_1$ and $\pi_2(In(\rho)) = Inf(\rho_2) \in \mathcal{T}_2$. Therefore, by definition of $\mathcal{T}$, ρ is a successful path for $\mathcal{A}$.

To recognize the complement of L_1, given that $\mathcal{A}$ is deterministic and its transition function is total, it is clearly sufficient to build the ωDOPMA $\mathcal{A}' = \langle \Sigma, M_1, Q_1, q_{01}, \wp(Q_1) \setminus \mathcal{T}_1, \delta \rangle$ whose table is the complement of $\mathcal{T}_1$ w.r.t. $\wp(Q_1)$.

To obtain the closure w.r.t. union, we can assume that $M_1 = M_2$ w.l.o.g. (otherwise one can apply Statement 3.1, increasing the number of states of each automaton of a factor $|\Sigma|^2$) and apply De Morgan's law. The number of states of the resulting automaton is $|Q_1| \cdot |Q_2|$ and this concludes the proof, recalling the factor $|\Sigma|^2 \cdot |\Sigma|^2$ implied by the possible application of Statement 3.1. Notice that, if one considers automata with compatible but not equal matrices, De Morgan's law could not be applied: in fact, the equality

$$L_1 \cup L_2 = L_{M_1 \cap M_2} \setminus [(L_{M_1} \setminus L_1) \cap (L_{M_2} \setminus L_2)]$$

does not hold, unless $M_1 = M_2$. □

Proposition 3.2. *Let L_1 and L_2 be ω-languages recognized by two ωOPBEA (resp. ωDOPBA, ωDOPBEA) defined over the same alphabet Σ, with compatible precedence matrices M_1 and M_2 and with s_1 and s_2 states respectively. Then $L = L_1 \cap L_2$*

is recognized by an ωOPBEA (resp. ωDOPBA, ωDOPBEA) with OPM $M = M_1 \cap M_2$ and $O(s_1 s_2)$ states.

Proof. For ωOPBEA, we can assume without loss of generality that the the automaton is in normal form with partitioned sets of states, (see Definition 3.5), and apply the same construction as for ωOPBA (see Theorem 3.3). The use of automata with partitioned sets of states guarantees that a run of $\mathcal{A}$ on an ω-word reaches infinitely often a final state with empty stack iff both $\mathcal{A}_1$ and $\mathcal{A}_2$ have a run for the word which traverses infinitely often a final state with empty stack.

For ωDOPBA and ωDOPBEA, the proof derives from the fact that, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are deterministic, then the resulting intersection automaton is deterministic too. □

Proposition 3.3. *Let L_1 and L_2 be ω-languages recognized by two ωOPBEA (resp. ωDOPBA, ωDOPBEA) defined over the same alphabet Σ, with compatible precedence matrices M_1 and M_2 and s_1 and s_2 states respectively. Then $L = L_1 \cup L_2$ is recognized by an ωOPBEA (resp. ωDOPBA, ωDOPBEA) with OPM $M = M_1 \cup M_2$ and $O(|\Sigma|^2 s_1 s_2))$ (resp. $O(|\Sigma|^4 s_1 s_2))$ states.*

Proof. The proof for ωOPBEA is analogous to the proof of closure under union for ωOPBA(see Theorem 3.3).

For the determistic models, the construction must be refined. Let A_1 and A_2 be ωDOPBA accepting L_1 and L_2 over OPMs M_1 and M_2, respectively. As usual, we assume that that both transition functions are complete and $M_1 = M_2$ (otherwise one can apply Statement 3.1, increasing the number of states of a factor $|\Sigma|^2$). Let $A_i = \langle \Sigma, M, Q_i, q_{0i}, F_i, \delta_i \rangle$, for $i = 1, 2$. An ωDOPBA (resp. ωDOPBEA) $\mathcal{A}_3$ which recognizes $L_1 \cup L_2$ is then defined by adopting the usual product construction for regular automata: $\mathcal{A}_3 = \langle \Sigma, M, Q_3, q_{03}, F_3, \delta_3 \rangle$ where:

- $Q_3 = Q_1 \times Q_2$,
- $q_{03} = (q_{01}, q_{02})$,
- $F_3 = F_1 \times Q_2 \cup Q_1 \times F_2$
- and the transition function is the product of δ_1 and δ_2.

The number of states of $\mathcal{A}_3$ is given by the product $|Q_1| \cdot |Q_2|$ and this concludes the proof, recalling the factor $|\Sigma|^2 \cdot |\Sigma|^2$ implied by Statement 3.1. □

Theorem 3.8 (ωDOPBA, ωOPBEA, ωDOPBEA are not closed under complement)**.** *Let L be an ω-language accepted by an ωDOPBA (resp. ωOPBEA, or ωDOPBEA) with OPM M on alphabet Σ. There does not necessarily exist an ωOPBEA (resp. ωOPBEA, or ωDOPBEA) recognizing the complement of L w.r.t. L_M.*

Proof. Language $L_{a\infty}$ can be recognized by an ωDOPBA with an OPM M (shown, for instance, in Figure 20), but there's no ωDOPBA that can recognize the complement of this language w.r.t. L_M, i.e. the language $L_{a-\text{finite}}$, as mentioned in Section 3.2.2. The same argument on $L_{a\infty}$ holds also for ωDOPBEAs.

Finally, as regards ωOPBEAs, L^{ω}_{abseq} is recognized by the ωOPBEA with OPM M and state graph presented in Section 3.2.2. However, no ωOPBEA can recognize the complement of this language w.r.t. L_M. Such an ωOPBEA, in fact, should have OPM M so that no word in L^{ω}_{abseq} can be accepted. The precedence relation $M_{aa} = \{\lessdot\}$ (which is necessary to verify that in a sequence of type $(a^k b^h)^{\omega}$ there is at least one substring with $k \neq h$), however, prevents an ωOPBEA from accepting the word a^{ω}, which belongs to the complement of L^{ω}_{abseq} w.r.t. L_M, since it implies that, while reading the word, the ωOPBEA can never reach a state with empty stack. □

Theorem 3.9 (ωDOPBA, ωOPBEA, ωDOPBEA, and ωDOPMA are not closed under concatenation)**.** *Let L_2 be an ω-language accepted by an ωOPBEA (resp. ωDOPBA, ωDOPBEA, or ωDOPMA) with OPM M on alphabet Σ and let $L_1 \subseteq \Sigma^*$ be a language (of finite words) recognized by an OPA with a compatible precedence matrix. The ω-language defined by the concatenation $L_1 \cdot L_2$ is not necessarily recognizable by an ωOPBEA (resp. ωDOPBA, ωDOPBEA, or ωDOPMA).*

Proof. For ωDOPBAs, let $\Sigma = \{a, b\}$ and consider the language $L_{a-\text{finite}}$, which can be seen as the concatenation $L_{a-\text{finite}} = L_1 \cdot L_2$ of a language of finite words $L_1 = \{a, b\}^*$, which can be clearly recognized by an OPA, and an ω-language $L_2 = \{b^{\omega}\}$, which can be recognized by an ωDOPBA, with compatible precedence matrices. Since language $L_{a-\text{finite}}$ cannot be recognized by an ωDOPBA, then the class of languages $\mathcal{L}(\omega\text{DOPBA})$ is not closed w.r.t. concatenation.

Given $\Sigma = \{c, r\}$, the language L_{repbsd} cannot be recognized by an ωOPBEA (respectively ωDOPBEA, or ωDOPMA), as shown in Section 3.2.2. Consider the OPA that accepts the language $L_1 = \Sigma^*$ of words of finite length whose OPM is the same as the precedence matrix depicted in Figure 21. These words have necessarily a finite number of pending calls, since they have finite length. Moreover, let $\mathcal{A}_2$ be an ωOPBEA (respectively ωDOPBEA, or ωDOPMA) that recognizes the ω-language $L_{\omega\text{Dyck-pr}(c,r)}$ and which is depicted in Figure 21. The concatenation ω-language $L_1 \cdot L_{\omega\text{Dyck-pr}(c,r)}$ is exactly the set of ω-words with a finite number of pending calls, i.e. L_{repbsd}. Hence, the class of languages $\mathcal{L}(\omega\text{OPBEA})$ (respectively $\mathcal{L}(\omega\text{DOPBEA})$, or $\mathcal{L}(\omega\text{DOPMA})$) is not closed w.r.t. concatenation. □

3.4 MONADIC SECOND-ORDER LOGIC CHARACTERIZATION OF ωOPLS

We now provide a characterization of ωOPLs in terms of a MSO logic that is interpreted over infinite words. As usual, we focus our attention on $\mathcal{L}(\omega\text{OPBA})$, the most general class of ωOPLs.

The same approach has been followed for VPLs in [11], where the class of languages accepted by VPAs with Büchi acceptance condition is characterized using the MSO logic recalled in Section 2.4, which is interpreted over infinite words.

Herein we adopt the same conventions and notations as in Section 2.4, and extend the formula evaluation over ω-strings in the natural way. To distinguish the infinite case from the finite one, we will use symbol $\models_\omega$ instead of $\models$. Given an OP alphabet (Σ, M) and a MSO formula φ, we denote the language of all strings $w \in \Sigma^\omega$ such that $\#w \models_\omega \varphi$ by $L^\omega(\varphi) = \{w \in \Sigma^\omega \mid \#w \models_\omega \varphi\}$.

Example 3.3 (Managing interrupts)**.** Consider again the system that manages interrupts described in Example 3.1. The same rules enforced by the automaton of Figure 18 are also formalized by the following sentences.

- All int_2 are eventually served by a corresponding $serve_2$:

$$\forall \boldsymbol{x}\ (int_2(\boldsymbol{x}) \Rightarrow \exists \boldsymbol{y}(serve_2(\boldsymbol{y}) \wedge (\boldsymbol{y} = \boldsymbol{x} + 1 \vee \boldsymbol{x} \curvearrowright \boldsymbol{y}))).$$

- Lower priority interrupts are not accepted when a higher priority one is pending:

$$\forall \boldsymbol{x}, \boldsymbol{y}\ (int_2(\boldsymbol{x}) \wedge serve_2(\boldsymbol{y}) \wedge \boldsymbol{x} \curvearrowright \boldsymbol{y} \Rightarrow \forall \boldsymbol{k}(\boldsymbol{x} < \boldsymbol{k} < \boldsymbol{y} \Rightarrow \neg int_1(\boldsymbol{k}))).$$

As another example consider the "weak fairness requirement" also mentioned in Example 3.1, which states that after a first $call_a$ not matched by ret_a but interrupted by a int_1 or int_2, a second $call_a$ cannot be interrupted by a new lower priority interrupt int_1 (but can still be interrupted at any time by higher priority ones): the sentence below formalizes such a constraint.

$$\begin{aligned}
&\neg\exists \boldsymbol{x}_1, \boldsymbol{x}_2\big(\boldsymbol{x}_1 < \boldsymbol{x}_2 \wedge call_a(\boldsymbol{x}_1) \wedge call_a(\boldsymbol{x}_2) \wedge \\
&\quad \forall \boldsymbol{x}_3(\boldsymbol{x}_1 \le \boldsymbol{x}_3 \le \boldsymbol{x}_2 \wedge call_a(\boldsymbol{x}_3) \Rightarrow \neg\exists \boldsymbol{y}_3(ret_a(\boldsymbol{y}_3) \wedge (\boldsymbol{y}_3 = \boldsymbol{x}_3 + 1 \vee \boldsymbol{x}_3 \curvearrowright \boldsymbol{y}_3))) \wedge \\
&\quad \exists \boldsymbol{z}_1, \boldsymbol{z}_2((int_1(\boldsymbol{z}_1) \vee int_2(\boldsymbol{z}_1)) \wedge int_1(\boldsymbol{z}_2) \wedge \bigwedge_{i=1}^{2}(\boldsymbol{z}_i = \boldsymbol{x}_i + 1 \vee \boldsymbol{x}_i \curvearrowright \boldsymbol{z}_i))\big)
\end{aligned}$$

Theorem 3.10. *Let (Σ, M) be an OP alphabet. L is accepted by a nondeterministic ωOPBA $\mathcal{A}$ over (Σ, M) if and only if there exists an MSO sentence φ such that $L = L^\omega(\varphi)$.*

The construction of a nondeterministic ωOPBA equivalent to an MSO formula is identical to the one given for finite strings.

The converse construction also follows essentially the same path as in the case of finite-length languages; thus, we only point out the relevant differences w.r.t. the construction of Section 2.4. Formula φ is defined as

$$\varphi := \begin{array}{l} \exists \boldsymbol{A}_0, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_N \\ \exists \boldsymbol{B}_0, \boldsymbol{B}_1, \ldots, \boldsymbol{B}_N \\ \exists \boldsymbol{C}_0, \boldsymbol{C}_1, \ldots, \boldsymbol{C}_N \end{array} \left(\bigvee_{q_i \in I} \mathrm{Start}_i \wedge \varphi_\delta \wedge \varphi_{unique} \wedge \bigvee_{q_f \in F} \mathrm{Accept}_f \right), \tag{10}$$

where Start_i si defined as in Section 2.4, and Accept_f is a shortcut representing the Büchi acceptance condition (a final state is reached infinitely often):

$$\mathrm{Accept}_f := \forall \boldsymbol{x} \exists \boldsymbol{y} (\boldsymbol{x} < \boldsymbol{y} \wedge \boldsymbol{y} \in Q_f).$$

Formula φ_δ encodes the nondeterministic transition functions of the automaton and is obtained from formula $\varphi_{\delta_{\text{push}}} \wedge \varphi_{\delta_{\text{shift}}} \wedge \varphi_{\delta_{\text{pop}}}$ defined in Section 2.4, by replacing expressions as $q_k = \delta(\dots)$ by expressions as $q_k \in \delta(\dots)$. Finally, formula φ_{unique} is defined as the conjunction of the following formulae:

$$\varphi_{uniqueA} := \forall \boldsymbol{x} \bigwedge_{i=0}^{N} \left(\boldsymbol{x} \in \boldsymbol{A}_i \Rightarrow \neg \bigvee_{j=0}^{N} (j \neq i \wedge \boldsymbol{x} \in \boldsymbol{A}_j) \right)$$

$$\varphi_{unique\_next} := \forall \boldsymbol{x}, \boldsymbol{y} \bigwedge_{k=0}^{N} \left(\mathrm{Next}_k(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow \neg \bigvee_{j \neq k} \mathrm{Next}_j(\boldsymbol{x}, \boldsymbol{y}) \right)$$

Such formula was not necessary in the finite case because it was implied by the determinism of the automaton.

The proof that formula φ is satisfied by all and only the words accepted by $\mathcal{A}$ is based on Lemmata 2.5 and 2.6, but we need some more properties to cope with infinite words.

Any ω-word $w \in \Sigma^\omega$ compatible with M can be factored, as in the proof of Theorem 3.5, as a sequence $w = w_1 w_2 w_3 \dots$ where either $w_i \in \Sigma$ is a pending letter, or w_i is the body of the chain ${}^{a_i}[w_i]^{b_i}$, where a_i is the last pending letter before w_i and b_i is the first symbol of w_{i+1}. A similar factorization holds for a finite word $\#w$ without end delimiter. We denote by $\boldsymbol{P}$ the set of positions in a (finite or infinite) string w that correspond to pending letters and by $\boldsymbol{E}$ the set of positions of the right delimiter of the chains of the factorization. These two sets are not necessarily disjoint, and ***EP*** is their union.

$$\begin{aligned} \boldsymbol{z} \in \boldsymbol{P} &:= \forall \boldsymbol{x}, \boldsymbol{y}\ (\boldsymbol{x} < \boldsymbol{z} < \boldsymbol{y} \wedge \boldsymbol{x} \curvearrowright \boldsymbol{y} \Rightarrow \#(\boldsymbol{y})) \\ \boldsymbol{z} \in \boldsymbol{E} &:= \exists \boldsymbol{x}\ (\boldsymbol{x} \in \boldsymbol{P} \wedge \boldsymbol{x} \curvearrowright \boldsymbol{z}) \\ \boldsymbol{z} \in \boldsymbol{EP} &:= \boldsymbol{z} \in \boldsymbol{P} \vee \boldsymbol{z} \in \boldsymbol{E} \end{aligned}$$

Any prefix of an infinite string w which ends in an EP position of w is called *EP-prefix* of w.

Let us define

$$\psi_{i,k}(\boldsymbol{A}_0, \ldots, \boldsymbol{A}_N, \boldsymbol{B}_0, \ldots \boldsymbol{B}_N, \boldsymbol{C}_0, \ldots, \boldsymbol{C}_N) \ := \mathrm{Start}_i \wedge \varphi'_\delta \wedge \mathrm{Final}_k$$

where

$$\mathrm{Final}_k := \exists \boldsymbol{y} \exists \boldsymbol{e}\ (\ \boldsymbol{y} \in Q_k\ \wedge\ \boldsymbol{y} \leq \boldsymbol{e}\ \wedge\ \boldsymbol{e} \in \boldsymbol{EP}\ \wedge\ \forall \boldsymbol{z}(\boldsymbol{y} \leq \boldsymbol{z}\ \wedge\ \boldsymbol{z} \in \boldsymbol{EP} \Rightarrow \boldsymbol{z} = \boldsymbol{e}))$$

and φ'_δ is as φ_δ except for the formula $\varphi_{\delta_{pop}}$, where the constraint $\neg\#(\boldsymbol{y})$ is conjuncted to the antecedent of $\varphi_{\delta_{pop\_fw}}$, and $\varphi_{\delta_{pop\_bwB}}$ and $\varphi_{\delta_{pop\_bwC}}$ are replaced by the unique formula

$$\varphi_{pop\_bw} := \forall \boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y} \bigwedge_{k=0}^{N} \left(\begin{array}{c} \boldsymbol{x} \in \boldsymbol{B}_k \wedge \boldsymbol{v} \in \boldsymbol{C}_k \\ \wedge \\ \neg\#(\boldsymbol{y}) \wedge \mathrm{Tree}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y}) \end{array} \Rightarrow \bigvee_{i=0}^{N} \bigvee_{j=0}^{N} \left(\begin{array}{c} \mathrm{Tree}_{i,j}(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{v}, \boldsymbol{y}) \\ \wedge \\ \delta_{\mathrm{pop}}(q_i, q_j) \ni q_k \end{array} \right) \right)$$

We will interpret formula $\psi_{i,k}$ over finite strings. More precisely, let w' be an EP-prefix of a string $w \in \Sigma^\omega$. It is $w \models_\omega \varphi$ if and only if there exist an initial state q_i, a final state q_f, and an assignment $A_0, \ldots, C_N$ such that $w' \models \psi_{i,f}(A_0, \ldots, C_N)$ for an infinite number of EP-prefixes w' of w. In this case, a position $\boldsymbol{x}$ in a prefix w' may start a chain that goes beyond the end of w', hence in such cases $\boldsymbol{x}$ is in $\boldsymbol{B}_k$ in the assigment satisfying $w \models_\omega \varphi$ but $w' \not\models_\omega \varphi_{pop\_bwB}$. This is the reason why we replace the backward formulae of $\varphi_{\delta_{\mathrm{pop}}}$ in φ'_δ.

For any assignment for $\boldsymbol{A}_0, \ldots, \boldsymbol{C}_N$, it is $w' \models \psi_{i,k}(\boldsymbol{A}_0, \ldots, \boldsymbol{C}_N)$ if and only if there exists a run of $\mathcal{A}$ for w' beginning from state q_i that visits state q_k somewhere after the last EP position before $|w'|$. The run can be built reasoning as in Lemmata 2.5 and 2.6 within the chains of the factorization, and using formulae $\varphi_{\delta_{\text{push}}}$ and $\varphi_{\delta_{\text{shift}}}$ for the positions of pending letters. The properties corresponding to states q_i and q_k are provided by formulae Start_i and Final_k. If w' and w'' are EP-prefixes of w and both satisfy $\psi_{i,k}$ with the same assignment to $\boldsymbol{A}_0, \ldots, \boldsymbol{C}_N$, then the corresponding runs built with such a construction are one the prefix of the other.

Hence $w \models_\omega \varphi$ if and only if there exist infinitely many (finite) runs of $\mathcal{A}$ on EP-prefixes of w, each of them beginning from q_i and visiting the same final state q_f somewhere after its last EP position; such runs are all prefixes of the same infinite run ρ.

Furthermore, since there is a move in ρ that reaches q_f while reading the suffix of each of those EP-prefixes after its last EP position, then ρ traverses infinitely often q_f, and hence ρ is accepting for $\mathcal{A}$.

Symmetrically, one can prove that if there exists an accepting run ρ for an ω-string w in $\mathcal{A}$, then $w \models_\omega \varphi$.

4 A FIRST-ORDER LOGIC FOR FREE LANGUAGES

The traditional MSO logic characterization of regular languages, which has been extended to larger classes such as VPLs and OPLs, is in general considered of intractable complexity for system verification; thus, the literature exhibits a fairly wide variety of language subclasses that are characterized in terms of simpler logics such as fragments of first-order logics or temporal ones. For instance the equivalence between star-free regular languages and Linear Temporal Logic (LTL) is proved in [58]; [8] characterizes classes of VPLs by means of various first-order and temporal logics.

[65], instead, presents a logical characterization of the class of context-free languages by means of a first-order logic, although extended with a quantification over matchings.

In this chapter we move a first step towards accomplishing a similar job with OPLs. We consider free grammars (FrGs) and languages (FrLs), which have been introduced with the main propose of supporting grammar inference [36, 35] for programming languages. *Grammatical inference* (or *induction*) is an active and rich field of research, where various kinds of machine learning techniques are employed to infer a formal grammar or a variant of finite state machine from a set of observations, thus constructing a model which accounts for the characteristics of the observed objects. We refer the interested reader to the recent comprehensive works [40, 43].

FrGs suffer from large size since their nonterminal alphabet is based on the power set of their terminal one; however they can be easily inferred on the basis of positive samples only, and can be minimized (by losing the property of being free) by applying classical algorithms [75, 22]. In this chapter we show that they are well suited to describe various language types, not only in the realm of programming languages. Furthermore, they can be used to define a sort of "superlanguage", possibly inferred in the limit from a set of strings of the user's desired language, and that can be further refined by imposing a few restricting properties in terms of first-order formulae.

The main result of this chapter is that FrL strings satisfy formulae written in a first-order logic that restricts the MSO one defined for general OPLs, and the structure over which such formulae are interpreted is the same as the one defined for general OPLs.

The chapter is organized as follows. In Section 4.1 we resume the basic definitions and properties of FrGs and languages. In Section 4.2 we provide a few simple examples of FrLs with the purpose of showing their usefulness in describing several types of languages, and we prove some of their properties. In Section 4.3 we introduce a first-order logic that defines FrLs. Finally, in Section 4.4 we discuss some related works.

4.1 PRELIMINARIES

We first introduce some notation that will be needed in the rest of the chapter.

Let $G = (N, \Sigma, P, S)$ be an OG. The definition of left and right terminal sets is extended from nonterminals to generic strings α over $(N \cup \Sigma)^*$ as follows:

$$\mathcal{L}(\alpha) = \begin{cases} \{a \in \Sigma \mid A \overset{*}{\Rightarrow} Ba\alpha\} & \text{if } \alpha = A \\ \{a\} & \text{if } \alpha = a\beta \\ \mathcal{L}(A) \cup \{a\} & \text{if } \alpha = Aa\beta \end{cases} \quad \mathcal{R}(\alpha) = \begin{cases} \{a \in \Sigma \mid A \overset{*}{\Rightarrow} \alpha aB\} & \text{if } \alpha = A \\ \{a\} & \text{if } \alpha = \beta a \\ \mathcal{R}(A) \cup \{a\} & \text{if } \alpha = \beta aA \end{cases}$$

where $A \in N$, $B \in N \cup \{\varepsilon\}$, $a \in \Sigma, \beta \in (N \cup \Sigma)^*$.

The following definitions are from [35].

Definition 4.1 (Free Grammar and Language [35])**.** Let G be an OPG with no renaming rules and no empty rule except possibly $C \to \varepsilon$, where C is an axiom not used elsewhere; G is a *free grammar* (FrG) iff the two following conditions hold

- for every production $A \to \alpha$, with $\alpha \neq \varepsilon$, $\mathcal{L}(A) = \mathcal{L}(\alpha)$ and $\mathcal{R}(A) = \mathcal{R}(\alpha)$, i.e., for every nonterminal symbol A, all of its alternative non-empty rules have the same pairs of left and right terminal sets;
- for every nonterminals A, B, $\mathcal{L}(A) = \mathcal{L}(B)$ and $\mathcal{R}(A) = \mathcal{R}(B)$ implies $A = B$.

A language generated by a FrG is a *free language* (FrL).

Notice that, by definition, a FrG is in Fischer normal form. Also, each nonterminal A is uniquely identified by the pair of sets $\mathcal{L}(A), \mathcal{R}(A)$; thus N is isomorphic to $\wp(\Sigma) \times \wp(\Sigma)$. Indeed, it is customary to use $\wp(\Sigma) \times \wp(\Sigma)$ as the nonterminal alphabet of a free grammar.

FrLs can also be defined in terms of a suitable automata family and extended to ω-languages in a similar way as it has been done for general OPLs.

Definition 4.2 (Maxgrammar [35])**.** Given an OPM M, the *maxgrammar* associated with M is the free grammar that contains all productions that are compatible with M, i.e., the productions that induce all and only the relations in M.

Notice that the maxgrammar associated with a complete OPM (i.e., an OPM with no empty case) generates the language Σ^*. The maxgrammar associated with an OPM is unique thanks to the hypothesis of $\doteq$-acyclicity or, in general, if we require that the length of the r.h.s. of the rules is a priori bounded. Also, the set of FrGs with a given OPM is a lattice whose top element is the maxgrammar associated with the matrix [35]. In [35] it is also shown that each free grammar is the top element of a Boolean algebra and that the whole family of OPLs compatible with a given OPM is itself a Boolean algebra whose top element is the language generated by the maxgrammar.

4.2 EXAMPLES AND FIRST PROPERTIES OF FREE LANGUAGES

In this section we investigate the generative power of free grammars: the following examples show that they are well suited to formalize some typical programming language features and various types of system behavior; we will also show that the class of FrLs is not comparable with other subclasses of OPLs such as, e.g., VPLs.

Furthermore, the examples below show that FrGs are not intended to be built by hand; being driven by the powerset of Σ, both N and P may suffer from combinatorial explosion. However, according to their original motivation to support grammar inference, they are well suited to be easily built by some automatic device: in fact the grammars of the following examples have been automatically generated.[10]

Example 4.1. The FrG G depicted in Figure 28 with its OPM generates unparenthesized arithmetic expressions with the usual precedences of $\times$ w.r.t. $+$, which cannot instead be expressed as a VPL. This grammar is obtained from the maxgrammar associated with the OPM by taking only those nonterminals that have letter n in both left and right sets. By this way we guarantee that all strings generated by the grammar begin and end with an n, and are thus well formed. All nonterminals of the grammar are axioms too.

Extending the above grammar to generate also parenthesized arithmetic expressions is a conceptually easy exercise since we only need new nonterminals, and corresponding rules, including $⦇$ and $⦈$ in their left and right terminal sets, respectively. The corresponding FrG has 22 nonterminals and 168 rules, and it can be found among the examples available in the Flup package [1].

$$\begin{array}{rcl}
\langle\{n\},\{n\}\rangle & \to & n \\
\langle\{+,\times,n\},\{+,n\}\rangle & \to & \langle\{\times,n\},\{\times,n\}\rangle + \langle\{n\},\{n\}\rangle \\
\langle\{+,n\},\{+,n\}\rangle & \to & \langle\{+,n\},\{+,\times,n\}\rangle + \langle\{n\},\{n\}\rangle \\
\langle\{+,n\},\{+,\times,n\}\rangle & \to & \langle\{+,n\},\{+,n\}\rangle + \langle\{\times,n\},\{\times,n\}\rangle \\
\langle\{+,\times,n\},\{+,\times,n\}\rangle & \to & \langle\{+,\times,n\},\{+,\times,n\}\rangle + \langle\{\times,n\},\{\times,n\}\rangle \\
\langle\{\times,n\},\{\times,n\}\rangle & \to & \langle\{\times,n\},\{\times,n\}\rangle \times \langle\{n\},\{n\}\rangle \\
\langle\{+,n\},\{+,\times,n\}\rangle & \to & \langle\{+,n\},\{+,\times,n\}\rangle + \langle\{\times,n\},\{\times,n\}\rangle \\
\langle\{+,\times,n\},\{+,n\}\rangle & \to & \langle\{+,\times,n\},\{+,n\}\rangle + \langle\{n\},\{n\}\rangle \\
\langle\{+,\times,n\},\{+,\times,n\}\rangle & \to & \langle\{+,\times,n\},\{+,n\}\rangle + \langle\{\times,n\},\{\times,n\}\rangle \\
\langle\{+,\times,n\},\{+,\times,n\}\rangle & \to & \langle\{\times,n\},\{\times,n\}\rangle + \langle\{\times,n\},\{\times,n\}\rangle \\
\langle\{+,\times,n\},\{+,n\}\rangle & \to & \langle\{+,\times,n\},\{+,\times,n\}\rangle + \langle\{n\},\{n\}\rangle \\
\langle\{+,n\},\{+,n\}\rangle & \to & \langle\{n\},\{n\}\rangle + \langle\{n\},\{n\}\rangle \\
\langle\{+,n\},\{+,\times,n\}\rangle & \to & \langle\{n\},\{n\}\rangle + \langle\{\times,n\},\{\times,n\}\rangle \\
\langle\{\times,n\},\{\times,n\}\rangle & \to & \langle\{n\},\{n\}\rangle \times \langle\{n\},\{n\}\rangle \\
\langle\{+,n\},\{+,n\}\rangle & \to & \langle\{+,n\},\{+,n\}\rangle + \langle\{n\},\{n\}\rangle
\end{array}$$

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

Figure 28.: A FrG for unparenthesized arithmetic expressions and its OPM.

10 The grammars presented here have been produced by the Flup tool (the whole package, which includes various utilities for the general class of OPLs, is available at [1]). In the future we plan to couple Flup with an additional tool that minimizes the original grammar by applying the classical procedure introduced in [75].

Example 4.2. Consider a simplified version of the software system described in Example 3.1, which serves requests of operations issued by various users but subject to possible asynchronous interrupts. We consider a system that does not work forever and we assume that there is only one type of interrupt.

We model the behavior of the system by introducing an alphabet with a pair of symbols *call*, *ret*, to describe the request and completion of a user's operation, and symbol *int*, denoting the occurrence of the interrupt. Under normal behavior *call*s and *ret*s must be matched according to the normal LIFO policy; however, if an interrupt occurs when some *call*s are pending, they are reset without waiting for the corresponding *ret*s; possible subsequent *ret*s remain therefore unmatched. Unmatched returns can occur only if previously some interrupt flushed away all unmatched calls.

A FrG that generates sequences of operations and occurrences of interrupts consistent with the above informal description has the OPM displayed in Figure 29. The same Figure shows a sample of productions of the FrG, which counts 21 nonterminals and 174 rules overall. It has been built starting from the maxgrammar associated with the OPM by taking as nonterminals only $\langle\{ret\}, \{ret\}\rangle$ and those that do not contain *ret* in their left set. The axioms are all nonterminals $A \in (\wp(\Sigma) \times \wp(\Sigma)) \setminus \{\langle\{ret\}, \{ret\}\rangle\}$. Nonterminal $\langle\{ret\}, \{ret\}\rangle$ is necessary to generate sequences of unmatched returns; the constraint on the other nonterminals guarantees that a sequence of *ret*s is either matched by corresponding previous *call*s or is unmatched but preceded by an interrupt. This FrG too can be found in the examples in the Flup package.

The resulting grammar can be easily modified to deal with more complex policies, e.g., different levels of interrupt, but with a possible consequent size increase.

All the FrGs in the above examples have been built by applying a top-down approach, starting from the maxgrammar associated with the OPM and "pruning" nonterminals and productions that would generate undesired strings. This approach complements the bottom up technique of traditional grammar inference, which builds a FrG generating a desired language by abstracting away from a given sample of language strings (it exploits the distinguishing property of FrGs that they can be inferred in the limit on the basis of a positive sample only [36]).

$$
\begin{array}{rcl}
\langle\{int\},\{int\}\rangle & \to & int \\
\langle\{ret\},\{ret\}\rangle & \to & ret \\
\langle\{call\},\{call\}\rangle & \to & call \\
\langle\{call\},\{ret\}\rangle & \to & call\ ret \\
\langle\{int,call\},\{int\}\rangle & \to & \langle\{call\},\{call,ret\}\rangle\ int \mid \\
 & & \langle\{call,int\},\{call,ret\}\rangle\ int \\
\langle\{call,int\},\{call\}\rangle & \to & \langle\{int\},\{int,ret\}\rangle\ call \mid \\
 & & \langle\{int,call\},\{int,ret\}\rangle\ call \\
\langle\{call,int\},\{ret\}\rangle & \to & \langle\{int\},\{int\}\rangle\ call\ ret \mid \\
 & & \langle\{int,call\},\{int\}\rangle\ call\ ret \mid \\
 & & \langle\{call,int\},\{ret\}\rangle\ call\ ret \mid \\
 & & \langle\{int\},\{int,ret\}\rangle\ call\ ret \\
\langle\{call,int\},\{call,ret\}\rangle & \to & \langle\{int\},\{int\}\rangle\ call\ \langle\{call\},\{ret\}\rangle \\
\langle\{int,call\},\{int,ret\}\rangle & \to & \langle\{call\},\{call\}\rangle\ int\ \langle\{ret\},\{ret\}\rangle \mid \\
 & & \langle\{call,int\},\{call,ret\}\rangle\ int\ \langle\{int\},\{int,ret\}\rangle
\end{array}
$$

	call	*ret*	*int*
call	$\lessdot$	$\doteq$	$\gtrdot$
ret	$\gtrdot$	$\gtrdot$	$\gtrdot$
int	$\gtrdot$	$\lessdot$	$\lessdot$

Figure 29.: A sample of rules of the FrG for the language of Example 4.2 and its OPM.

The typical canonical form of FrGs makes also easy the application of the classical minimization procedure that extends to structure grammars the minimization of finite state machines [75, 22].

The above examples also help comparing the generative power of FrLs with other subclasses of OPLs.

Proposition 4.1. *The class of FrLs is incomparable with the classes of regular languages and VPLs.*

Proof. The language described in Example 4.2 is a FrL but is not regular, due to the necessity to match corresponding *call* and *ret* symbols, nor a VPL: although, in fact, it retains the rationale of VPLs in that it allows for unmatched "parenthesis-like" symbols (calls and returns in this case), it generalizes this VPLs feature in that such unmatched symbols can occur even inside a matching pair, which is impossible in VPLs. On the other hand, it is known that FrGs generate only non-counting languages [33], whereas regular languages and VPLs, which strictly contain regular ones, can be counting [76]. □

Proposition 4.2. *FrLs (with a fixed OPM) are closed w.r.t. intersection [33] but not w.r.t. concatenation, complement, union and Kleene *.*

Proof. Closure under intersection has been already stated in [33], and follows from the fact that, given an OPM, the parsing of any string w is the same for any FrG (all FrG's nonterminal alphabets are pairs of subsets of Σ); it follows that $L(G_1) \cap L(G_2) = L(G_1 \cap G_2)$ where $G_1 \cap G_2$ denotes the grammar whose production set is the intersection of the production sets of G_1 and G_2 (possibly "cleaned up" of the useless productions) and is a FrG.

To prove that FrLs are not closed w.r.t. concatenation, consider language $L = \{a\}$ with $a \lessdot a$. L is a FrL but $L \cdot L$ is not: to generate $\# \lessdot a \lessdot a \gtrdot \#$ a FrG needs the productions $\langle\{a\},\{a\}\rangle \to a$ and $\langle\{a\},\{a\}\rangle \to a\langle\{a\},\{a\}\rangle$ which generate a^+. For the same reason $\neg L = \{a^n \mid n > 1 \vee n = 0\}$ is not a FrL; thus FrLs are not closed w.r.t. complement.

Consider the FrGs G_1 and G_2 below (both grammars have axiom $\langle\{a,b\},\{b\}\rangle$):

$$\begin{array}{lcl@{\qquad}lcl}
G_1: & & & G_2: & & \\
\langle\{a,b\},\{b\}\rangle & \to & \langle\{a,b\},\{b\}\rangle\, b \mid \langle\{a\},\{a\}\rangle\, b & \langle\{a,b\},\{b\}\rangle & \to & \langle\{a\},\{a\}\rangle\, b \\
\langle\{a\},\{a\}\rangle & \to & a & \langle\{a\},\{a\}\rangle & \to & a \mid \langle\{a\},\{a\}\rangle\, a
\end{array}$$

which generate, respectively, $L_1 = ab^+$ and $L_2 = a^+b$: all productions of G_1 and G_2 are necessary to generate all strings of $L_1 \cup L_2$ but the union of (productions of) G_1 and G_2 generates strings a^+b^+, which do not belong to $L_1 \cup L_2$.

Finally, consider the FrG G:

$$\begin{array}{rcl}
\langle\{a,b\},\{b\}\rangle & \to & \langle\{a,b\},\{a\}\rangle\, b \\
\langle\{a,b\},\{a\}\rangle & \to & \langle\{a,b\},\{b\}\rangle\, a \mid \langle\{b\},\{b\}\rangle\, a \\
\langle\{b\},\{b\}\rangle & \to & b
\end{array}$$

with axiom $\langle\{a,b\},\{b\}\rangle$, which generates $L = (ba)^+b$ (with $a \gtrdot b$, $b \gtrdot b$ and $b \gtrdot a$). To generate a string in L^* we need to generate two consecutive b, corresponding respectively to the last and the first character of two consecutive words of L; this can be obtained only by means of a new rule for a nonterminal with right terminal set $\{b\}$, such as $\langle\{a,b\},\{b\}\rangle \to \langle\{a,b\},\{b\}\rangle b$ or the rule $\langle\{b\},\{b\}\rangle \to \langle\{b\},\{b\}\rangle b$, which however imply the generation also of strings containing any number of consecutive b, which do not belong to L^*. □

Ultimately, the above examples show that on the one hand FrGs can model the essential features of various systems but, on the other hand, they exhibit some unexpected limits in generative power which are not suffered even by regular languages. These limits must be ascribed to their distinguishing property of being inferrable in the limit by using only a set of positive strings (in fact the class of FrLs is not closed under complement). Thus they are better suited to define a sort of "skeleton language" to be refined by superimposing further constraints specified by means of some complementary formalism. A natural way to pursue such an approach is, e.g., to "intersect" them with some finite state machine. Herein, instead, we will exploit the fact that FrLs can be defined in terms of first-order logic sentences, but first-order logic can also be used to define further, even more sophisticated, constraints on these languages.

4.3 FIRST-ORDER LOGIC DEFINABILITY OF FREE LANGUAGES

In this section we show that FrLs can be defined in terms of a FO logic rather than a MSO one. The converse property however does not hold: by Proposition 4.2, in fact, the class of FrLs is not closed under complement; hence, there are languages that can be defined in terms of FO logic but are not FrLs. On the other hand FO formulae can be used to refine FrLs by superimposing further properties.

The key difference between the traditional MSO language formulation and the new FO one is that in the MSO formulation each position in the string (over which the MSO logic formula is interpreted) may be associated with several states of an automaton recognizing the language defined by the MSO formula, i.e., to several second-order variables denoting subsets of positions according to Büchi's approach; in our FO formulation instead, we associate positions with the left and right terminal sets of the nonterminal of a FrG that is the root of the subtree whose leftmost and rightmost leaves are in the given positions. Thanks to the fact that in FrGs the number of possible nonterminals is a priori bounded and they are univocally identified by their left and right terminal sets, we can express such association by means of first-order formulae, without the need to resort to second-order variables denoting sets of positions.

We now introduce the syntax of our FO logic and we then prove that, for every FrG, a FO sentence can be automatically built that is satisfied by all and only the strings generated by the grammar.

Definition 4.3 (First-order Logic over (Σ, M))**.** Let (Σ, M) be an OP alphabet, and let $\mathcal{V}$ be a countable infinite set of first-order variables (denoted by $\boldsymbol{x}, \boldsymbol{y}, \dots$). The $\mathrm{FO}_{\Sigma,M}$ (*first-order logic* over (Σ, M)) is defined by the following syntax:

$$\varphi := c(\boldsymbol{x}) \mid \boldsymbol{x} \leq \boldsymbol{y} \mid \boldsymbol{x} \curvearrowright \boldsymbol{y} \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists \boldsymbol{x}.\varphi$$

where $c \in \Sigma \cup \{\#\}$, $\boldsymbol{x}, \boldsymbol{y} \in \mathcal{V}$.

A $\mathrm{FO}_{\Sigma,M}$ formula is interpreted over a (Σ, M) string w, with respect to assignments $v : \mathcal{V} \to \{0, 1, \dots, |w| + 1\}$ analogously to $\mathrm{MSO}_{\Sigma,M}$ formulae (Definition 2.10) with $\mathcal{V}_2 = \emptyset$.

Example 4.3. Consider the OP alphabet given in Figure 30. In all strings compatible with M, such that ${}^{\#}[w]^{\#}$ is a chain, all parentheses are well-matched.

The sentence in Figure 31 restricts the set of strings compatible with the OPM to the language where parentheses are used only when they are needed (i.e., to invert the natural precedence between $\times$ and $+$).

	$+$	$\times$	$\llparenthesis$	$\rrparenthesis$	n	$\#$
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\times$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
$\llparenthesis$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
$\rrparenthesis$	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
n	$\gtrdot$	$\gtrdot$		$\gtrdot$		$\gtrdot$
$\#$	$\lessdot$	$\lessdot$	$\lessdot$		$\lessdot$	$\doteq$

Figure 30.: An OP alphabet (Σ, M) for arithmetic expressions.

We now state our main result of this chapter.

$$\forall \boldsymbol{x} \forall \boldsymbol{y} \left(\begin{array}{c} \boldsymbol{x} \curvearrowright \boldsymbol{y} \wedge \\ \llparenthesis(\boldsymbol{x}+1) \wedge \\ \rrparenthesis(\boldsymbol{y}-1) \end{array} \Rightarrow \exists \boldsymbol{z} \left(\begin{array}{c} (\times(\boldsymbol{x}) \vee \times(\boldsymbol{y})) \\ \wedge \\ \boldsymbol{x}+1 < \boldsymbol{z} < \boldsymbol{y}-1 \wedge +(\boldsymbol{z}) \wedge \\ \neg \exists \boldsymbol{u} \exists \boldsymbol{v} \left(\begin{array}{c} \boldsymbol{x}+1 < \boldsymbol{u} < \boldsymbol{z} \wedge \llparenthesis(\boldsymbol{u}) \wedge \\ \boldsymbol{z} < \boldsymbol{v} < \boldsymbol{y}-1 \wedge \rrparenthesis(\boldsymbol{v}) \wedge \\ \boldsymbol{u}-1 \curvearrowright \boldsymbol{v}+1 \end{array} \right) \end{array} \right) \right)$$

Figure 31.: A $\mathrm{FO}_{\Sigma,M}$ sentence on the OP alphabet (Σ, M) of Figure 30.

Theorem 4.1. *Let $G = \langle N, \Sigma, P, S \rangle$ be a FrG: then a $FO_{\Sigma,M}$ formula ψ can be effectively built such that $w \in L(G)$ iff $w \models \psi$.*

Proof. We first introduce some shortcut notation to make formulae more compact and understandable.

As in Section 2.4.3, when considering a chain ${}^{a}[w]^{b}$, 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_i may be empty). We denote by s_i the position of symbol a_i, for $i = 1, 2, \ldots, \ell$ and set $a_0 = a$, $s_0 = 0$, $a_{\ell+1} = b$, and $s_{\ell+1} = |w| + 1$.

Notation TreeC is defined as follows ($n > 1$):

$$\mathrm{TreeC}(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1}) := \boldsymbol{x}_0 \curvearrowright \boldsymbol{x}_{n+1} \wedge \bigwedge_{0 \le i \le n} \left(\begin{array}{c} \boldsymbol{x}_i + 1 = \boldsymbol{x}_{i+1} \\ \vee \\ \boldsymbol{x}_i \curvearrowright \boldsymbol{x}_{i+1} \end{array} \wedge \bigwedge_{i+1 < j \le n} \neg(\boldsymbol{x}_i \curvearrowright \boldsymbol{x}_j) \right)$$

If $\boldsymbol{x}_0 \curvearrowright \boldsymbol{x}_{n+1}$, there exist (unique) $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1}$ such that $\mathrm{TreeC}(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{x}_{n+1})$ holds: in particular, $\boldsymbol{x}_0 \lessdot \boldsymbol{x}_1$, $\boldsymbol{x}_i \doteq \boldsymbol{x}_{i+1}$ for $1 \le i \le n-1$, and $\boldsymbol{x}_n \gtrdot \boldsymbol{x}_{n+1}$.

Let w be a chain body $w = w_0 a_1 w_1 a_2 \ldots a_\ell w_\ell$: if every w_i is empty (the chain is simple), then $0 \curvearrowright \ell + 1$ and $\mathrm{TreeC}(0, 1, 2, \ldots, \ell, \ell + 1)$ holds; if w is the body of a composed chain, then $0 \curvearrowright |w| + 1$ and $\mathrm{TreeC}(s_0, s_1, s_2, \ldots, s_\ell, s_{\ell+1})$ holds (see Figure 32).

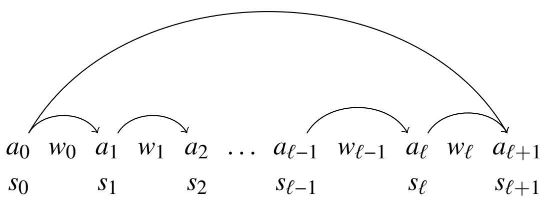

a_0 w_0 a_1 w_1 a_2 ... $a_{\ell-1}$ $w_{\ell-1}$ a_ℓ w_ℓ $a_{\ell+1}$
s_0 s_1 s_2 $s_{\ell-1}$ s_ℓ $s_{\ell+1}$

Figure 32.: Chain ${}^{a_0}[w_0a_1w_1a_2\dots a_\ell w_\ell]^{a_{\ell+1}}$, for which TreeC$(s_0,s_1,s_2,\dots,s_\ell,s_{\ell+1})$ holds.

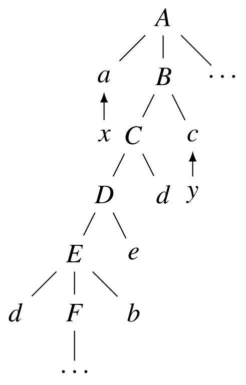


Figure 33.: Pair of positions $\boldsymbol{x},\boldsymbol{y}$ for which $\mathcal{L}_{\{d,e\}}(\boldsymbol{x},\boldsymbol{y})$ holds.

We consider notation Tree as it has been defined in Section 2.4.3, and it is reported here again for convenience.

$$\text{Tree}(\boldsymbol{x},\boldsymbol{u},\boldsymbol{v},\boldsymbol{y}) := \boldsymbol{x} \curvearrowright \boldsymbol{y} \wedge \begin{pmatrix} (\boldsymbol{x}+1=\boldsymbol{u} \ \vee \ \boldsymbol{x} \curvearrowright \boldsymbol{u}) \wedge \neg\exists \boldsymbol{t}(\boldsymbol{u}<\boldsymbol{t}<\boldsymbol{y} \wedge \boldsymbol{x} \curvearrowright \boldsymbol{t}) \\ \wedge \\ (\boldsymbol{v}+1=\boldsymbol{y} \ \vee \ \boldsymbol{v} \curvearrowright \boldsymbol{y}) \wedge \neg\exists \boldsymbol{t}(\boldsymbol{x}<\boldsymbol{t}<\boldsymbol{v} \wedge \boldsymbol{t} \curvearrowright \boldsymbol{y}) \end{pmatrix}$$

This notation represents a "projection" of TreeC over positions $\boldsymbol{x}_0$, $\boldsymbol{x}_1$, $\boldsymbol{x}_n$ and $\boldsymbol{x}_{n+1}$ (here corresponding to $\boldsymbol{x},\boldsymbol{u},\boldsymbol{v},\boldsymbol{y}$), and is used when we do not need to refer to positions $\boldsymbol{x}_2,\dots,\boldsymbol{x}_{n-1}$ within a chain.

Also, for every $A \subseteq \Sigma$, we define notations:

$$\mathcal{L}_A(\boldsymbol{x},\boldsymbol{y}) := \begin{pmatrix} \forall \boldsymbol{u},\boldsymbol{v},\boldsymbol{z}\left(\boldsymbol{u} \le \boldsymbol{v} < \boldsymbol{z} \le \boldsymbol{y} \wedge \text{Tree}(\boldsymbol{x},\boldsymbol{u},\boldsymbol{v},\boldsymbol{z}) \Rightarrow \bigvee_{a\in A} a(\boldsymbol{u})\right) \\ \wedge \\ \bigwedge_{a\in A} \exists\, \boldsymbol{u},\boldsymbol{v},\boldsymbol{z}\,(\boldsymbol{u} \le \boldsymbol{v} < \boldsymbol{z} \le \boldsymbol{y} \wedge \text{Tree}(\boldsymbol{x},\boldsymbol{u},\boldsymbol{v},\boldsymbol{z}) \wedge a(\boldsymbol{u})) \end{pmatrix}$$

$$\mathcal{R}_A(\boldsymbol{x},\boldsymbol{y}) := \left(\begin{array}{c} \forall \boldsymbol{u},\boldsymbol{v},\boldsymbol{z} \left(\boldsymbol{x} \le \boldsymbol{u} < \boldsymbol{v} \le \boldsymbol{z} \wedge \mathrm{Tree}(\boldsymbol{u},\boldsymbol{v},\boldsymbol{z},\boldsymbol{y}) \Rightarrow \bigvee_{a \in A} a(\boldsymbol{z}) \right) \\ \wedge \\ \bigwedge_{a \in A} \exists\, \boldsymbol{u},\boldsymbol{v},\boldsymbol{z} \, (\boldsymbol{x} \le \boldsymbol{u} < \boldsymbol{v} \le \boldsymbol{z} \wedge \mathrm{Tree}(\boldsymbol{u},\boldsymbol{v},\boldsymbol{z},\boldsymbol{y}) \wedge a(\boldsymbol{z})) \end{array} \right)$$

For instance, with reference to Figure 33, for positions $\boldsymbol{x},\boldsymbol{y}$, $\mathcal{L}_{\{d,e\}}(\boldsymbol{x},\boldsymbol{y})$ holds. Notice that for each pair of positions $\boldsymbol{x},\boldsymbol{y}$ there exists a unique pair of sets A, B such that $\mathcal{L}_A(\boldsymbol{x},\boldsymbol{y})$ and $\mathcal{R}_B(\boldsymbol{x},\boldsymbol{y})$ hold true.

Furthermore, for every $\langle L, R\rangle \in \Gamma$, we add notation $P_{\langle L,R\rangle}(\boldsymbol{x},\boldsymbol{y})$, which represents the *terminal profile* of the chain, if any, between positions $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$P_{\langle L,R\rangle}(\boldsymbol{x},\boldsymbol{y}) := \boldsymbol{x} \curvearrowright \boldsymbol{y} \wedge \mathcal{L}_L(\boldsymbol{x},\boldsymbol{y}) \wedge \mathcal{R}_R(\boldsymbol{x},\boldsymbol{y})$$

Intuitively, $P_{\langle L,R\rangle}(\boldsymbol{x},\boldsymbol{y})$ holds iff, in the syntax tree, the chain between positions $\boldsymbol{x}$ and $\boldsymbol{y}$ is the frontier of a subtree that has as root nonterminal $\langle L, R\rangle$.

Finally, for every $\langle L, R\rangle \in \Gamma$, set

$$\psi_{\langle L,R\rangle} := \forall \boldsymbol{x},\boldsymbol{y} \left(\begin{array}{c} P_{\langle L,R\rangle}(\boldsymbol{x},\boldsymbol{y}) \\ \Rightarrow \\ \displaystyle\bigvee_{\langle L,R\rangle \to \langle L_0,R_0\rangle c_1 \langle L_1,R_1\rangle c_2 \ldots c_k \langle L_k,R_k\rangle} \exists \boldsymbol{x}_1 \ldots \boldsymbol{x}_k \left(\begin{array}{c} \mathrm{TreeC}(\boldsymbol{x},\boldsymbol{x}_1,\ldots,\boldsymbol{x}_k,\boldsymbol{y}) \wedge \\ \displaystyle\bigwedge_{1 \le i \le k} c_i(\boldsymbol{x}_i) \wedge \\ \displaystyle\bigwedge_{\substack{1 \le i \le k-1: \\ \langle L_i,R_i\rangle \ne \varepsilon}} P_{\langle L_i,R_i\rangle}(\boldsymbol{x}_i,\boldsymbol{x}_{i+1}) \wedge \\ \boldsymbol{x}+1 \ne \boldsymbol{x}_1 \Rightarrow P_{\langle L_0,R_0\rangle}(\boldsymbol{x},\boldsymbol{x}_1) \wedge \\ \boldsymbol{x}_k+1 \ne \boldsymbol{y} \Rightarrow P_{\langle L_k,R_k\rangle}(\boldsymbol{x}_k,\boldsymbol{y}) \end{array} \right) \end{array} \right)$$

where the disjunction is considered over the rules of G:

$$\rho = \langle L, R\rangle \to \langle L_0, R_0\rangle c_1 \langle L_1, R_1\rangle c_2 \ldots c_k \langle L_k, R_k\rangle,$$

with $\langle L_i, R_i\rangle \in N \cup \{\varepsilon\}$, $0 \le i \le k$, and $L = L_0 \cup \{c_1\}$, $R = R_k \cup \{c_k\}$.

To complete the construction and the proof of Theorem 4.1 we define:

$$\psi := \bigwedge_{\langle A,B\rangle} \psi_{\langle A,B\rangle} \land \exists \boldsymbol{e}\left(\#(\boldsymbol{e}+1) \land \neg\exists \boldsymbol{y}(\boldsymbol{e}+1 < \boldsymbol{y}) \land \bigvee_{\langle L,R\rangle \in S} P_{\langle L,R\rangle}(0, \boldsymbol{e}+1)\right)$$

The proof of the theorem is a direct consequence of the following Lemma 4.1 when $\langle L, R\rangle$ is an axiom of G. □

Lemma 4.1. *For every $\langle L, R\rangle \in N$ and for every body w of a chain, we have $\langle L, R\rangle \overset{*}{\Rightarrow} w$ iff $w \models P_{\langle L,R\rangle}(0, |w| + 1) \land \bigwedge_{\langle A,B\rangle} \psi_{\langle A,B\rangle}$.*

Proof. Consider first the direction from left to right of the lemma. The proof is by induction on the length h of a derivation.

If $h = 1$, then $\langle L, R\rangle \overset{*}{\Rightarrow} w$ implies that $\rho = \langle L, R\rangle \to a_1 a_2 \dots a_l$ is a production of G, and $w = a_1 a_2 \dots a_l$ is the body of a simple chain. G being a FrG, it is $L = \{a_1\}$ and $R = \{a_l\}$. Since $0 \curvearrowright l + 1$ and $w \models \mathcal{L}_{\{a_1\}}(0, l + 1) \land \mathcal{R}_{\{a_l\}}(0, l + 1)$, then $w \models P_{\langle L,R\rangle}(0, l + 1)$.

For every $\langle A, B\rangle \in \Gamma$ and positions $\boldsymbol{x}, \boldsymbol{y}$, $w \models P_{\langle A,B\rangle}(\boldsymbol{x}, \boldsymbol{y})$ holds true only if $\langle A, B\rangle = \langle L, R\rangle$ and $\boldsymbol{x} = 0, \boldsymbol{y} = l + 1$. Furthermore, there exist (unique) $\boldsymbol{x}_1 = 1, \boldsymbol{x}_2 = 2, \dots, \boldsymbol{x}_l = l$ such that $\mathrm{TreeC}(0, 1, \dots, l, l + 1)$ holds, and for every $j = 1, \dots, l$, $a_j(\boldsymbol{x}_j)$ holds true. Thus, $w \models \psi_{\langle A,B\rangle}$ for every $\langle A, B\rangle \in \Gamma$, and $w \models P_{\langle L,R\rangle}(0, |w| + 1) \land \bigwedge_{\langle A,B\rangle} \psi_{\langle A,B\rangle}$.

Assume that this direction of the lemma holds for every derivation of length $\leq h$. Let $\langle L, R\rangle \overset{h+1}{\Rightarrow} w$, with $\langle L, R\rangle \Rightarrow \langle L_0, R_0\rangle a_1 \langle L_1, R_1\rangle a_2 \dots a_l \langle L_l, R_l\rangle$ and, for each $i = 0, 1, \dots, l$, $\langle L_i, R_i\rangle \overset{h_i}{\Rightarrow} w_i$ such that $h_i \leq h$ and $w = w_0 a_1 w_1 \dots a_l w_l$ is the body of a composed chain ($w_i = \varepsilon$ if $\langle L_i, R_i\rangle = \varepsilon$).

By the inductive hypothesis, for every $i = 0, 1 \dots, l$ such that $w_i \neq \varepsilon$, we have $w_i \models P_{\langle L_i,R_i\rangle}(0, |w_i| + 1) \land \bigwedge_{\langle A,B\rangle} \psi_{\langle A,B\rangle}$. Let $\rho = \langle L, R\rangle \to \langle L_0, R_0\rangle a_1 \langle L_1, R_1\rangle a_2 \dots a_l \langle L_l, R_l\rangle$: G being a FrG, we have $L = L_0 \cup \{a_1\}$ and $R = R_k \cup \{a_l\}$; thus $w \models \mathcal{L}_L(0, |w| + 1) \land \mathcal{R}_R(0, |w| + 1)$, and $w \models P_{\langle L,R\rangle}(0, |w| + 1)$. Furthermore, let $\boldsymbol{x}, \boldsymbol{y}$ be positions such that $w \models P_{\langle A,B\rangle}(\boldsymbol{x}, \boldsymbol{y})$ for some $\langle A, B\rangle \in \Gamma$ and $\boldsymbol{x}, \boldsymbol{y}$ are not both inside the same w_i, and they are not s_i and s_{i+1}; then necessarily $\boldsymbol{x} = 0, \boldsymbol{y} = |w| + 1$, and $w \models P_{\langle A,B\rangle}(0, |w| + 1)$ only if $\langle A, B\rangle = \langle L, R\rangle$. Also, there exist $\boldsymbol{x}_0 = 0, \boldsymbol{x}_1 = s_1$,

$\dots, \boldsymbol{x}_l = s_l, \boldsymbol{x}_{l+1} = |w| + 1$ such that $\mathrm{TreeC}(\boldsymbol{x}_0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_l, \boldsymbol{x}_{l+1})$ holds, and for every $j = 1, \dots, l$, $a_j(\boldsymbol{x}_j)$ holds true. Hence, $w \models \bigwedge_{\langle A,B \rangle} \psi_{\langle A,B \rangle}$.

Consider then the direction from right to left of the lemma. The proof is by induction on the depth d of the chain.

If $d = 1$, then $w = a_1 a_2 \dots a_l$ is the body of a simple chain. Since $w \models P_{\langle L,R \rangle}(0, |w| +1)$, then there exist $\rho = \langle L, R \rangle \to \langle L_0, R_0 \rangle c_1 \langle L_1, R_1 \rangle c_2 \dots c_k \langle L_k, R_k \rangle$ and $\boldsymbol{x}_1, \dots, \boldsymbol{x}_k$ such that $\mathrm{TreeC}(0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_k, |w| + 1)$ and $c_j(\boldsymbol{x}_j)$ for every $j = 1, \dots, k$ hold. By definition of TreeC, we have $\boldsymbol{x}_j = j$ for every $j = 1, \dots, k$ and $k = l$, and $a_j = c_j$ for every j. There is, thus, a production of G: $\rho = \langle L, R \rangle \to a_1 a_2 \dots a_l$, and $\langle L, R \rangle \overset{*}{\Rightarrow} w$ holds.

Let now $d > 1$, then $w = w_0 a_1 w_1 \dots a_l w_l$ is the body of a composed chain and s_j $(1 \le j \le l)$ are the unique positions such that $\mathrm{TreeC}(0, s_1, \dots, s_l, |w| + 1)$ holds true. Since $w \models P_{\langle L,R \rangle}(0, |w| + 1) \wedge \bigwedge_{\langle A,B \rangle} \psi_{\langle A,B \rangle}$, then there exists a production ρ of G such that $\rho = \langle L, R \rangle \to \langle L_0, R_0 \rangle c_1 \langle L_1, R_1 \rangle c_2 \dots c_k \langle L_k, R_k \rangle$ and there exist $\boldsymbol{x}_j$ $(1 \le j \le k)$ with $\mathrm{TreeC}(0, \boldsymbol{x}_1, \dots, \boldsymbol{x}_l, |w| + 1)$ and $c_j(\boldsymbol{x}_j)$; thus we have $k = l$ and $c_j = a_j$ for each j. Furthermore, let $\boldsymbol{x}_0 = 0$, $\boldsymbol{x}_{l+1} = |w| + 1$: for every $i = 0, 1 \dots, k$ such that $\langle L_i, R_i \rangle \neq \varepsilon$, $w \models P_{\langle L_i,R_i \rangle}(\boldsymbol{x}_i, \boldsymbol{x}_{i+1})$ holds true, and we have $w_i \models P_{\langle L_i,R_i \rangle}(0, |w_i| + 1) \wedge \bigwedge_{\langle A,B \rangle} \psi_{\langle A,B \rangle}$. By inductive hypothesis, thus there exists in G a derivation $\langle L_i, R_i \rangle \overset{*}{\Rightarrow} w_i$. Hence, $\langle L, R \rangle \Rightarrow \langle L_0, R_0 \rangle a_1 \langle L_1, R_1 \rangle a_2 \dots \langle L_{k-1}, R_{k-1} \rangle a_k \langle L_k, R_k \rangle \overset{*}{\Rightarrow} w$. □

4.4 RELATED WORK

In the last decades an independent branch of research generated a flourishing of new results in terms of logic characterization of language families, ignited by the pioneering results by Büchi and others [24, 78] on the monadic second-order (MSO) logic characterization of regular languages over finite or infinite words (ω-languages) and motivated mainly by the breakthrough application of model-checking, which is rooted in closure properties and decidability of the emptiness problem, besides correspondence between automata-theoretic and logic language characterization. The present state of the art exhibits plenty of language families and related characterization in terms of various forms of logic formalisms as, e.g., first-order, propositional or temporal logics.

The earliest investigations on first-order logic characterizations of language classes date back to [99, 76], which proved respectively that the first-order logic of the successor FO[+1] characterizes the class of locally threshold testable languages and the first-order logic of the linear order FO[<] characterizes the larger class of star-free languages. A fundamental result by Kamp [58] showed that FO[<] is also equivalent to Linear Temporal Logic (LTL), which has long been considered the temporal logic of choice for program verification, because of its expressiveness, conciseness and model checking complexity (linear time in the size of the system and PSPACE in the size of the formula). In [95] Thomas obtained a further characterization for first-order definable languages of infinite words as those accepted by a counter-free or aperiodic Büchi automaton.

First-order logic with predicate $x + y = z$, i.e. FO[+], has been also recently investigated in [28], where they provide a characterization of the bounded languages definable in this logic as the class of semilinear languages. A characterization of logic FO[+] in terms of a class of generating or recognizing devices, instead, is still an open problem.

Recently, new first-order or specialized logical formalisms have been proposed mainly with the goal of extending model-checking techniques, i.e., decidability of system properties, beyond the natural scope of finite state machines. E.g., [5] investigates propositional dynamic logic (PDL) on trees and studies the complexity of its model checking problem; [20] studies the model-checking problem for the same logic on parse trees. [8] has lately introduced some temporal logics for VPLs of finite and infinite length words, which are natural extensions of LTL, and has proved that these logics have the same expressive power as the first-order logic of the successor augmented with the binary predicate $\rightsquigarrow$. This work defines a Nested Word Temporal Logic (NWTL), which extends LTL with a future and past variant of the standard *Until* operator, which is interpreted on paths of positions augmented by edges that link pair of positions labeled by matching call and return alphabet symbols. It provides a tableaux-based model checking algorithm and studies its complexity, which is polynomial in the size of the model and EXPTIME in the size of the formula (precisely, EXPTIME-complete). A less expressive temporal logic for VPLs, named CaRet, which is instead not first-order expressively complete, has been introduced in [9] with the main motivation of specification and verification of non-regular program specifications. [8] also extends CaRet with variants of the classical temporal operators, yielding logics that are also first-order expressively-complete, but with a worse complexity for model checking.

As regards FrLs, since they are not closed under the operations of complementation and intersection, they cannot directly allow for the classical automata-based approach for model checking. Nevertheless, as explained in the previous sections of this chapter, they can be naturally exploited for system specification by exploiting their property of being inferable in the limit on the basis of a positive sample of sentences only and further refining the inferred language by stating, e.g., user's constraints expressed as first-order formulae; the behavior of the resulting system might then be model-checked with a more expressive (first-order) logic.

As future work, we plan to investigate new logic formalisms simpler than MSO logic that characterize suitable subclasses of general OPLs and further study (variants of) our logic, in the same vein as it has been done for regular languages, VPLs and for various cases of tree-languages; in particular, it would be interesting to identify a class of OPLs characterized exactly by the first-order OP logic introduced in Section 4.3.

Part III.

Parallel processing of OPLs

5 PARALLEL SYNTACTIC AND LEXICAL ANALYSIS

The renewed interest in OPLs, which has recently motivated a new flourishing of research on this class of languages, is not only driven by the discovery of their closure and decidability properties, which have been illustrated and investigated for verification purposes in the previous chapters of this thesis.

A second fundamental property enjoyed by this class of languages consists in their local parsability, which is crucial to support data-parallel processing.

In this chapter we illustrate the exploitation of this property for parallel parsing and we complement and streamline this approach to parallel syntactic analysis by parallelizing also the lexical analysis stage. Processing based on semantic analysis –e.g., processing an input document against a user's query– is the object of the next chapter (Chapter 6).

5.1 INTRODUCTION

When one considers the present state of the art on parsing, it is surprising to note that the literature exhibits only a few historical proposals of parallel parsing algorithms which had no follow-up, let alone application, despite the growing amount of data which need to be processed (that has grown to such a pressing need that hardware accelerators have been developed to tackle it [66]). The only, although ad-hoc, exceptions to this trend are some works tackling the parsing of XML [72] and HTML5 [102].

The most likely reason for this lack of practical parallel parsers is the intrinsically sequential nature of the classical deterministic (LR and LL) algorithms. For instance, assume to parse the language $L = L_1^*$, where L_1 is $\{1a^nb^n \mid n \geq 1\} \cup \{0a^nb^{2n} \mid n \geq 1\}$ through a deterministic shift-reduce parallel algorithm. Intuitively, it would be natural to map the parsing of each substring candidate to belonging to L_1 into a separate computation, henceforth called *worker*, and then to collect the partial results to decide whether the global sentence belongs to L or not. However, since the substrings belonging to L_1 can be arbitrarily long, any random or fixed policy to split the input into substrings is likely to be far from optimal; it may even be impossible to determine whether *ab* or *abb* groups are to be reduced in case a substring contains no 0s or 1s.

To cope with these issues, two straightforward approaches were pursued in the literature: speculative computations [77] or pre-scanning [72]. The former approach nondeterministically (speculatively) performs the parsing computation for all the possible cases: in our example, it carries on two parsing processes depending on whether the current substring is of type $1a^nb^n$ or $0a^nb^{2n}$, and discards the results of the incorrect computation as soon as possible. This approach, despite being effective, is not quite efficient: the computational effort of the parsing can be doubled or more, depending on the degree of nondeterminism. The latter approach involves a first lightweight scanning of the input to determine the viable splitting points: in the aforementioned example it would look for occurrences of 0 and 1 and split the input right before them. However, this approach introduces an overhead for the preliminary scanning which could require a pre-scanning over the whole input string. Summing up, the first approach may require a computational overhead to cope with the nondeterminism which is potentially more significant than the benefits provided by the parallelism, while the latter implies an $O(n)$ worst-case preprocessing which, in case of simple languages, may take as much as a sequential parsing process.

A decisive point to overcome these impasses resides in the exploitation of the *local parsing* properties of OPGs, which we leverage on to construct a *non-speculative* parallel parser. Intuitively, a language is locally parsable if, by inspecting a substring of bounded length, an (e.g., bottom-up) algorithm can deterministically decide whether the substring contains the right-hand-side of a production and can unequivocally replace it with the corresponding left-hand side. Local parsability is the key property that enables data-parallel parsing of isolated parts of the input so that their partial results can be recombined in a global syntax tree without backtracking: in other words, all the isolated partial syntax trees of a valid text are *final*.

In the following, we will report a systematic approach to exploit the local parsability of OPGs, describing a publicly available parser generator tool, and we will provide the results of an experimental campaign highlighting the speed-ups achievable w.r.t. popular sequential parsers, such as those generated by GNU Bison [3]. We chose as practical test-benches for our approach the JavaScript Object Notation (JSON) data description language, which offers a real-world validation for large input files, and the Lua programming language, to gauge how a language far richer than a data description language performs with the parallel lexing and parsing. Furthermore, we show that the minor theoretical limitations in terms of generative power of OPGs do not significantly affect the applicability of the approach. The changes needed to adapt the original BNF of the source language to OPG constraints are obtained in an original way by augmenting, and parallelizing as well, the initial phase of lexical analysis (or scanning): the lexer can be used to generate a stream of tokens where new tokens are emitted or modified to compose a string that can be analyzed by an OP parser. These transformations allow for naturally and effectively dealing with practical languages whose grammar is not readily expressible in OP form.

The chapter is organized as follows: Section 5.2 recalls the definition of local parsability property of OPGs and reports the parallel parsing algorithm. Section 5.3 provides our methodology for parallel lexical analysis, while Section 5.4 describes how we adapted the JSON and Lua languages to OP-based parsing. Section 5.5 describes the architecture of our parser generator, and Section 5.6 presents the results of the benchmark campaign. Section 5.7 concludes by comparing our approach to previous research on parallel parsers and lexers.

5.2 PARALLELIZATION OF SYNTACTIC ANALYSIS

This section presents a theory supporting the parallel parsing of OPLs which has been developed in [14]. After outlining the classical sequential OP parsing algorithm, we report the proof provided in this work that OP grammars and languages enjoy the local parsability property, which is the key to make parsing parallel. Subsequently, we describe the sequential parsing algorithm for OPGs presented in [14], which is generalized to apply both to terminal substrings and to partially processed ones (sentential forms). In this way it provides the basis for a parallel procedure consisting of two (or more) passes: at first the source string is split into chunks and each one is assigned to a separate worker, then the outputs are recombined and either a sequential or more parallel parsing passes are applied to them, computing the final parse tree.

5.2.1 *Basic operator precedence parsing algorithm*

OPGs support a very efficient bottom-up sequential parsing algorithm (a detailed description of sequential OP parsing is presented, e.g., in [54]). In traditional bottom-up parsing the input string is reduced, through a series of sentential forms, to the axiom of the grammar. At each step, the parser identifies in the current sentential form a segment that equals the r.h.s. of a rule and reduces it to the nonterminal of the l.h.s. The crucial issue in this process consists in finding, in an efficient way, the leftmost segment that can be reduced using a rule, so that it will create a node that is guaranteed to be part of the parse tree.

OP parsing relies on precedence relations to decide if a substring that matches the r.h.s. of a rule can be validly reduced to its l.h.s., and this test is very efficient. Intuitively, precedence relations control parsing for OPGs as follows.

- The source string s is enclosed between a pair of *end-marks* $\# \notin \Sigma$. Recall that by convention, # yields precedence to every terminal character and every terminal character takes precedence over #; furthermore, $\# \doteq \#$.
- Consider a rule $A \to \beta$, whose r.h.s. β occurs in a sentential form and is going to be reduced to A. Then β is "enclosed" between the pair $\lessdot$, $\gtrdot$, and relation $\doteq$ holds between every two consecutive terminal symbols of β. More formally, if there exists a derivation $S_1 \overset{*}{\Rightarrow} \alpha A \gamma \Rightarrow \alpha\beta\gamma$ $(S_1 \in S)$ then it must hold that $\alpha = \alpha' a$, $\gamma = b\gamma'$, $\beta = N_1 c_1 N_2 c_2 \ldots c_{n-1} N_n$, with $N_i \in N \cup \{\varepsilon\}$, $c_i \doteq c_{i+1}$, $1 \le i < n$, $a \lessdot c_1$, $c_{n-1} \gtrdot b$. Note that, as a particular case, a and/or b may be #: hence, given the OP relations between the end-marks and the other alphabet symbols, every r.h.s.s (including those at the border in a sentential form) will be enclosed within a pair $\lessdot$, $\gtrdot$. Observe that nonterminals are "transparent" in OP

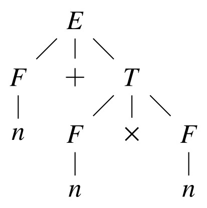


Figure 34.: A sample syntax tree for the OPG of arithmetic expressions of Figure 5.

parsing, i.e., they are not considered when evaluating the precedence relations between *consecutive*[11] terminals.

The basic idea of the algorithm is then straightforward. The parser uses a pushdown stack to identify the substrings to reduce to build the parse tree: the symbols of the input string are shifted onto the stack as long as they are read, and the stack keeps track of the precedence relation between consecutive symbols. Whenever a $\gtrdot$ relation holds, the parser identifies a pair of relations $\lessdot, \gtrdot$ on the stack, with a possible series of $\doteq$ precedence relations within them, and the substring enclosed between the pair corresponds to the segment to reduce. Hence, the stack holds only $\lessdot, \doteq$ markers and terminals, plus a $\gtrdot$ on the top each time a substring to be reduced is found.
Figure 34 reports the derivation tree of the string $n + n \times n$ generated by the OPG in FNF of the language of arithmetic expressions of Figure 5. Note that the second occurrence of terminal n is enclosed by the relations $+ \lessdot n$ and $n \gtrdot \times$ and can be reduced unequivocally to nonterminal F, thanks to the fact that G has no repeated r.h.s. Similarly the r.h.s. $F \times F$, in the context $\langle +, \# \rangle$ with $+ \lessdot \times$ and $n \gtrdot \#$, is deterministically reduced to nonterminal T.

Although the sequential OP parsing algorithm is quite efficient, a key improvement in performance can be obtained by tailoring the algorithm to exploit modern parallel architectures. The crucial property of OPGs that allows for the parallelization of syntactic analysis is the property of local parsability, presented next.

11 Recall that in an OF string α two terminals are *consecutive* if they are at positions $\alpha[j], \alpha[j+1]$; or at positions $\alpha[j], \alpha[j+2]$ and $\alpha[j+1] \in N$.

5.2.2 *Local parsability property and its exploitation for parallel parsing*

Intuitively, a language L generated by a CF grammar G is locally parsable if, for every sentential form, the r.h.s. of a production to be reduced can be uniquely determined through inspecting only a bounded context of the r.h.s. For instance, the language $L = \{a^n 0\, b^n\} \cup \{a^n 1\, b^{2n}\}$ generated by the grammar $S \to A \mid B$; $A \to aAb \mid a0b$; $B \to aBbb \mid a1bb$, is locally parsable because the "separators" $\{0, 1\}$ allow to decide the r.h.s. to be reduced. By contrast, $L = \{0\, a^n b^n\} \cup \{1\, a^n b^{2n}\}$, though being deterministic and generated by the LR grammar $S \to 0A \mid 1B; A \to aAb \mid ab; B \to aBbb \mid abb$, is not locally parsable since there is no way to decide whether to reduce a substring ab to A or abb to B without inspecting the first character of the string, which may be arbitrarily far away. The concept of local parsability has been formalized in the literature in similar ways; we adopt the definition of bounded-context CF grammar [48].

Definition 5.1. Let G be a CF grammar and $h \geq 1$. G is a *locally parsable* (or *bounded context*) grammar *with bound* h, iff for every rule $A \to \alpha$ of G, whenever

$$\#^h S \#^h \overset{*}{\Rightarrow} \zeta = \beta\gamma A\delta\eta \Rightarrow \beta\gamma\alpha\delta\eta \overset{*}{\Rightarrow} x \tag{11}$$

with $|\gamma| = |\delta| = h$, any other derivation $\#^h S \#^h \overset{*}{\Rightarrow} \vartheta\gamma\alpha\delta\phi$ where $h, \gamma, \alpha, \delta$ are the same as before, can be obtained exclusively by using the same rule $A \to \alpha$ to obtain α.

Thus, h specifies the length of the left and right neighborhood, i.e., the surrounding *context*, needed to make sure that string α must be reduced to nonterminal A. Floyd proved that, besides being decidable for any given value of h, the local parsability property implies that G is deterministically parsable (therefore also unambiguous).

An issue with OP parsing regards the possible presence of rules in the OPG having the same r.h.s.: when the substring corresponding to the r.h.s. of these productions is to be reduced, it is not possible to unequivocally determine which l.h.s. should replace the segment in a sentential form. In principle, the parser could keep all the alternative choices open until the uncertainty can be resolved; in the sequel, however, we shall rather avoid this form of ambiguity by considering OPGs in Fischer normal form (FNF) (hence, the grammars are invertible). Working with an OPG in FNF will allow us to simplify the parsing algorithm, without impairing generality nor the efficiency of the tools that implement it.

Theorem 5.1. *Every OPG in FNF is locally parsable with bound 1.*

Proof. Consider the step $\beta\gamma A\delta\eta \Rightarrow \beta\gamma\alpha\delta\eta$ of derivation (11), with $|\gamma| = |\delta| = 1$. Then, necessarily $\gamma, \delta \in \Sigma \cup \{\#\}$, $\gamma \lessdot$ the first terminal of α, and the last terminal of α $\gtrdot \delta$; only $\doteq$ occur within α. Then, whenever a string $\gamma\alpha\delta$ occurs in a sentential form of G, the same precedence relations hold between its terminals since G's OPM is conflict-free; thus α is the r.h.s. of some rule and, since G is in FNF, no other rule can produce α in a sentential form within the context (γ, δ). □

Although there exist contrived examples of locally parsable languages with bound 1 that cannot be generated by an OPG, such as the above language $\{a^n b a^n \mid n \geq 1\}$, they are of no practical relevance when taking into account real world programming language. The following corollary establishes the basis for parallelizing the standard OP parsing algorithm.

Corollary 5.1 ([14]). *For every substring $a\,\delta\,b$ of a sentential form $\gamma\,a\,\delta\,b\,\eta$, 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 $\gtrdot$.*

Proof. Consider the substring $a\delta b$: any r.h.s. contained therein is preceded by $\lessdot$ and followed by $\gtrdot$, and we reduce it to its l.h.s. Then we iterate the procedure until no pair $\lessdot \ldots \gtrdot$ (with possibly $\doteq$ in between) exist. At this point, necessarily, the condition of the corollary has been reached. □

THE PARSING ALGORITHM To allow its use in parallel parsing, [14] generalizes the traditional OPG parsing algorithm in order to analyze strings that may include nonterminals: such strings must begin and end with terminals or with #, and are in OF. This generalization is needed in the parallel setting in order to parse internal text segments,

Algorithm 1 : Generalized-OP-parsing$(\alpha, \text{head}, \text{end}, \mathcal{S})$ [14]

```
1. Let $X = \alpha[\text{head}]$ and consider the precedence relation between the top-most terminal $Y$ found in $\mathcal{S}$ and $X$.
2. If $Y \lessdot X$, push $(X, \lessdot)$; head := head + 1.
3. If $Y \doteq X$, push $(X, \doteq)$; head := head + 1.
4. If $X \in N$, push $(X, \perp)$; head := head + 1.
5. If $Y \gtrdot X$, consider $\mathcal{S}$:
    a) If $\mathcal{S}$ does not contain any $\lessdot$ then push $(X, \gtrdot)$; head := head + 1.
    b) Else, let $\mathcal{S}$ be $(X_0, p_0)(X_1, p_1)\ldots(X_{i-1}, p_{i-1})(X_i, \lessdot)\ldots(X_n, p_n)$ where $\forall j, i < j \leq n, p_j \neq \lessdot$.
        i. if $X_{i-1} \in N$ (hence $p_{i-1} = \perp$), and there exist a rule $A \to X_{i-1}X_i \ldots X_n$ replace $(X_{i-1}, p_{i-1})(X_i, \lessdot)\ldots(X_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$;
        ii. if $X_{i-1} \in \Sigma \cup \{\#\}$, and $\exists A$: $A \to X_i \ldots X_n \in R$,
            replace $(X_i, \lessdot)\ldots(X_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$;
        iii. otherwise start an error recovery procedure.
6. If (head < end) or (head = end and $\mathcal{S} \neq (a, \perp)(B, \perp)$), for any $B \in V_N$, repeat from step (1);
   else return $\mathcal{S}$.
```

and is reported in Algorithm 1. Algorithm 1 uses a stack $\mathcal{S}$ containing symbols that are pairs of type (X, p), where $X \in \Sigma \cup N$ and p is one of the precedence symbols $\{\lessdot, \doteq, \gtrdot\}$ or is undefined, denoted by $\perp$. The second component encodes the precedence relation found between two consecutive terminals – thus, it is always $p = \perp$ if X is nonterminal. The projection on the first component is denoted by $(X, p)|_1 = X$, and will be used to drop the precedence symbols when not needed. As a convention, the stack grows rightwards. Also, define a *handle* as a candidate r.h.s., i.e. a portion of a string in OF included within a pair $\lessdot$, $\gtrdot$ and with $\doteq$ between consecutive terminals.

The algorithm takes as input the string $\alpha = \alpha[1]\alpha[2]\ldots\alpha[m]$, $m \geq 3$ to be parsed: α is in OF and has the form $\alpha \in (\Sigma \cup \{\#\})V^*(\Sigma \cup \{\#\})$. It also receives as input parameters two pointers, head and end, to elements of α pointing to the second and last element of α, respectively. The last parameter taken is the parsing stack $\mathcal{S}$, initialized with $\alpha[1]$ on top of it.

Remarks

- Initially the algorithm will be applied to a terminal input string that is a substring of the input text. Therefore it will be $\alpha = asb$, with $a, b \in (\Sigma \cup \{\#\})$, $s \in \Sigma^*$ and the stack $\mathcal{S} = (a, \perp)$, consequentially in this case the condition in step (4) will not be met.

- Note that Algorithm 1 behaves as a traditional sequential OP parser when $\alpha = \#s\#$ and $\mathcal{S} = (\#, \perp)$. In this case the input is accepted if, and only if, the algorithm halts having read the whole input and $\mathcal{S} = (\#, \perp)(S, \perp)$.

- If the initial stack – disregarding precedence symbols – $\mathcal{S}|_1$ is irreducible (more precisely it has the form $a\alpha b$ of Corollary 5.1), then the same property will hold for $\mathcal{S}$ upon algorithm termination (unless an error occurs); in other words the property of Corollary 5.1 is an invariant w.r.t. the algorithm execution. As a particular case, this is true when the initial stack is the singleton $(a, \perp)$. In fact, the creation of a reduction handle at runtime can only happen if a $\gtrdot$ is going to be pushed on the stack when a $\lessdot$ is already in it. However, Algorithm 1 is designed to perform a reduction in such cases thus eliminating the possibility of an non-reduced handle sitting on the stack.

- An error is detected either whenever a handle does not match any of the valid r.h.s., or if no precedence relation holds between two consecutive terminals. In these cases then an appropriate error recovery strategy can be started.

We will now describe the parallel parsing strategy mapping the computation on multiple workers and recombining their results to obtain the full parse of the input. The term *worker* is used to denote the independent unit of processing in an abstract way from the chosen architecture. In principle it could even be a virtual process mapped into a mono-processor architecture, though in this case obviously there would be no benefit in terms of speed-up.

Let k be the number of available workers: the input source string is split into k substrings in an arbitrary fashion. Note that, despite there is no functional constraint concerning the splitting of the input token stream, this freedom is not incompatible with profitable heuristics for choosing the substrings, e.g., based on suitable pre-processing during lexical analysis.

Algorithm 1 is applied to each substring, obtaining a partial parse, which (thanks to the local parsability property) is a correct portion of the complete parse tree. Since OP parsing needs a look-ahead/look-back of one character to evaluate the precedence relations between consecutive terminals, when the source string is split, a 1-character overlap is left between consecutive substrings.

As an example of execution of this procedure, consider the grammar that generates the language of arithmetic expressions of Figure 5, assume that $k = 3$, and segment the source text: $\# \, n + n + n \times n \times n + \llparenthesis n \times n \rrparenthesis + n \, \#$ into:

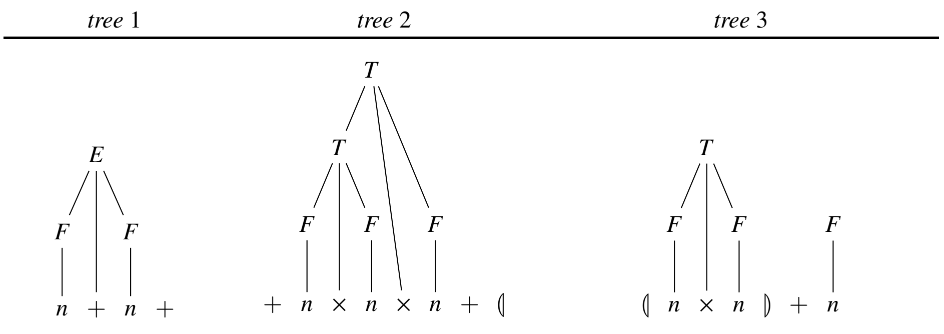
$$\# \overbrace{n + n}^{1} + \overbrace{n \times n \times n +}^{2} n \overbrace{\times n + n}^{3} \#$$

where the unmarked symbols $+$ and n are shared by the adjacent segments, and are used for look-ahead and look-back. After each parser has processed its segment, the partial trees and the stacks are shown in Figure 35.

Thanks to Corollary 5.1, after a sequential step the stack contents $\mathcal{S}$ of each worker can be split into two parts $\mathcal{S}^L$ and $\mathcal{S}^R$, such that $\mathcal{S}^L$ does not contain $\lessdot$ relations, and $\mathcal{S}^R$ does not contain $\gtrdot$ relations (in case of several $\doteq$ between the last $\gtrdot$ and the first $\lessdot$, the separation between the two parts is arbitrary). Notice either one of $\mathcal{S}^L$ or $\mathcal{S}^R$ may be empty. In our example, $\mathcal{S}_1^L$ and $\mathcal{S}_3^R$ are empty, while the workers produce the stacks:

$(\#, \perp) \quad (E, \perp) \quad (+, \lessdot)$	$\mathcal{S}_1 = \mathcal{S}_1^R$
$\overbrace{(+, \perp)(T, \perp)(+, \gtrdot)}^{\mathcal{S}_1^L} \quad \overbrace{(\llparenthesis, \lessdot)}^{\mathcal{S}_2^R}$	$\mathcal{S}_2$
$(\llparenthesis, \perp) \quad (T, \perp) \quad (\rrparenthesis, \doteq) \quad (+, \gtrdot) \quad (F, \perp) \quad (\#, \gtrdot)$	$\mathcal{S}_3 = \mathcal{S}_3^L$

To prepare the input for the next pass, one could simply concatenate the outputs, i.e., the stack contents delivered by the workers of the first pass, erase their precedence



$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_3$
$(\#,\bot)(E,\bot)(+,\lessdot)$	$(+,\bot)(T,\bot)(+,\gtrdot)(\llparenthesis,\lessdot)$	$(\llparenthesis,\bot)(T,\bot)(\rrparenthesis,\doteq)(+,\gtrdot)(F,\bot)(\#,\gtrdot)$

Figure 35.: Partial trees and corresponding stacks after the first parallel pass on text $n+n+n\times n\times n+\llparenthesis n\times n\rrparenthesis+n$

components by applying the $|_1$ projection, iterating the same schema as in the first pass, i.e. splitting the obtained string again into $k' \leq k$ chunks to be assigned to k' workers. Instead, [14] proposes an heuristic approach aiming at maximizing the chance to produce a complete sub-tree or at least to include a fairly large one as soon as possible. Intuitively, this goal is achieved by pairing strings containing $\lessdot$ – at the left – with others containing $\gtrdot$ – at the right.

Figure 36 depicts the construction of the initial configuration of the stacks and inputs for pass two. Let $\mathcal{W}$ and $\mathcal{W}'$ be consecutive workers of the previous pass, and let their bipartite stacks be $\mathcal{S}^L\ \mathcal{S}^R$ and $\mathcal{S}'^L\ \mathcal{S}'^R$. Define the stack initialization function as $\mathcal{S}_{combine}(\mathcal{S}^L,\ \mathcal{S}^R) = (a,\bot)\ \mathcal{S}^R$ where a is the top symbol of $\mathcal{S}^L$. Note that the precedence value listed with a, becomes undefined since in the new stack a is not preceded by a terminal. The input string initialization function is defined as $\alpha_{combine}(\mathcal{S}'^L) := \alpha'$, where α' is the suffix of $\mathcal{S}'^L|_1$ without its first symbol (which is already on the top of $\mathcal{S}^R$).

Note that, in case $\mathcal{S}'^L$ (or, symmetrically, $\mathcal{S}^R$) is empty, $\mathcal{S}^R$ is simply concatenated with $\mathcal{S}'^R$ and the output of the following worker, say $\mathcal{S}''^L\ \mathcal{S}''^R$ is used to complete the construction of the new pair $(\mathcal{S},u)$. Another notable exception where the aforementioned stack composition strategy cannot be applied as-is is the $\mathcal{S}^L$ component of

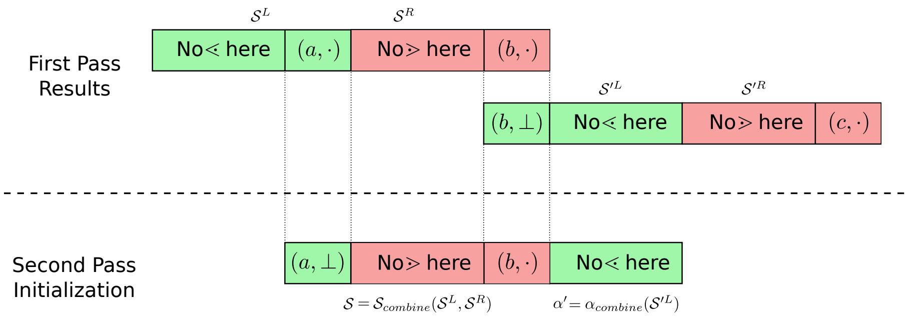


Figure 36.: Preparation of the initial stack and input string for the next parsing phase of a worker: the stack is $\mathcal{S}$ and the input string is α'.

the leftmost worker's output (symmetrically, the $\mathcal{S}^R$ component of the rightmost one), as it will always be empty. As a consequence, the initial stack of the new input for the leftmost worker will be $\mathcal{S}_1^R$ (concatenated with $\mathcal{S}_2^R$ if $\mathcal{S}_2^L$ is empty).

The complete parallel parsing schema is summarized by Algorithm 2, which provides a complete schema for parallel parsing a generic string β by means of k workers.

Remark 5.1. Algorithm 2 is a "core formulation" amenable to several variations and improvements. The most relevant one concerns the number of passes of parallel parsing: employing many workers when the whole input size is small enough to be assigned to a single worker may obviously incur in performance penalties due to the worker spawning overhead. For instance, whenever the source string exhibits a fairly balanced structure (e.g., many functions of comparable size) it is likely for the first pass to produce fairly short stacks suitable to be concatenated into a unique string. By contrast, the partition feeding the first pass may generate chunks which do not correspond to large sub-trees of the complete syntax tree. In this case, the construction of the new chunks by pairing $\mathcal{S}^R$ sides with $\mathcal{S}^L$ ones should increase – if not maximize – the number of handles belonging to the same chunk, and therefore the number of reductions performed by the second – and possible subsequent – passes. In practice, no more than two parallel passes are usually needed to produce a small enough input for the final pass.

Algorithm 2 : Parallel-parsing(β, k) [14]

```
1. Split the input string $\beta$ 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 $\mathcal{S} = (a, \perp)$, $\alpha = a\beta_i b$, head $= |\beta_1\beta_2 \dots \beta_{i-1}| + 1$, end $= |\beta_1\beta_2 \dots \beta_i| + 1$; $a$
   is the last symbol of $\beta_{i-1}$, and $b$ the first of $\beta_{i+1}$. Conventionally $\beta_0 = \beta_{k+1} = \#$.
   The result of this pass are $k' \le k$ pairs of stacks $\mathcal{S}_i^L \mathcal{S}_i^R$, as specified above.
3. Repeat:
   a) For each adjacent non-empty stack pair $\mathcal{S}_i^L \mathcal{S}_i^R$ and $\mathcal{S}_{i+1}^L \mathcal{S}_{i+1}^R$, launch an in-
      stance of Algorithm 1, with $\mathcal{S} = \mathcal{S}_{combine}(\mathcal{S}_i^L, \mathcal{S}_i^R)$, $\alpha = \alpha_{combine}(\mathcal{S}_{i+1}^L)$,
      head $= 1$, end $= |\alpha|$.
   b) Until either we have a single reduced stack $\mathcal{S}'$ or the computation is
      aborted and some error recovery action is taken.
4. Return $\mathcal{S}'$.
```

Algorithm Complexity

In terms of asymptotic complexity, the requirements for a positive evaluation of the whole approach are: a best-case linear speedup w.r.tthe number of processors and a worst-case complexity not exceeding the one of a fully sequential parsing.

By inspecting Algorithm 1 and Algorithm 2, it is clear that the total number of elementary operations (shifts and reductions) is $O(n)$ since no reduction is performed more than once exactly like the sequential case. Indeed, some terminal symbol could be shifted more than once during the various passes, but this occurs only for the few of them which have not been reduced by the previous passes, and for a number of times that does not exceed the number of passes.

To achieve a worst case parsing time not exceeding the sequential parsing, it is essential that the combination of stacks $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$, inside step (3)(a) of Algorithm 2, takes $O(1)$ time (hence overall $O(k)$ for k workers). A possible technique to achieve this goal consists in storing, during the execution of Algorithm 2, a marker that keeps track of the separation between $\mathcal{S}^L$ and $\mathcal{S}^R$. Such a marker can be initialized at the position where the first $\lessdot$ sign is detected and then updated every time a reduction is applied that removes the sign and a new element is shifted on the stack as a consequence of a new $\lessdot$ relation.

For instance, in the case of $\mathcal{S}_2$ in Figure 35, the marker is initialized at the position of the first $+$ symbol and remains there after the five reductions $F \Rightarrow n$, $F \Rightarrow n$, $T \Rightarrow F \times F$, $F \Rightarrow n$, $T \Rightarrow T \times F$ since $+ \lessdot n$ and $+ \lessdot \times$. When the second $+$ (the third of the whole string) is shifted (without removing the previous one as the $\gtrdot$ between the two $+$ is not matched by a corresponding $\lessdot$ at its left), it is moved to the position of the second $+$ as $+ \lessdot ($, where it marks the beginning of $\mathcal{S}_2^R$.

These operations require $O(1)$ time regardless of the stacks being implemented by means of arrays or by means of more flexible linked lists; thus, they do not affect the overall $O(n)$ complexity of the whole algorithm.

It is then clear that the ideal linear speed-up w.r.t. the number of processors will be most representative of the actual one whenever most of the parsing is done during the first pass. By contrast, the worst case occurs when either only $\lessdot$ relations or $\gtrdot$ relations are present in the whole input; this is the case of regular languages respectively generated, e.g., by a left-linear or by a right-linear grammar. In such cases, only one worker (respectively the leftmost or rightmost one) performs useful parsing whereas the others leave their input unaffected. The second pass would produce a unique string – of length $(k-1)/k \cdot n$ – which would be parsed sequentially.

5.3 PARALLELIZATION OF LEXICAL ANALYSIS

Lexical analysis takes place before parsing and translation, and it is a common belief that it is a fairly easy and less time consuming job compared with the following phases. While this may be true in other settings, we report that lexical and syntax analysis for operator precedence languages often require comparable effort. Thus the gain in performing parallel parsing alone would be small without coupling it with parallel lexical analysis and preprocessing. Furthermore, apart from a few idiosyncrasies of some languages – which tend to go unused by programmers – lexical analysis is even better suited for parallel execution. However, to achieve this goal a few non-trivial technical difficulties must be tackled.

In this section we present a fairly general schema for parallelizing lexical analysis, which can be applied to most programming languages. A distinguishing feature of our lexical analysis is that it produces a stream of tokens which, rather than being compatible with the original BNF of the source language, is ready to be parsed according to an "OP version" of the official grammar, thus yielding an advantage from both a performance, and an adaptation to OP parsing point of view. Typically such a preprocessing allows for disambiguating some terminals which are overloaded in the language and would induce conflicting relations in the operator precedence matrix: depending on the context in which they occur, the lexer can associate them to distinct token classes, so that the resulting string of tokens can be parsed according to the OP version of the syntactic grammar. Note that such a disambiguation is complementary to the usual operations performed in the lexing phase to cope with the presence of ambiguities in the lexical grammar (as, e.g., in the presence of reserved words and identifiers corresponding to the same lexical pattern). However, this is hidden from the user who does not have to worry about the internal format.

We now provide the definitions required to describe lexical analysis parallelization. The lexicon of the language is described by a lexical grammar, which assumes as terminal alphabet the characters present in the input stream. Often, a lexical syntax can be analyzed by means of a finite state machine (FSM) as opposed to a pushdown one, which is reserved for parsing. In this work we will tackle both a language where this assumption holds (JSON) and one where it does not (Lua) in Section 5.4, and will describe the issues in generalizing the approach in Section 7.1.

We adopt the following conventions to distinguish terminal and nonterminal symbols of syntactic and lexical grammars: terminals of syntactic grammars and nonterminals of lexical grammars are in boldface font: $\boldsymbol{n}, \boldsymbol{s} \ldots$, nonterminals of syntactic grammars are denoted by capital letters: $A, B \ldots$, and terminals of lexical grammars are in monospace font: $\texttt{if}, \texttt{+} \ldots$. We also introduce some basic terminology (as in, e.g., [6]) which tailors some general terms to the scope of this section.

Definition 5.2.

- A *lexeme* is a sequence of characters corresponding to a valid sentence of the lexicon grammar (e.g., a built-in identifier, a reserved keyword, an operator).

Its form depends on the lexical part of the language definition, and it can be typically recognized by a FSM.

- A *string* is a lexeme built as a sequence of characters enclosed within a pair of delimiters, typically either single or double quotes. It cannot contain any other delimiters of the same kind without proper escaping (e.g. prefixing them with a \ character). Strings may contain control characters (e.g. newlines).

- A *token* is a pair ⟨*token-name*, *semantic value*⟩ resulting from the analysis of a sequence of characters matching the form of a valid lexeme (*token-name* denotes a nonterminal of the lexical grammar, and there is a finite number thereof). Sample token instances are ⟨**LPAREN**, `(`⟩, or ⟨**STRING**, `"yesterday I ate an icecream"`⟩. Since the focus of this work is the lexical and syntactic analysis of a text, from now on we will identify the token with the first element of the pair.

- A *comment* is a sequence of characters delimited by special symbols according to language dependent rules, and does not correspond to a lexeme. A comment should be matched and discarded during the lexing process. Many languages use different markers for single-line and multi-line comments.

The goal of lexical analysis (*lexer*) is to recognize the lexemes in the source character stream and generate a sequence of tokens, removing the comments. The lexical grammar, in spite of the fact that it typically defines a regular language, may be not locally parsable in its immediate form and in most cases is ambiguous. Yet, lexical analysis can be made suitable for parallel execution and, given the typical "flat structure" of programming language lexicon, is a more natural candidate for efficient parallelization than parsing, which has to deal with the nesting of syntactic structures, as in fact it happened in practice (see the discussion in Section5.7). To achieve this goal, however, two issues must be addressed. First of all, splitting the source text randomly into chunks to be processed by parallel workers may split a lexeme across different segments. Thus, the results produced by lexers working on adjacent chunks will have to be reconciled to cope with this issue.

The second issue concerns the occurrence of very long comment sections, possibly longer than the chunk assigned to a single worker. It is commonplace among some programmers to comment portions of obsolete or temporary code, effectively preventing the lexer from knowing if it is analyzing a portion of a comment or not, in the case of lack of comment delimiters in its chunk. This is further exacerbated by the fact that some languages adopt some exotic syntactic rules for comment and/or string delimiters in general. For instance, in Lua strings can be delimited either by quotes or by opening and closing symbols of unbounded number of forms.

To cope with this problem, we accept a minimum amount of nondeterminism during the lexing phase only. We run several speculative computations for each worker, corresponding to the different states of the lexing machine which are legitimate on the splitting point of the input stream. If the worker is able to remove the ambiguity during the analysis of its chunk, the incorrect computation(s) are halted and only the correct one proceeds. We will show that the number of language dependent simultaneous computations, never exceeds 3 in our case.

Therefore, every worker will produce several candidate token lists for its chunk. Any disambiguation that cannot be performed during the single worker analysis is done when the partial token lists are joined into a single one. For instance, assume that a worker w reaches a given state s_k after analyzing completing the lexical analysis of its input chunk. If there is a computation starting with s_k among the ones performed by the worker acting on the input chunk after the one assigned to w, the corresponding candidate token list is merged with the output of w.

We now illustrate our approach to parallel lexical analysis in a similar way to the path followed in the case of parsing: we first revise normal sequential lexers to make them suitable to work on partial chunks and to produce partial outputs to be later integrated; then we will show how a parallel lexer splits the source code into chunks, assigns them to different workers, and reassembles their partial outputs. We will also make use of a running example to better illustrate the various steps of our algorithms.

Example 5.1. Consider the grammar generating arithmetic expressions presented in Figure 5, extended to allow also for operations on strings. The resulting OPG is depicted in Figure 37(b), whereas the lexical grammar is reported in Figure 37(a), with

n denoting an identifier and ***s*** a literal string; Figure 37(c) depicts a FSM recognizing tokens and comments compatible with the lexical grammar; to keep its appearance easily understandable we restricted its description to the recognition of these fundamental elements of the lexicon; however, its extension to cope with a whole chunk consisting of a sequence of such elements, possibly broken at the boundaries of the chunk, is conceptually straightforward.

The lexical grammar specifies two formats for strings: single-line ones are delimited by single-quotes `'  '`, multi-line ones delimited by a pair of triple-double-quotes `"""  """`. The lexicon also allows for introducing both single-line and multi-line comments with a C-like syntax: a multi-line comment is delimited by `/*` and `*/`, while a single-line comment begins with `//`.

In our running example we will refer to the sample code snippet in Figure 38.

If such a complete code snippet were supplied to the FSM of Figure 37(c), the output produced by the lexer would be (for completeness we include also the semantic components of the tokens):
$\langle \boldsymbol{n}, \texttt{var}\rangle \langle + \, \texttt{+}\rangle \langle \boldsymbol{n}, \texttt{x}\rangle \langle \times, \times\rangle \langle \boldsymbol{n}, \texttt{y}\rangle \langle +, \texttt{+}\rangle$ $\langle \boldsymbol{s},$ `a multi-line string interrupted here ×timesToConcat + ` $\rangle \langle +, \texttt{+}\rangle$ $\langle \boldsymbol{n}, \texttt{a}\rangle \langle +, \texttt{+}\rangle \langle \boldsymbol{n}, \texttt{z}\rangle$.

Our goal is to obtain the same result by splitting the job among several workers: in our example we will use three of them.

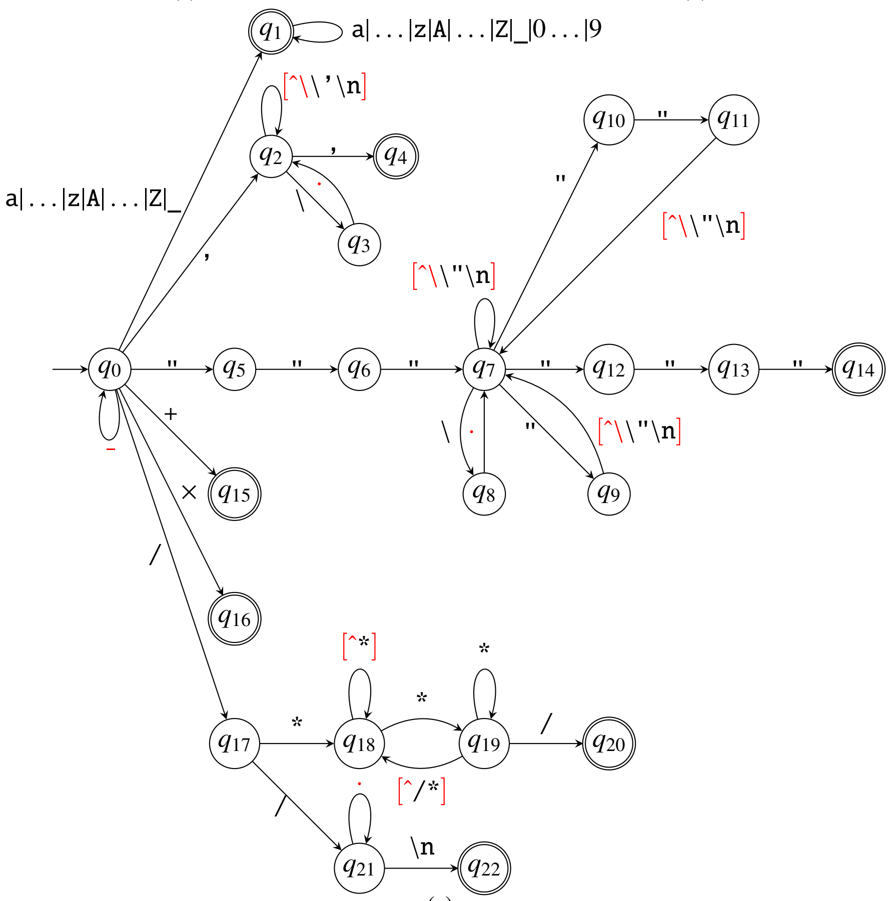
SOURCE CHARACTER STREAM PARTITIONING First, the input stream is split into segments of equal size; most likely, however, such a "blind" split may break a lexeme, typically an identifier which normally is not very long; thus, it is often sufficient to consider a look-ahead/look-back of a few characters to find a lexeme separator (i.e. a white-space): in such cases the splitting point may be conveniently set right after it and the boundaries of the chunks are updated trying to avoid splitting any token in the middle. For instance, in our example of Figure 38, the original ending points are denoted by | (red); by analyzing a bounded context in the neighborhood of the split points we identify the occurrence of two newline characters, and update the splitting points moving them to the positions denoted by | (blue). In general it cannot be known a priori how far is the end of lexeme from a given point; thus, the length of the search

```
input     →  element | input element
element   →  _ | token | comment
token     →  n | s | + | ×
n         →  letter (letter | digit)*
s         →  '(\\.|[^\\'\n])*' |
             """(char|"char|""char)*"""
+         →  +
×         →  ×
letter    →  [a–zA–Z_]
digit     →  [0–9]
char      →  \\.|[^\\"]
comment   →  /* ([^*] | *+[^/*])**+/ |
             // .*\n
```

(a)

$$
\begin{aligned}
S &\rightarrow A\\
S &\rightarrow B\\
A &\rightarrow A + B\\
A &\rightarrow B + B\\
B &\rightarrow B \times \boldsymbol{n}\\
B &\rightarrow B \times \boldsymbol{s}\\
B &\rightarrow \boldsymbol{n}\\
B &\rightarrow \boldsymbol{s}
\end{aligned}
$$

(b)



(c)

Figure 37.: Lexical (a) and syntactic (b) grammars of arithmetic expressions without parentheses, extended to deal also with strings, and a FSM recognizing tokens and comments of the lexical grammar (c). In figures (a) and (c), metasymbols used in regular expressions are in red while the terminals of the lexical grammar in black; symbol _ stands for any control character (whitespace, newline, etc.).

```
var + x /* This is a multi-line comment that
contains part of an arithmetic expressi|on:|
var + ’string*/×/*’ //and this is an embedded single-line comment
+ ’6’ × */ y +| """a  multi-line string interrupted here|
× timesToConcat + """+a//-(5×b)""" × id
+ z
```

Figure 38.: Sample string generated by the grammars of arithmetic expressions in Figure 37.

should be stated on the basis of some, possibly language dependent, heuristic criteria and should not exceed a few characters in any case.

Assuming that such a separator is found, the possible ambiguity of the starting state of the lexer reduces either to the beginning of a lexeme, a position within a string, or to a position within a comment. If the search of a separator within a bounded context of the start of the segment is not conclusive, the initial state can also correspond to an internal point of a non-string lexeme. In the worst case, i.e., when a non-string lexeme is cut by the splitting, the worker will need to carry on 4 computations at once. However, we report that no such cases took place in our experimental evaluation.

In our example the worker assigned to the first chunk is deterministic and begins its analysis in the initial state of the FSM of Figure 37(c); The second and third workers, instead must carry over three simultaneous computations each, starting at the beginning of a token, or inside a multi-line string or inside a comment; thus, the corresponding starting state of the FSM are, respectively, q_0 or q_7 or q_{18} (or q_{19}).

PARALLEL LEXICAL ANALYSIS Once assigned to a given chunk, each worker carries on a computation for each possible alternative initial state of the lexing machine. Most likely, during this phase, some disambiguation among the open alternatives in the lexical analysis may occur: this happens a) if a worker meets a character forbidden in some of the active states, thus aborting the erroneous computations; b) if a sequence of characters causes the recognition of a lexeme on more than one computation, thus collapsing them into a single one. Although this collapse is quite uncommon, we will provide an example thereof when detailing the parallel lexing strategy for the Lua language.

Below we give an abstract version of the algorithm executed by each worker on the assigned chunk.

- Let M be the FSM recognizing the language generated by the lexical grammar; let Σ denote its alphabet, Q its set of states, $\Delta : Q \times \Sigma \rightarrow Q$ its transition function, q_0 its initial state and F its set of accepting states. We assume, as usual, that when an accepting state is reached a lexeme in the input string is recognized and its semantic value is output. The transition function Δ can be partial: we use symbol $\perp$ to denote an undefined value thereof. Δ^* denotes the reflexive and transitive closure of Δ.

- Let N be the number of concurrent computations of M and let s be the N-tuple of states currently reached along each of the N computations of M: the undefined state value $\perp$ is used to denote that the computation has been interrupted because of an error in the input string, while a value i $(1 \leq i \leq N)$ is used to denote that the computation has been merged with computation i.

- Let b be a N-tuple of strings where b^j, for $1 \leq j \leq N$, contains the partial semantic value of the token currently under recognition along the $j-th$ computation of M.

- Let T be a N-tuple of lists of tokens.

Example 5.1 (continued). *Going back to our running example, the worker scanning the first chunk performs only one computation, whose initial state is the initial state the FSM of Figure 37(c). The worker returns the tokens* $\mathbf{n} + \boldsymbol{n}$ *and finishes in a state specifying the presence of a non terminated comment. The choice of a newline separator as the ending character of the chunks excludes the possibility that their initial portion belongs to a single-line string.*

Algorithm 3 : Sequential-lexing(γ, s)

```
1. Initialization: head := 1; end := $|\gamma|$; $X = \gamma[\text{head}]$.
   The strings in $b$ and the lists in $T$ are set to empty and $s$ is the $N$-tuple of initial states.
2. For each state $s^j$ $(1 \leq j \leq N)$ such that $s^j \in Q$:
   a) If $s^j = s^k$ for a $k$ such that $1 \leq k < j$ then merge computations $j$ and $k$,
      by setting $s^j := k$ and joining the end of list $T^j$ to the end of list $T^k$.
   b) Else, let $q := \Delta(s^j, X)$.
      i. If $q \neq \perp$
         A. If, while in state $s^j$, $M$ is not reading a whitespace character
            outside a string or a comment, set $b^j := b^j X$.
         B. $s^j := q$.
      ii. Else
         A. If $s^j \in F$ and $b^j \neq \varepsilon$ then append $b^j$ to $T^j$ and set $s^j := \Delta(q_0, X)$
            and $b^j := X$.
         B. Else $s^j := \perp$.
3. $head := head + 1$.
4. If $head \leq end$ repeat from step 2, else return the tuple $s$ (where each state
   $s^j = i$ $(1 \leq i \leq N$ is updated to $s^j := s^i)$ and the tuple $T$ of lists of tokens.
```

The worker returns the tokens $\boldsymbol{n} + \boldsymbol{n}$ *and finishes in a state specifying the presence of a non terminated comment. The choice of a newline separator as the ending character of the chunks excludes the possibility that their initial portion belongs to a single-line string.*

The second and third chunk are scanned along three simultaneous computations starting at the beginning of a token or inside a multi-line string or inside a comment, respectively. In the case of the second worker, the first of the three computations generates a list of tokens $\boldsymbol{n} + \boldsymbol{s} + \boldsymbol{s} \times$ *before being aborted due to the occurrence of the unexpected end of a comment */. Along the second computation a multi-line string is recognized, and the computation is aborted after reading the word* `multi-line`.

*The third computation detects the end of a comment */ and produces a list of tokens* $\times\, \boldsymbol{n} +$. *The run ends in a state denoting the presence of a non-terminated string, while the prefix which has been read is stored to allow for a possible concatenation with the suffix in the following chunk.*

Finally, the third worker carries on three simultaneous computations: the first and second one end in a final state, generating as a list of tokens respectively $\times\, \boldsymbol{n} + \boldsymbol{s} \times \boldsymbol{n} + \boldsymbol{n}$ *and* $\boldsymbol{s} + \boldsymbol{n} + \boldsymbol{n}$. *The third one does not collect any token and ends in a state which signals the presence of a non-terminated comment.*

LIST JOINING PHASE Once the various workers have processed their chunks in parallel their partial outputs must be integrated into a unique sequence of tokens to be supplied (after further partitioning) to the parallel parsing phase. This job can be done in a similar way to the case of parsing, with two important differences:

- Whereas each worker in the parsing phase delivers just one output, in general parallel lexers will produce several candidate outputs among which the integration phase will choose the right one.
- Whereas after a first parsing pass further parsing is applied, possibly in more than one pass, the partial outputs of the parallel lexical analysis need only to be selected and integrated without further analysis.

The integration of the partial outputs is carried over sequentially. Since the leftmost worker W_1 performs a deterministic computation, its final state is the correct initial one for the following worker W_2. The correct list among the ones produced by its right neighbor is thus selected by matching the final state of the first worker against one of the members of s^2. In case no match occurs, an error is signaled and error recovery strategies are enacted. The output of the following workers is handled similarly during the whole list joining phase, which has linear complexity in the number of workers h. If the lexical grammar of the language is not regular, some additional actions may be required. From our experience this case is quite uncommon in practical programming languages, and does not affect significantly the efficiency of the whole process; nevertheless we will deal with the noticeable exception of Lua.

Algorithm Parallel-lexing summarizes the coordination of the activities of the various sequential lexers.

```
Algorithm 4 : Parallel-lexing($(\delta, h)$)
  1. Split the input string $\delta$ into $h$ substrings of equal length.
  2. Scan a fixed length context of the ending points of the substrings to check
     whether any lexeme is broken. If a bounded look-ahead/look-back does not
     suffice to determine whether the boundaries of a substring are inside a lexeme,
     move the substring boundary so as to reduce as much as possible the ambiguity
     on the starting state for its analysis. (The bound for the look-ahead/look-back
     and the special character to seek in this search are heuristically chosen in a
     language-dependent way).
  3. Let $\delta_1\delta_2\dots\delta_h$ be the resulting substrings and, for each $1 \le i \le h$, let $s_i$ be a
     tuple of states of $M$, such that from each state $s_i^j$ $(1 \le j \le |s_i|)$ the scanning of
     substring $\delta_i$ can start.
  4. Launch $h$ instances of Algorithm 3 (sequential lexing), where, for each $1 \le$
     $i \le h$, the parameters are $\gamma = \delta_i$, $s = s_i$. The results of this pass are $h$ tu-
     ples $q_1, q_2 \dots q_h$ of states of $M$ such that, for each $1 \le i \le h, 1 \le j \le |q_i|$,
     $q_i^j = \Delta^*(s_i^j, \delta_i)$, and $h$ tuples $T_1, T_2 \dots T_h$ of lists of tokens built along the
     corresponding computations, where, for each $1 \le i \le h$, $|T_i| = |s_i|$.
  5. Build a unique list $T$ of tokens by choosing exactly one list from each tuple $T_i$
     $(1 \le i \le h)$ and concatenating them. The selection is performed sequentially,
     starting from the result of the instance that processed the leftmost substring, as
     its computation is unambiguous (in fact, $|s_1| = 1$).
  6. Return the list $T$ of tokens.
```

Thus, the overall complexity of the parallel lexing phase is $O(n/h) + O(h)$.

Example 5.1 (continued). *In the case of our running example, after each worker has completed the scanning of its character stream segment, the last step of the algorithm builds a single list of tokens from those generated for each chunk, eliminating the initial ambiguity on the start state of the lexical analysis. The partial list of the first worker is concatenated with the token list generated by the second worker along the computation starting from inside a comment. The resulting list is then concatenated* *with the one produced by the last worker along its second computation, updating also the semantic value of the string split across the second and third chunks.*

The complete list of tokens returned to be processed by the parsing workers, together with their semantic values provided for clarity, is the same as it would have been produced by a single sequential lexer, i.e., $\langle \boldsymbol{n}, \mathtt{var}\rangle$ $\langle +, \mathtt{+}\rangle$ $\langle \boldsymbol{n}, \mathtt{x}\rangle$ $\langle \times, \times\rangle$ $\langle \boldsymbol{n}, \mathtt{y}\rangle$ $\langle +, \mathtt{+}\rangle$ $\langle \boldsymbol{s}, \mathtt{a\ multi\text{-}line\ string\ interrupted\ here} \times \mathtt{timesToConcat\ +}\ \rangle$ $\langle +, \mathtt{+}\rangle$ $\langle \boldsymbol{n}, \mathtt{a}\rangle$ $\langle +, \mathtt{+}\rangle$ $\langle \boldsymbol{n}, \mathtt{z}\rangle$.

We now detail how we tailored the above general schema to JSON and Lua.

5.3.1 *The case of JSON*

JSON (JavaScript Object Notation) is a data description language, described in the Internet Engineering Task Force document RFC4627 [37], and based on a subset of JavaScript. JSON is widely employed in web applications, where it is progressively superseding XML as a more efficient and compact format for serializing and exchanging structured data.

JSON source code is not intended to be written or read by humans, but rather to be processed by machines. Consequently the JSON grammar lacks some of the typical lexical features of programming languages such as comments and does not mandate a code indentation style as some programming languages do, e.g. Python. Our purpose in selecting JSON as our first case study is to prove the practicality of parallel parsing, providing a realistic benchmark for speedups. In particular, with JSON representing a valid data description language alternative to XML in an ever increasing amount of scenarios, the average size of the JSON files to be processed is already sizeable and increasing. We now detail the steps of the algorithmic schema *Parallel-lexing*, tailoring it to the lexical features of JSON.

SOURCE CHARACTER STREAM PARTITIONING The lexemes in the input stream are separated by white-space characters, i.e. spaces, tabulator characters and newlines. Thus, we split the input stream on white-space characters, which occur reasonably frequently. This choice has a drawback since spaces and tabulators are also allowed within strings, so that a string could be split across two chunks. An alternative choice is to break chunks only in correspondence of newline characters, since JSON strings are constrained to be single-line. However, automatically generated JSON code may lack newline characters, making this alternative choice unpractical. Thus, we choose to use white-spaces as chunk separators and accept the consequent limited ambiguity of the lexing. Since the JSON grammar includes only strings as arbitrary length lexemes, the ambiguity for each chunk reduces to two possible initial states only: outside a token or within a single-line string.

PARALLEL LEXICAL ANALYSIS Each worker carries on at most two computations for a chunk, corresponding to the two possible starting states of the analysis. To reduce the level of ambiguity during the lexing action, we exploit the fact that the set of characters composing the non-string lexemes is a proper subset of the ones allowed to appear within a string. Thus if one of the characters which can only appear inside a string is met, the lexing action assuming to be outside a string lexeme can be stopped. To perform the token list recombination, each worker counts the number of string delimiters symbols (double quotes) occurring in the chunk.

LIST JOINING PHASE The last phase of the algorithm, which merges the token lists generated by the lexing workers solving the initial ambiguity, i.e., determining whether or not the chunk started within a string. This is done by checking the parity of the number of quotes read by all the workers preceding the one in need of disambiguation.

5.3.2 *The case of Lua*

Lua is a lightweight multi-paradigm programming language widely used as a domain specific language support engine, with a widespread use in video game development. Currently, Lua is the leading scripting language in this application area (as reported in [42] and in [49] which declares Lua the winner of the 2011 Game Developer Magazine Front Line Award). In particular, it has been adopted by prominent industrial game developers such as LucasArts (Grim Fandango, Escape from Monkey Island) and BioWare. Besides the video games programming application area, Lua has been used in various projects (such as Celestia [27]) and since February-March 2013 has been adopted as a template scripting language on Wikipedia [94].

Lua is a full fledged programming language and exhibits some of the syntactic "liberalities" that are fairly typical in various modern programming languages. We stress that the point of employing Lua as benchmark is to show that a richer grammar does not adversely impact on the performance gains obtainable through parsing it in parallel. In particular, the one of its key peculiarities is that the lexical grammar of Lua, unlike most programming languages, is not a regular one. This is due to a non regular syntax for strings and comments, which requires ad-hoc solutions when tailoring the schema for parallel lexical analysis.

In Lua, strings may be delimited by the so called *long brackets*, in addition to the usual single and double quotes. An opening long bracket is defined by the character pattern `[`$=^n$`[`, with the corresponding closing long bracket being `]`$=^n$`]`. The pair of long brackets must have the same number $n \geq 0$ of = characters to be recognized as a syntactically valid pair. Opening and closing long brackets can be nested, but a valid pair of long brackets cannot contain a closing long bracket of the same type.

Similarly, comments can be single-line or multi-line. Single-line comments start with a double hyphen (`--`) and extend until the end of the line. Multi-line comments have a syntax similar to strings, as they begin with an opening long bracket, preceded by a double hyphen, and end at the corresponding closing long bracket. Strings and comments can be arbitrarily nested, except that they cannot properly contain other comments or strings delimited by brackets with the same number of = symbols, lest an ambiguity on closing long brackets should arise.

Lua's complex syntax for multi-line strings and comments may lead to an intolerable source of ambiguity when different chunks of the input character stream are scanned in parallel. In particular, the arbitrary length marker defined as a delimiter for multi-line strings and comments results in an infinite number of possible delimiters for these constructs. Since there is no way to discern via a fixed look-ahead/look-back analysis which of these delimiters, if any, is enclosing the stream chunk to be analyzed by a lexer, the possible number of starting states and token lists generated along the corresponding computations on the segment could potentially be infinite. A lexing worker, thus, cannot carry on distinct runs for all the possible alternatives for the starting point of the analysis.

To deal with multi-line strings and comments in Lua, we introduce a few constraints on the source programs that can be processed by the schema. To bound the possible degrees of ambiguity in the lexical analysis, we forbid the non-regular syntax for string delimiters: we require that opening and closing long brackets that delimit multi-line strings has $n = 0$ characters = in the patterns `[=`^{n}`[` and `]=`^{n}`]`, i.e., we admit only the `[[` and `]]` as string delimiters. Instead, we do not restrict the syntax to specify comments, so that they can be delimited by long brackets with an arbitrary number of = signs between the brackets, retaining also the non regular constraint on the number of = characters. This choice is consistent with the common use for multi-line comments, as a container of legacy code, which in turn mandates the need to specify a different comment delimiter from the ones already in use in the enclosed portion of text. Furthermore, to limit the complexity deriving from the possible arbitrary nesting of strings and comments, we assume that multi-line comments are always ended by a newline, so that they cannot end inside a single-line string or comment.

Given these constraints on the lexical grammar of Lua, which we have verified to match widely employed programming practices, we now detail the steps in the algorithm *Parallel-lexing* left open in the general schema.

SOURCE CHARACTER STREAM PARTITIONING The partitioning of the source character stream for Lua has been operated by employing the newline characters as effective splitting points. This choice is justified by Lua being a programming language which is expected to be written and read by humans, and thus endowed with proper indentation. Given the hypotheses made on the source form, the possible open ambiguities concern whether the lexing worker is acting on characters belonging to a non-string lexeme, a string, or a comment.

PARALLEL LEXICAL ANALYSIS Each worker, save for the first one, starts with the initial ambiguity of being either at the beginning of a proper lexeme, or within a multi-line comment, or within a multi-line string. The worker carries on two computations: one of them handles simultaneously the first two possibilities, while the other deals with the third one.

The workers starting as if they are at the beginning of a lexeme, or in a multi-line comment, process their chunk along a single computation both matching the lexemes and keeping track of all the comment marker positions it encounters in a list. Whenever they find a closing multi-line comment symbol, they go back to the state where the beginning of a lexeme should be matched, keeping track of the comment-closing point. The position of the delimiters of multi-line comments is used in the last phase, to find which portions of the token stream should be kept and which ones should be discarded. The workers must deal with two possible causes of ambiguity:

I) Closing a multi-line comment within a string. The closing symbol of a multi-line comment may occur within a multi-line string beginning in the same chunk. In this case it is not possible to determine whether the closing symbol ends a construct started in a previous chunk or belongs to the contents of the current string. To solve this, the worker keeps tokenizing the character stream, until a closing string symbol is met. The worker memorizes the position of the closing string token, allowing for the identification of the string end during recombination.

II) Closing a multi-line string within a comment. A second ambiguity may appear, conversely, when the ending symbol of a multi-line string which began in the same chunk occurs within a single-line comment or a single-line string (delimited by quotes) and the worker cannot ascertain whether the closing delimiter represents the end of a previously interrupted string. The worker has to start two simultaneous computations, collecting the following tokens into two corresponding lists. The two computations can be possibly merged again into a single one when –and if– the ambiguity can be solved.

The workers can also reduce the two initial computations to a single one, eliminating the ambiguity, either when an error occurs in one of the two runs and the corresponding execution is aborted or when the worker reads a symbol with an overloaded semantics (as, e.g., `]]`). In particular, a sequence of two closing square brackets `]]` may represent either the end of a multi-line comment, or the end of a multi-line string or two lexemes used to index a table (e.g., the two closed brackets in `a[b[i]]`). If this sequence occurs, the worker may continue the scanning along the first computation only, since the following characters of the chunk lie necessarily outside of such comments or strings, and may only start a new lexeme or belong to another type of multi-line comment; thus, the two possibilities are handled by a single computation, as stated above.

LIST JOINING PHASE Starting from the first chunk, the list of delimiters produced by each worker is scanned, matching open and closed markers of multi-line strings or comments. The actual starting state of the analysis for each chunk is thus identified by checking the presence of possible open delimiters in previous segments, and the final token list is built by concatenating the portions of token lists generated along the correct computations on the chunks.

Example 5.2. Consider again the grammar of arithmetic expressions in Figure 37, extended with the possibility of employing the multi-line string and comment definitions of Lua, including the restrictions required to apply our parallel lexing schema. Consider the following source code chunk, which gets assigned to a worker for lexing:

```
x + [[a string]] × [[this
string may end here ]=]
y + [[another string]] + z
```

The worker scans the chunk along two computations, C_{token} and C_{string}, corresponding to the hypotheses of starting at the beginning of a token or inside a multi-line string respectively. C_{token} generates deterministically the partial list of tokens $\boldsymbol{n} + \boldsymbol{s} \times$. Subsequently, while it's matching the contents of a multi-line string beginning upon the `[[` delimiter, it detects the closing comment symbol `]=]` and memorizes its position for recombination. Upon the recognition of `]=]`, the worker needs to manage the possibility of the `]=]` symbol being an actual end of a comment which began on a previous chunk. To this end, C_{token} computation diverges on two paths: $C_{token-1}$ resumes the computation from the state corresponding to the beginning of a token and moves on, while $C_{token-2}$ scans the segment following the closing comment symbol as if contained the interior of the string. Upon matching the `]]` delimiter, $C_{token-1}$

has recognized the tokens $\boldsymbol{n} + \boldsymbol{s}$ and annotates the presence of the matched `]]` symbol, while $C_{token-2}$ completes the string recognition and merges back with $C_{token-1}$ yielding the initial computation C_{token}. C_{token} then recognizes the last tokens $+\boldsymbol{n}$.

The second computation C_{string}, on the other hand, scans the segment until the first `]]` (on the first line of the chunk) is encountered, collecting the characters as they were a portion of a string token, and is merged with the ongoing computation $C_{token-1}$ from that point on.

After analyzing the code chunk, information from the previous chunks is employed to determine whether C_{token} or C_{string} has performed the correct computation, and discard the incorrectly lexed tokens accordingly.

5.4 ADAPTING GRAMMARS TO PARALLEL ANALYSIS

Almost no grammar in its original user-oriented BNF is immediately ready to be used as an input for a general-purpose deterministic parser generator. For instance, to be suitable for a classical top-down parser such as LL ones, a grammar must avoid left-recursive derivations such as $A \overset{*}{\Rightarrow} A\alpha$, which must be automatically transformed into right-recursive ones. Typically, official technical language definitions do not even comply with the requirements of an LR(k) grammar, as their syntax specification exhibits shift/reduce or reduce/reduce conflicts that require a refactoring of the grammar so that it can be properly handled by parser generation tools. Even worse, most modern programming languages tend to be highly – perhaps too much – liberal towards the users and allow for overloading some symbols, e.g. parentheses, and/or using different symbols as aliases, e.g. ‘;’ and newline. Among those, it is also well known the extreme case of Perl whose parsing has been proved undecidable [60]. Thus, it is common practice and need, before building a compiler front-end for any new language, to carefully redesign its grammar to make it well-suited for the chosen deterministic parsing algorithm.

Such a preliminary work is needed as well to exploit an algorithm based on operator precedence. In our experience, the effort required to transform the official language specification of a technical language in OP form exceeds, but is comparable with,

S → OBJECT
OBJECT → { } | { MEMBERS }
MEMBERS → PAIR | PAIR, MEMBERS
PAIR → STRING : VALUE
VALUE → STRING | number | OBJECT | ARRAY | bool
STRING → “ ” | “ CHARS ”
ARRAY → [ ] | [ELEMENTS]
ELEMENTS → VALUE | VALUE, ELEMENTS
CHARS → CHAR | CHAR CHARS CHARS → char | char CHARS
CHAR → char

Figure 39.: Official JSON grammar. The productions for nonterminals CHARS and CHAR (highlighted in red) are replaced by the one highlighted in green to transform the grammar in OP form.

the one necessary to apply a standard LR or LALR-based parsing algorithm such as those used by Bison. We do not believe that such an increased difficulty hides a real impossibility due to the lesser theoretical generative power of OPGs w.r.t. LR ones: so far we did not find in real programming languages features preventing a language from being generated by an OPG, such as the ones reported in Section 5.2.

A drawback of the proposed approach is that the readability of the OP compliant grammar is lower than the one of the LL and LR ones. However, the programmer employing the language targeted by the syntax analyzer will not need to be aware of the employed parsing strategy, and thus of the actual grammar employed by the syntactic analyzer.

In this section we show how we managed the two languages we employed as case studies, i.e., JSON and Lua. Both have been treated with ad-hoc techniques and it turned out that the adaptation to OP form was trivial in the case of JSON, while it required more effort (a few man-days) in the case of Lua; other attempts – e.g., with JavaScript – generated more problems and suggested to resort to a more systematic and fully automated approach. We note that OPGs are already available in the literature for other programming languages, with considerable syntactic richness, such as ALGOL 68 and Prolog [47, 39].

5.4.1 *The case of JSON*

The official JSON syntactic grammar, reported in Figure 39, can be trivially transformed into operator precedence form. The only required modification is the one replacing the productions of the CHARS and CHAR nonterminals, to generate a sequence of printable alphanumeric characters, with a plain right-recursive rule.

5.4.2 *The case of Lua*

Tackling the transformation of the Lua grammar in OP form has proven more challenging than with JSON. As it happens for other standard technical definitions of classical programming languages, Lua's syntactic grammar (defined in the official reference manual [85] is not expressed in LR form. Concerning its transformation in OP form, the significant issues to be dealt with are the following ones:

1. The language statement terminator `;` is optional and can be replaced by either a white-space and/or a newline character.
2. Function definitions and calls allow one or more newline characters to appear between the function name and the parameter list. For instance:

   ```
   a = b + c
   (print or io.write)("done")
   ```

   is to be interpreted as

   ```
   a = b + c(print or io.write)("done")
   ```

3. Functions are first-class citizens in Lua, thus they can be returned as the result of a call to another function. This feature, in combination with the possibility of employing an in-place defined table as the single parameter passed to a function, allows to write the following code snippet:

   ```
   i = SecOrderFunct{A=3,B=25}
   ("hazelnut","strawberry")
   ```

where SecOrderFunct is a call to a second order function. The resulting first order function is subsequently invoked with the `hazelnut` and `strawberry` strings as parameters.

To deal with the above issues, we imposed the following constraints on the sources to be parsed (besides the lexicon restrictions described in Section 5.3.2), following what we have observed to be the best programming practices in Lua.

1. Multiple statements on the same line must be separated by a `;` character.
2. Comments between statements must be preceded and/or followed by a newline.
3. Multi-line comments are always followed by a newline.
4. We forbid the presence of newline characters between the in-place table declaration of a function parameter, and the parameter lists of the possible lower-order functions returned as the result. This prevents the programmer from using the same code indentation of the example at point 3. Possible ways to reformat are:

   ```
   i = SecOrderFunct{A=3,B=25}("hazelnut","strawberry")
   ```

   and

   ```
   i = SecOrderFunct{A=3,B=25}("hazelnut",
   "strawberry")
   ```

We emphasize that, in the whole set of real world code-bases examined for regression testing purposes, no violation of these constraints has been found. Moreover, the aforementioned constraints stand well within the common best practices in programming, allowing a better readability of the source code.

Provided the aforementioned constraints are respected by the source code, it is possible to solve all the remaining issues to allow OP parsing of Lua by means of a proper token relabelling done during the lexing phase. The transformations applied by our

Parallel-lexing algorithm are:

Token Disambiguation. The overloading of various tokens is disambiguated by emitting specialized tokens during the lexing phase. For instance, separate tokens are emitted for:

1. `;` used as a separator between statements or as a separator between fields in a table
2. = within assignment statements or in the initialization of table fields
3. The round parentheses enclosing a function parameter list and all the others
4. The classical ambiguity between unary and binary minus

It is easy to verify that translations (3) and (4) can be performed by a finite state automaton. Transformations (1) and (2), instead, need a stack to distinguish whether the innermost context where the symbols occur is a statement or a table; such an ad-hoc stack managing is fully integrated with our *Parallel-lexing* algorithm, and does not incur in a significant performance penalty as the lexical grammar of Lua is not regular anyways.

Semicolon Insertion When a newline is employed as a separator between a token that ends a statement, i.e, an element of the set S = {`nil`, `false`, `true`, a number, a string, `...`, }, `)`, `]`, `name`, `end`} and `(`, `name` or any other initial keyword for a statement, (i.e. `break`, `if`, `do`, `while`, `local`, `for`, `function`, `repeat`, `::`, `goto`) we replace the newline with a semicolon: in this way all statements are separated by a semicolon. Note that this substitution does not add a semicolon between the closing parenthesis of a function parameter list and the beginning of the function body. We also insert a semicolon between the elements of these two sets whenever they are separated by a comment.

Thanks to these transformations, the output produced by our enriched *Parallel-lexing* algorithm is fully compatible with the operator precedence constraints. We were thus able to define an OPG which matches the Lua programs with the above restrictions (for completeness the OPG of Lua is reported in the Appendix). One more step is necessary, however, to enable the parallel parsing algorithm described in Section 5.2: the grammar must be not only an OPG but must also be in Fischer normal form. This last step is performed by the well-known algorithms to eliminate renaming rules and repeated r.h.s. While the previous transformations did not affect significantly the size of the original Lua grammar, in this case the total number of nonterminals and productions increased, respectively, from 38 nonterminals and 144 rules to 49 nonterminals and 8547 rules. However, this increase in the size of the grammar is perfectly tolerable from the point of view of the memory fingerprint of a modern system and does not affect at all neither the run-time efficiency nor the end programmer (since she can fully ignore the new syntax grammar). This last transformation does not significantly alter the shape of the abstract syntax tree (AST) corresponding to a language sentence, save for the compression caused by the elimination of the renaming rules.

As a summary of the actions taken to obtain OPG grammars for both Lua and JSON we can state that performing this step for JSON was trivial, as only one rule modification was required (see Figure 39). Lua, on the other hand, proved to be in need of intervention not only to obtain an OPG description, but also to have an LR(1) one that could be provided as an input to Bison. The developer effort required to transform the reference Lua grammar in LR(1) form was comparable, if a bit smaller, than the one required to obtain its OPG. We note that the actual LR(1) Lua grammar, ready to be employed in Bison, is constituted of 143 productions, with the resulting LALR pilot automaton being 243 states wide, which is in itself quite a sizable one.

5.5 PAPAGENO TOOLCHAIN STRUCTURE

In this section we describe the general architecture of the PArallel PArser GENeratOr (PAPAGENO) toolchain for parallel operator precedence language analysis, in which we implemented the algorithms described in previous sections. PAPAGENO is an open source project available under GNU General Public License and it is written in ANSI/ISO C and Python: the codebase can be downloaded at [83].

The PAPAGENO toolchain provides an automatic parallel parser generator that converts a specification of a syntactic grammar into an implementation of the oper-

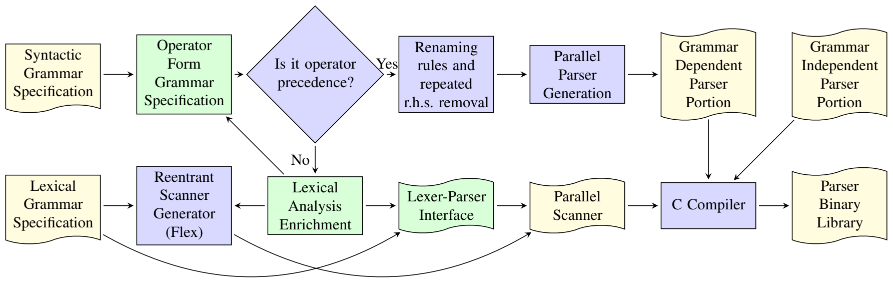

Syntactic Grammar Specification
Operator Form Grammar Specification
Is it operator precedence?
Yes
Renaming rules and repeated r.h.s. removal
Parallel Parser Generation
Grammar Dependent Parser Portion
Grammar Independent Parser Portion
No
Lexical Grammar Specification
Reentrant Scanner Generator (Flex)
Lexical Analysis Enrichment
Lexer-Parser Interface
Parallel Scanner
C Compiler
Parser Binary Library

Figure 40.: Typical usage of the PAPAGENO toolchain. The human operator stages are marked in green, while the PAPAGENO automated staged are marked in blue.

ator precedence parallel parsing algorithm described in Section 5.2. The generator has been already implemented and described in [15]. The contribution of this thesis to the architecture of PAPAGENO consists in the design and implementation of parallel scanners, which complement the parallel parsers generated by the tool, hence obtaining a complete parallel lexer and parser library. The programming language of choice for the library is C, as it provides strict control over the computation process and memory management. For the sake of portability, all the C code generated by PAPAGENO employs fixed-size types standardized in the C99 standard; furthermore, it relies exclusively on the standard C runtime and a POSIX-compliant thread library, thus avoiding any architecture-specific optimization. The generated lexers and parsers have been run successfully on x86, x86_64, ARMv5 and ARMv7 based-platforms with no code modifications.

In the following, we will first detail the stages of PAPAGENO's workflow for parallel source analysis from the end-user standpoint. Second, we describe the design choices and optimization techniques which proved crucial in exploiting the parallelism exposed by the lexing and parsing algorithms.

5.5.1 *Architecture of PAPAGENO toolchain*

The architecture of the PAPAGENO toolchain is depicted in Figure 40. The input of the process contains the specifications of the lexical and syntactic grammars of the target language. If the syntactic grammar of the language is not in operator precedence form, the tool notifies the inconsistency in the input specification and the user is given proper diagnostics pointing out the rules where precedence conflicts or adjacent nonterminals occur. The user has thus to modify the grammar: a convenient approach to eliminate precedence consists in enriching the lexical analysis stage with proper transformations, as insertions or renaming of tokens. In Section 5.4 we described how to adapt into a form suitable for parallel parsing the two case study grammars of JSON and Lua.

Then PAPAGENO automatically eliminates from the OPG both the repeated r.h.s. rules and the renaming rules. At last, the C code of a parallel parser is generated.

The parallel parser generator in PAPAGENO has been designed as a replacement for the classical GNU Bison generator and adopts the same basic syntax conventions, allowing an easy porting of the grammar descriptions available in Bison-compliant format. The generated parallel parser is logically split into two parts, as shown in Figure 40: a language independent support library, and a language dependent parser code portion. This choice was made to allow for easy extensions and possibly further architecture dependent optimizations of the language independent portion, while retaining the automated code generation feature.

The parsing process is invoked by means of a function call, where the developer may specify at runtime the input stream to be analyzed and the number of workers to be employed to perform the analysis. Each worker is mapped to a single POSIX thread, belonging to a thread pool initialized at the beginning of the parsing process.

The developer can choose between two parallel parsing strategies in the generated code. In the first strategy, after a first execution of the parallel parsing algorithm, the recombination of the partial stacks is assigned to a single worker which operates in sequential mode. In the second strategy, instead, the first parallel pass of the parsing algorithm is followed by parallel recombination of the partially parsed substrings along the lines described in the general Algorithm 2: the number of initial workers is reduced by at least two, and each of the remaining workers has to recombine two partial parsing stacks generated in the first pass (the number of threads can be reduced even further if the part $\mathcal{S}_L$ or $\mathcal{S}_R$ of some partial stacks is empty). This recombination process is iterated until a single thread is left to complete the parsing. The second strategy aims at exploiting the parallelism offered by particularly deep parsing trees. We anticipate that in our case studies the input exhibits a regular or shallow tree structure, causing the difference in parsing time between the two strategies to be small. As stated in Section 5.2, we feel that quite seldom more than two recombination passes will be advantageous.

The PAPAGENO generated parsers can be naturally combined with either a sequential Flex generated scanner, or a parallel scanner resulting from the implementation of our algorithmic schema described in Section 5.3. Unlike the generation of a parallel parser, which is fully automatic, the phase of parallel lexer generation currently requires some interaction with the user.

In particular, the programmer is expected to provide the specification of the grammar in the Flex input format for reentrant lexers, write the code managing the input character stream splitting, and the one handling the token list recombinations. The input splitting code performs the actual chunking, possibly employing a fixed-width search window as described in Section 5.3, and inputs the data into the Flex-generated scanners. The multiple working states of the scanner are mapped onto the multi-state lexer features offered by Flex, requiring from the programmer the definition of the language-specific transitions from one state to the other. At the end of the parallel lexing process, the information on the multiple lexer is exploited by the code written by the programmer to perform the constant-time recombination of the token lists produced by the parallel lexers.

Finally, once the parallel scanner is obtained, as a combination of the output of Flex and the user's lexer-parser interface, it is possible to compile all the sources generated by the toolchain, resulting in a complete binary lexing and parsing library.

5.5.2 *Optimization Techniques*

The internal architecture of PAPAGENO relies on carefully designed implementation strategies and data structures, which play a fundamental role to obtain high performances of parallel lexers and parsers. In this section we recall the well-known bottlenecks preventing efficient parallelization and present the solutions adopted in our tool to cope with them.

Two commonplace issues in achieving practical parallelism are 1) the data representation and handling geared towards efficient memory use, and 2) a proper management of the synchronization issues, typically minimizing the use of locks. Thanks to the computationally lightweight parsing algorithm devised for OP grammars, and the minimal requirement for synchronization actions, issue 2) is less important for us, and memory management and memory allocation locality was found to be the crucial issue. Therefore, we start from a discussion of issue 1) and conclude with the synchronization requirements and thread orchestration performed by PAPAGENO generated parsers.

We describe several simple yet effective memory optimizations.

- First, we encoded terminal and nonterminal symbols as word-sized integers, taking care of employing one bit of the encoding to distinguish terminal from non-. By default, the most significant bit is used; however PAPAGENO allows to choose its position at parser generation time to allow room for further information packing. Such information packing does not prevent the definition of large target languages, as the architecture word length in modern devices is at least 32 bit, and 64 bit for most of them. Adopting this technique, we can do without a look-up table to check whether a symbol on the parsing stack is a terminal or non-.

- A second optimization towards improved data locality comes from the observation that the precedence relation between may take one out of four values ($\lessdot, \doteq, \gtrdot, \perp$). Using a bit-packed representation of the precedence matrix, we obtain significant savings for large matrices (which occur in large languages), and, moreover, we manage to fit entirely the matrix in the highest level caches, thus significantly improving the average memory access latency.

- Furthermore, in order to avoid serialization among the workers upon the system calls for dynamic memory allocation, we adopt a memory pooling strategy for each thread, wrapping every call to the `malloc` function. This strategy has also the advantage of reducing memory fragmentation, since the memory allocation is done in large contiguous segments. To evaluate the memory needed for pre-allocation during parsing, we estimate the number of nodes of the parsing tree by computing the average branching factor of the AST as the average length of the r.h.s. of the productions. Then, the parallel parser generator initially pre-allocates half of the guessed size of the AST and augments the memory pool of a worker by one fifth of this quantity, every time the thread requires more memory. A similar memory pooling strategy is employed in the lexing phase, in order to avoid serialization among the lexing threads in need for memory to allocate the token lists.

- One of the most computationally intensive parts of OP parsers is the matching of a production r.h.s against the ones present in the grammar. By representing the r.h.s.'s as a prefix tree (trie), it becomes possible to find the corresponding left-hand side in linear time with respect to the length of the longest r.h.s. of the grammar. Furthermore, to optimize the size and the access time to the trie, we followed the technique described in [50], that represents the structure as an array, storing the pointers to the elements of the trie within the same vector. To take advantage of the trie compression provided by this technique, we assume an upper bound of 2^{16} for the total number of terminal and nonterminal symbols, which clearly does not affect applicability for any common language. The vectorized trie is fully precomputed by PAPAGENO, and is included in the generated parser as a constant vector.

For the synchronization and locking issues in OP-based parallel parsing, we used rather straightforward techniques.

- Since each parallel worker performs the parsing action on separate tokenized input chunks, it is completely independent from the other workers, and there is no need for any synchronization or communication between them. This in turn allows the proposed strategy to scale easily even in the cases where the inter-worker communication has a high cost, e.g. whenever the input is so large that they have to be run on different hosts.
- Similarly, all the lexers act independently on the input, without need of communication or synchronization while performing the lexing actions.
- The requirement for enforced synchronizations is only present in the following two cases: i) a single barrier-type synchronization point is required between the end of the lexing phase, and the beginning of the parsing one whenever the lexical grammar requires a constant-time chunk combination action to be performed by the lexer; ii) synchronizations are required to enforce data consistency if the user desires to perform multiple parallel parsing recombination passes, instead of a single one.
- While the first barrier synchronization cannot be subject to optimizations, the synchronizations between multiple parallel parsing recombination passes can be fruitfully organized hierarchically. In particular, a parsing worker from the n-th pass will only need to wait for the completion of the $n - 1$ pass workers producing its own input, effectively avoiding the need of a global barrier synchronization between passes. Such a strategy allows to effectively exploit the advantages of multiple parallel passes whenever the parse tree is very high.

5.6 EXPERIMENTAL RESULTS

In this section, we present and discuss the experimental results of our parallel lexing and parsing system on both JSON and Lua languages.

THE BENCHMARKS We chose real world JSON and Lua inputs of various sizes, on which we performed the parsing and the construction of the abstract syntax tree (AST)

Table 4.: Total text analysis times of the JSON test-bench files, for both the server and mobile platform.

Input Size	Elapsed Time [ms]					
	Server			Mobile		
	Lexing	Parsing	Total	Lexing	Parsing	Total
2.7 kiB	0.6	1.8	2.4	0.8	1.8	2.6
30 kiB	2.7	5.1	7.8	2.8	5.6	8.4
80 kiB	7.7	15.4	23.1	8.3	18.0	26.3
150 kiB	15.0	37.4	52.4	24.8	69.0	93.8
1.6 MiB	98.4	255.0	353.4	153.1	431.1	584.2
10 MiB	588.1	1584.7	2172.8	1033.6	2665.3	3698.9
75 MiB	3462.6	8892.2	12354.8	-	-	-

in memory. This is the only semantic action associated to the parsing process. The rational is to evaluate the computational load of the parsing process, regardless of any subsequent use of the parsed data. Since in typical compilers the semantic actions are more computationally demanding than AST construction, it follows that even greater performance benefits can be achieved if they can be parallelized. In other words, our results evidence the speed-up that is achieved by parallelizing syntax analysis and nothing else.

For JSON, the set of inputs includes a shopping list from an online shop (30kiB), the configuration file of AdBlocker, a common browser plugin (80 kiB), the Gospel of John (150kiB), a statistic data-bank (1.6MiB) on food consumption (source Italian Institute of Statistics), a file containing statistics on n-grams present in English in Google Books (10MiB), and the index of all the documents available on the UK Comprehensive Knowledge Archive Network (75MiB).

For Lua, benchmarks were derived, instead, from the codebase of Lucasarts's Grim Fandango, which is available together with the game. This code-base size amounts to 2.5MiB, and, to obtain benchmarks of different sizes, a suitable number of compilation units have been concatenated together. To explore the scalability of the parallel parsing approach we tested the PAPAGENO generated analyzer with files ranging from 7 kiB to 35 MiB, with the ones larger than the whole code-base being generated by concatenating all the files from the code-base more than once.

HARDWARE PLATFORMS To evaluate the practical speedups obtained, we used two platforms:

1. A quad-Opteron 8378 host, thus amounting to 16 physical cores (4 cores per socket): the Opteron 8378 CPUs are endowed with independent, per CPU, L1 and L2 caches, and a chip-wide shared L3 cache. The host runs Ubuntu Linux 14.04 (`x86_64` architecture) server and is endowed with enough RAM to contain the whole AST materialized during the parsing process and token list. The purpose of the evaluation on this platform is to highlight the scalability of our approach, even in the context of a multi-socket system with a non uniform memory access.

2. An Odroid-XU Lite board, endowed with a Exynos 5 Octa SoC, which is driven by four Cortex-A15 and four Cortex-A7 CPUs, in big.LITTLE configuration, clocked at 1.4GHz, and 2GB of DDR3 DRAM. The platform runs a Debian 7.6 Linux (`armv7l` architecture), and the main choice is to use the four Cortex-A15 CPUs, as the architectural constraint do not allow to employ all the 8 cores simultaneously. The benchmarks run on this platform are representative of the actual performance benefits obtainable on a high-end embedded system, such as the ones which are increasingly more common in mobile phones and tablets. Such platforms are typically characterized by a uniform memory access, and limited main memory resources with respect to desktop machines.

All the executable binaries have been produced through `gcc` 4.9.1, employing standard release grade optimizations to obtain an efficient binary (`-O3 -march=native` optimization options). All the timing results presented have been collected employing Linux real-time clock primitives, and are the average of 50 runs to reduce measurement noise.

PURELY SEQUENTIAL EXECUTION Tables 4 and 5 report the absolute processing times respectively obtained for JSON and Lua, using a purely sequential PAPAGENO

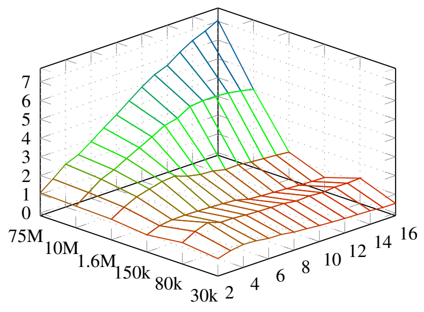
Table 5.: Total text analysis times of the Lua test-bench files, for both the server and mobile platform.

Input Size	Elapsed Time [ms]					
	Server			Mobile		
	Lexing	Parsing	Total	Lexing	Parsing	Total
7 kiB	0.9	2.2	3.1	1.0	2.3	3.3
70 kiB	5.8	9.7	15.5	7.2	12.0	19.2
700 kiB	24.6	41.7	66.3	63.5	98.2	161.7
3.5 MiB	105.8	161.0	266.8	212.8	314.5	527.3
7 MiB	203.4	313.4	516.8	424.4	602.4	1026.8
35 MiB	998.9	1559.3	2558.2	-	-	-

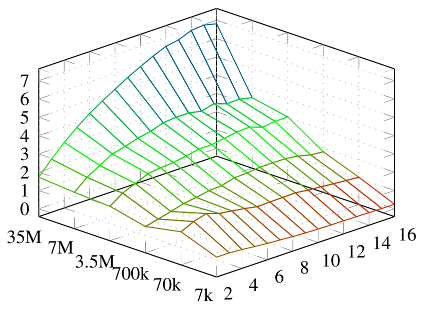
lexer-parser pair: they establish a practical baseline for comparison. Notice that the absolute times for the larger files are quite important, especially on the mobile platform. Moreover, the latter is not endowed with enough memory to materialize the whole AST for the largest test cases; as a consequence the two largest benchmarks cannot be run on it. A point worth noting is that, both in the case of JSON, and in that of Lua, the time spent in the lexical analysis of the input is non negligible: more specifically, it is around 30% for JSON and 40% for Lua. This result substantiates our claim that, for OP-based parsing, the lexical analysis accounts for a non trivial amount of the text processing time.

PARALLEL EXECUTION Figure 41 shows the speedup obtained by the parallel lexer versus a sequential run of a Flex generated lexer, while Figure 42 reports the speedups of the parsing phase of the computation, computed against a sequential run of a PAPAGENO generated parser.

Consider the results for the JSON lexing phase reported in Figure 41a: for all file sizes $\geq$ 80kiB, parallel lexing achieves a significant speedup over the plain sequential Flex generated lexer. This is more evident when the file size allows all the workers to perform a significant amount of computation.



(a) JSON: Lexing phase

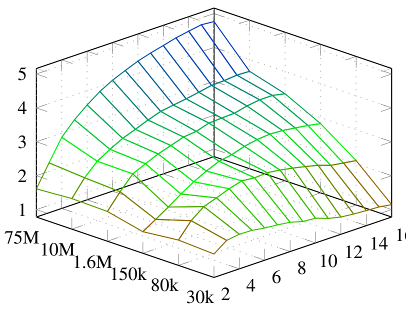


(b) Lua: Lexing phase

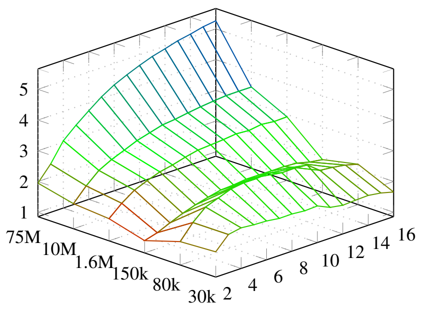
Figure 41.: Speedups achieved on the lexing for JSON and Lua, taken against a sequential lexer, depicted as a function of the number of workers and the input size expressed in bytes.

A point worth discussing is the relatively low speedup achieved when only a few workers are employed: in this case, the lightweight computation required by JSON lexing, together with the possible thread migration from one CPU to the other in the NUMA machine employed, negatively affect the performances. More in detail, a thread migration from one CPU to another implies a significant drop in the effectiveness of the caches, as the computation is moved to a processor where the working-set is not pre-heated in cache. By contrast, a higher load on all the available CPUs will prevent the scheduler from moving the tasks in an attempt to equalize the load. Although this issue can be solved pinning the threads to a specific CPU through processor affinity settings, we chose not to perform the measurements with such a technique as it may yield overly optimistic results with respect to running environment where CPU pinning is forbidden (e.g. large data-centers where the computation is taking place inside virtual machines). To finish, we note that the maximum achieved speedup is 7× in the case of a 75 MiB JSON file, cutting down its lexing time from 12 sec. to less than two.

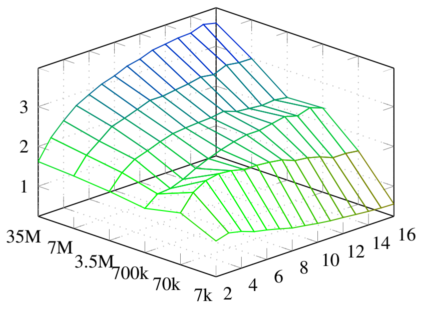
The lexing phase results for Lua (Figure 41b) confirm the speedups achievable through the parallelization of the lexing stage, even in the case of a lexical grammar much more complex than JSON one. We know Lua parallel lexer needs to perform



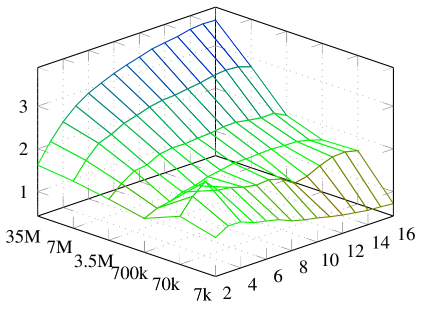
(a) JSON: Parsing with sequential lexer



(b) JSON: Parsing with parallel lexer



(c) Lua: Parsing with sequential lexer



(d) Lua: Parsing with parallel lexer

Figure 42.: Comparison of speedups achieved on the lexing and parsing phase for JSON and Lua employing either a sequential or a parallel lexer, taken against a sequential PAPAGENO parser. The speedups are represented as a function of the number of employed workers, and the input size in bytes

a non-trivial recombination at the end of lexing, and the results show that the recombination phase does not impact adversely performances. The reduced performance gained on the 150kiB input file is to be ascribed to low level cache contention, as the file exceeds by a small amount the least level cache size for the involved CPUs.

The JSON parsing phase (Figures 42a and 42b) also benefits from significant speedups (up to 5.3×) for large files, and show how a parallel parsing approach is advantageous even in the case of small files. In particular, Figures 42a and 42b report the speedups achieved during the parsing phase only, for an implementation with a

Flex-generated sequential lexer (Figure 42a), and a parallel lexer (Figure 42b). Comparing two situations, we get an interesting insight on the use of a parallel lexer. As it can be seen, combining the parallel parser with the parallel lexer, has a positive synergistic effect, even in the case of small files, yielding effective speedups already for the 30kiB file. This effect is to be ascribed to the L2 and L1 cache pre-heating effect caused by having the text lexically analyzed by different independent workers. In fact, such an approach is more likely to be fetching the data which will be parsed by a worker into the dedicated L1 and L2 caches of the corresponding core, effectively reducing the memory pressure for the parsing action, and thus increasing the performances. As a further confirmation of this fact, we note that the performance boost does not take place in the case of the parsing of the 150kiB file, which is a good cache fit already with a sequential lexing process.

Concerning the Lua parsing phase, Figures 42c and 42d show how this can be effectively parallelized, notwithstanding the much richer structure of the language. On the other hand, the synergistic effect between parallel lexing and parsing in Lua is less evident than in JSON; we ascribe this fact to the higher memory requirements for the Lua parsing process, which in turn add extra pressure on the caches, preventing the pre-heating from having a significant impact.

COMPARISON WITH FLEX AND BISON Since the lexers and parsers produced by Flex and Bison are the current state-of-the-art for tool generated language processors, it is interesting to compare in Figure 43 the performance of the parallel lexer/parser library generated with PAPAGENO, against a text analysis library produced by Flex and Bison, selecting the LALR(1) parser generation algorithm. The results evidence, in both the server and mobile platform, a significant speedup with respect to the state-of-the-art of tool generated parsers, for all but the smallest test-bench files; and a good scalability of the approach. We discuss two aspects in particular:

- The ability to exploit per-die caches in JSON parsing leads to significant benefits for small texts, whenever the parsing action is contained within a single multicore CPU on the server platform, i.e., up to 4 simultaneous workers.

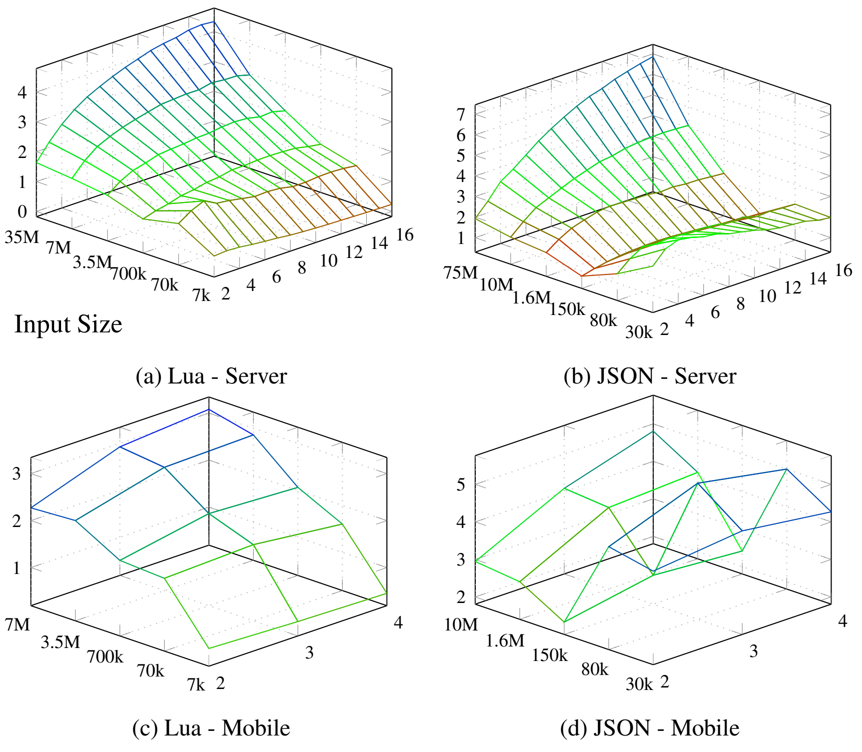
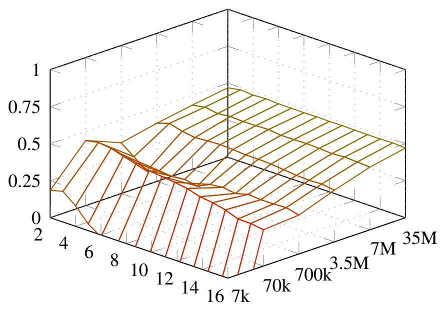


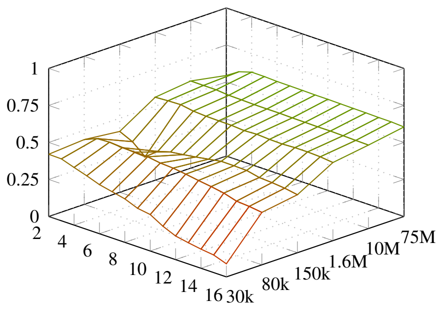
Figure 43.: Speedups obtained against a sequential Flex-Bison generated text analyzer on both the server and mobile platforms. Darker lines indicate smaller input files.

- The simplicity of the OP parsing algorithm represents an effective advantage on RISC architectures, such as the one of the ARM platform, where it is able to obtain a speedup of up to 5.5×, as a combination of the parallel processing technique on multiple cores and the lesser computational requirements with respect to a classic LALR(1) parser.

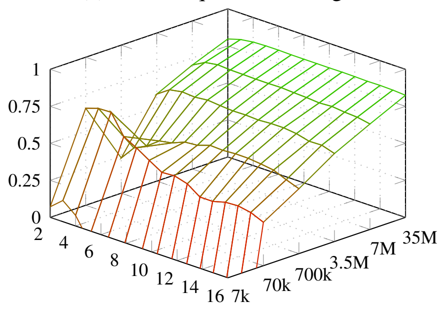
AMOUNT OF PARALLEL CODE We maintain that performing the lexing actions in parallel gives a substantial advantage in terms of the actual amount of code which



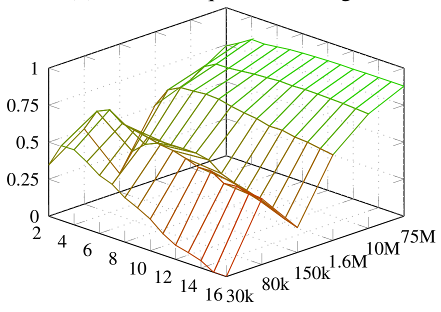
(a) Lua - Sequential Lexing



(b) JSON - Sequential Lexing



(c) Lua - Parallel Lexing



(d) JSON - Parallel Lexing

Figure 44.: Parallel code portion in both the JSON and Lua lexing/parsing process obtained as the complement to the sequential code portion obtained via Karp-Flatt metric, plotted as a function of the number of workers, and the size of the input expressed in bytes.

is executed in parallel. To substantiate this claim, we computed the sequentially executed code portion e from our obtained speedups according to the Karp-Flatt metric [59] as $\frac{1/s-1/p}{1-1/p}$, with s the achieved speedup, and p the number of involved processors. We chose to employ the Karp-Flatt metric as it is designed to provide a concrete counterpart to Amdahl's law, as the latter only states the maximum achievable speedup, while assuming no interference by the operating system and runtime on which the processes are run. Figure 44 depicts the portion of code which is executed in parallel, obtained as $1-e$, for both JSON and Lua parsing on the server platform, with Figures 44a and 44b reporting the serial-lexer implementation while Figures 44c and 44d report the parallel-lexer one. It is evident that, eliminating the parallelization of the lexing action, has the effect to significantly reduce the parallel code portion, namely around 20%-25% for JSON and 40% for Lua for all the input sizes/number of workers combinations, where the operating system overhead is not dominating the computation. Moreover, the parallel portion analysis additionally shows how an input size increase determines a larger portion of the code being executed in parallel, in turn implying that the scaling of our approach on the input size does not introduce significant system overhead, thus allowing the user to fully reap its benefits.

Finally, it can be noticed that, for sufficiently large inputs, the overhead of spawning more workers is negligible: this is a consequence of the absence of communication between the workers, which only need to synchronize at the end of their computation. This in turn translates into a high efficiency in scaling to high amounts of parallel threads.

5.7 RELATED WORK

The literature on parallel parsing and lexing is vast and extends over half a century.

5.7.1 *Parallel parsing*

As regards parsing, the valuable survey and bibliography [7] lists some two hundred publications in the period 1970 - 1994, and research has continued since, though perhaps less intensively. It is worth contrasting this quantitative aspect with the paucity of existing realizations and, even more so, of tools for producing parallel parsers. We omit, as less relevant to our objectives, some categories: the work on grammar types not belonging to the context-free family, the studies based on connectionistic or neural models of parallel computations, and the large amount of work on natural language processing.

We are left, roughly speaking, with the following categories:

1. *Theoretical analysis* of algorithmic complexity of parallel context-free language recognition and parsing, in the setting of abstract models of parallel computation, such as P-RAM.

2. *Parallel-parser design and performance analysis* for specific programming/web languages, sometimes combined with experimentation, or, more often, simply with demonstration, on real parallel machines.

Category 1. is mainly concerned with the asymptotic complexity of recognition/-parsing algorithms on abstract parallel machines. The algorithms proposed for unrestricted CF grammars require an unrealistic number of processors: for instance Rytter's [87] recognizer has asymptotic worst-case time complexity $O(\log n)$, with n the input length, and requires $O(n^6)$ processors; the numbers of processors grows to $O(n^8)$ if parsing, i.e., syntax-tree construction, is required. Several researchers have shown that such complexity bounds can be lowered, by restricting the language class, sometimes so much that it loses practical interest. We mention some cases, from the simplest to the more general ones, for the recognition problem. For the *input-driven* (also known as *visibly pushdown*) languages, the time complexity is $O(\log n)$ and "only" $O(n/\log n)$ processors are used [52]. The *deterministic CF* languages are recognized in time $O(\log n)$ on a P-RAM machine using $O(n^3)$ processors [61]. A series of papers (e.g., [30]) have gradually refined the complexity bounds for the case of unambiguous CF languages.

Such idealized results are, of course, not really comparable with experimental findings, as already asserted by [7], yet they offer some interesting indications. In particular, all the subfamilies of deterministic CF languages for which the theoretical complexity analysis reports a close to linear use of processors, are included in the family of OP languages we use.

Such abstract complexity studies had little or no impact on practical developments, for several reasons. First, it is known that the abstract parallel machines, such as P-RAM, poorly represent the features of real computers, which are responsible for performance improvements or losses. Second, asymptotic algorithmic time complexities disregard constant factors and mainly focus on worst cases, with the consequence that they are poorly correlated with experimental rankings of different algorithms.

Last, most theoretical papers do not address the whole parsing task but just string recognition. In the following years 1995-2013 the interest for research on the abstract complexity of parsing algorithms has diminished, with research taking more practical directions.

The classical *tabular* recognition algorithms (CKY, Earley) for unrestricted CF languages have attracted much attention, and a number of papers address their parallelization. It is known that such parsers use a table of configurations instead of a pushdown stack, and that their time complexity is related to the one for matrix multiplication, for which parallel algorithms have been developed in many settings. Parallel algorithms derived from CKY or from Earley sequential parsers (*s-parsers* for brevity) may be pertinent to natural language processing, but have little promise for programming/data description languages. As tabular s-parsers are significantly slower than LR or OPG s-parsers (up to some orders of magnitude), it is extremely unlikely that the parallelization of such a heavy computational load would result in an implementation faster than a deterministic parallel parser. Moreover, as we are not aware of existing tabular parallel-parser generation tools, confirmation by experiment is not possible at present.

The comparison with previous work in category 2. is more relevant and reveals the precursors of several ideas we use in our generator. We only report on work dealing, as our own, with deterministic CF languages.

BOTTOM-UP PARSING Some early influential efforts, in particular [45] (described in [32]) and [77], introduced data-parallelism for LR parsers, according to the following scheme: a number of LR s-parsers are run on different text segments. Clearly, each s-parser (except the leftmost one) does not know in which parser state to start, and the algorithm must spawn as many deterministic LR s-parsers as the potential states for the given grammar; each parser works on a private stack. When a parser terminates, either because it has completed the syntax tree of the text segment or because the lack of information on the neighboring segments blocks further processing, the stack is merged with the neighboring left or right stack, and the s-parser process terminates; similar ideas occur in other papers too. However, the idea of activating multiple deterministic bottom-up s-parsers is often counterproductive: the processes, associated to the numerous parser states of a typical LR grammar, proliferate and reduce or nullify the speedup over sequential parsing.

Two ways of reducing process proliferation have been proposed: by *controlling the points of segmentation*, and by *restricting the family of languages* considered. An example of the first is in [89], so explained: "The given input string . . . is divided into approximately q equal parts. The i-th processor starting at token . . . scans to the right for the next synchronizing token (e.g., *semicolon*, *end*, etc.) and initiates parsing from the next token". If synchronizing tokens are cleverly chosen, the number of unsuccessful parsing attempts is reduced, but there are drawbacks to this approach: the parser is not just driven by the language grammar, but needs other language-specific indications, to be provided by the parser designer; thus, [89] chooses the synchronizing tokens for a Pascal-like language. Furthermore, to implement this technique, the lexer too must be customized, to recognize the synchronizing tokens.

Similar language-dependent text segmentation policies have been later adopted by other projects, notably by several developments for XML parsers; such projects have the important practical goal to speed-up web page browsing, and investigate the special complexities associated to parallel HTML parsing. Although they do not qualify as general purpose parsers, their practical importance deserves some words. The recent [102] paper surveys previous related research, and describes an efficient parallel parser, *Hpar*, for web pages encoded in HTML5. HTML5 has a poorly formalized BNF grammar and tolerates many syntax errors. A HTML5 source file may include a script (in JavaScript), which in turn can modify the source file; this feature would require costly synchronization between lexing and parsing threads, which make a pipelining scheme inconvenient. *Hpar* splits the source file into *units* of comparable length, taking care not to cut an XML tag. Each unit is parsed by an independent thread, producing a partial DOM tree; at last, the DOM trees are merged. A complication comes from the impossibility to know whether a *unit*, obtained by splitting, is part of a script, a DATA section, or a comment. The parser uses heuristics to speculates that the unit is, say, part of a DATA section, and rolls-back if the speculation fails (in Section 5.3 we have described a similar approach to parallel lexing.) More speculation is needed for another reason: when a unit parser finds a closing tag, say < `/Table` >, it does not know if the corresponding opening tag occurred before in the preceding unit, or if it was missing by error. The best speedup achieved (2.5× using five threads) does not scale for the current web page sizes.

Returning to parsers purely driven by the grammar, in view of the popularity of (sequential) LR parser generation tools like Bison, the fact that no parallel-parser generators exists is perhaps an argument against the feasibility of efficient parallel parsers for LR grammars. This opinion is strengthened by the fact that several authors have developed parallel parsers for language families smaller than the deterministic CF one, but it would be too long to cover all of them, and one example suffices. The grammars that are LR *and* RL (right to left) deterministic enjoy some (not quantified) reduction in the number of initial parser states to be considered by each unit parser. Such grammars are symmetrical with respect to scanning direction: rightwards/leftwards processing, respectively, uses look-ahead/look-back into the text to choose legal moves. By combining the two types of move into a bidirectional algorithm, dead-ended choices are detected at an earlier time. We observe that Floyd's OPGs too have the property of reversibility with respect to the parsing direction and benefit from it for making local parsing decisions, which are unique and guaranteed to succeed if the input text is grammatically legal.

Indeed, thanks to the local parsability property, OP languages do not incur in the penalties that affect LR parsers; the latter, as said, need to activate multiple computations for each deterministic unit parser, since many starting states are possible. For OP parsers, in fact, all the actions can be deterministically taken by inspecting a bounded context (two lexemes) around the current position, and do not depend on information coming from the neighboring unit parsers: thus, each text unit can be processed by an OP parser instance along a single computation, without incurring on the risk of backtrack.

To complete the topic of restricted CF language families suitable for local parsability, we mention two papers not surpassing the preliminary proposal stage. A list of requirements for local parsability is in [63]. The work we consider to be closest to our choice of OPGs is [73], that uses bounded-context grammars, a grammar model [48] generalizing OPGs, which however has been rarely considered for s-parsing.

TOP-DOWN PARSING Less effort has been spent on top-down deterministic LL parsers, possibly because, at first glance, their being goal-oriented makes them less suitable for parsing arbitrarily segmented text. The article [97] surveys the state-of-the art for such parsers and reports in detail a parallel (non-experimented) algorithm that works for a subclass of LL grammars, named LLP. Imagine that the text is segmented into substrings and on each segment a classical LL(k) s-parser is applied. Similar to the LR case, each s-parser does not know the result (i.e., a stack representing the prefix of a leftmost derivation) for parsing the substring to its left: therefore each s-parser has to spawn as many s-parsers working on the same segment, as there are possible initial stacks, too many to be practical. Therefore it is proposed to limit the number of possible initial stacks by imposing a restrictive condition on LL(k) grammars. The subfamily thus obtained is named $LLP(q,k)$ and is based on the idea of inspecting a look-back of length q tokens as well as the classical look-ahead of k tokens. Although not compared in the paper, $LLP(q,k)$ grammars look quite similar to the already mentioned bounded-context grammars. This and earlier studies on parallel LL parsers may be theoretically interesting but do not offer any hint on practical usability and performances.

5.7.2 *Parallel lexing*

The problem of breaking up a long string into lexemes is a classical one for data parallel algorithms, well described in [56]. They assume, as such studies invariably do, that each lexeme class is a regular language, therefore the sequential lexer is a deterministic finite automaton (DFA) that makes a state transition reading a character. For a string x, the chain of state transitions define a *lexing function* that maps a state p to another state q; moreover the function for the string $x \cdot y$ obtained by concatenation is obtained by function composition. The data-parallel algorithm is conceptually similar to the one for computing all partial sums of a sequence of numbers, also known in computer arithmetic as the parallel *sum prefix* algorithm. In essence, the source text is split into pieces, and the DFA transition function is applied to each piece, taking each DFA state as a possible starting state. Then the functions obtained for neighboring pieces are composed and the cases of mismatch are discarded. Such processing can be formulated by means of associative matrix operations. This parallel algorithm is reported to be optimal from a purely theoretical viewpoint, but early simulation on fine-grained architectures with very many processing units is not conclusive. More recently, various experiments of similar algorithms on GPGPU and on multi-core architectures have been reported. A criticism is that such algorithms are very speculative, performing a significant amount of computation which may be later on discarded, thus yielding fairly poor energy efficiency. Some authors have considered the *regular expression matching* problem, instead of the *lexing* problem, and, although regular expressions and DFA models are equivalent, the parameters that dominate the experiments may widely differ in the two cases. An example suffices: [90] presents a notable new version of the mentioned [56] approach. They claim that for certain practical regular expressions that are used in network intrusion/detection systems, the size of the parallel lexer remains manageable and not bigger than the square of the minimal DFA. Then, they are able to construct the parallel scanner on-the-fly, i.e. delaying as much as possible the construction of the states. Clearly, algorithm [90] is not intended as a lexer to be invoked by a parallel parser, but as a self-standing processor for matching regular expressions – yet partially so, since it does not address the central issue of ambiguous regular expression parsing, which fortunately does not concern our intended applications.

Recently, [91] has experimented on the Cell Processor a parallel version of the Aho-Korasick string matching algorithm. This work was motivated by the good performance of that algorithm on multi-core machines for string search against large dictionaries. But a downside of that approach is that it apparently assumes that the input file can be unambiguously divided into text segments; therefore it does not apply to the case of general programming- or data-representation languages, since, for such languages, scanning cannot avoid an initial degree of nondeterminism caused by the absence of a separator between tokens (as a newline) that could be identified by inspecting a bounded portion of the segments.

Compared with the mentioned studies, our approach to parallel lexing in Section 5.3 addresses further critical issues. First, the approach is suitable for more general lexical grammars that involve pushdown stacks and cannot be recognized by a

DFA (as the lexical grammar of Lua). Second, our approach integrates some preprocessing steps that enhance the performance of the following parsing stage. Furthermore, we address a complexity that made previous approaches such as [56, 77] unpractical: splitting the input file into segments may cause ambiguity, in the sense that the lexing function associated to a segment may return multiple values (states), depending on the assumed input state. To compute such function, several workers are needed, but in our design their number does not equal the number of states of the automaton, but is limited to two or three, and does not critically affects performance, as attested by the experimental results achieved by PAPAGENO.

5.7.3 *Local parsability property*

Besides parallel parsing, the local parsability property of OPLs has been exploited in another research direction, to support *incremental parsing* of software artifacts.

It is in fact universally acknowledged that, thanks to its malleability, software is subject to continuous evolution, whether for corrective or evolutionary maintenance. Most often the changes applied to a large program are local, as they affect only a small fraction of its syntax tree. This asks naturally for incremental parsing, i.e., to modify the existing parse only in the affected part without redoing much identical work. Starting from the early work [51] a fairly rich literature on incremental parsing has been developed (see, e.g., [41, 64]) which, unlike the case for parallel parsing, has also produced several practical tools. Such results, however, normally concern more widely adopted families of deterministic languages, which do not enjoy the local parsability property; OPLs instead, can add to the techniques adopted for the more general family, more specific ones directly based on the local parsability property, which may produce simpler and more efficient algorithms. If a change is applied to an input string that has been already parsed, in fact, an algorithm based on operator precedence relations can restart parsing from the factor of the string affected by the transformation and rebuild the part of the syntactic tree for the outermost handle in the string involved by the change. The remaining part of the tree, instead, cannot change and thus does not need to be processed again. A preliminary algorithm for incremental parsing is sketched in [18].

6 PARALLEL SEMANTIC ANALYSIS

Lexing and parsing are normally preliminary to a subsequent semantic phase. In the case of programming languages such semantic analysis consists typically in code interpretation or compilation and is often based on some attribute schema. If we assume, as it is sometimes the case in bottom-up compilation, that the attribute schema is of synthesized type, then, we can automatically integrate parallelism (and incrementality) for semantic analysis with parallel (and incremental) lexical and syntactic analysis. The benefits of such exploitation can be enormously extended beyond the realm of programming language compilation since most "structured" design activities can be formalized in terms of a tree-shaped syntax paired with an attribute-based semantic evaluation; furthermore, in many cases the semantic algorithms exhibit a high computational complexity so that the efficiency gained by means of parallelism (and incrementality) can become really impressive.

A preliminary description of an approach for incremental syntax-semantic analysis for software artifacts, which exploits the local parsability of OPLs, is presented, as mentioned in the previous chapter, in the recent work [18].

In this chapter we follow a complementary approach, aimed at parallel syntax-semantic processing of structured and semistructured documents, which pairs the actions of the OP parsing algorithm with a parallel evaluation of users' queries. In the sequel of this chapter we illustrate a case study of application of the approach: we adapt the OP parsing algorithm to process XPath expressions on (streaming or offline) XML documents in parallel on multicore machines. The technique can be naturally extended to deal with more complex data description languages as, e.g., JSON or domain specific languages, or more complex queries.

6.1 INTRODUCTION

Applications in several domains (e.g., financial data processing, web analytics, network monitoring) must process huge datasets or continuous high-rate streams of XML data to provide answers to queries posed by the users. Furthermore, the answers usually must be computed satisfying specific efficiency criteria.

An effective way to cope with the continuously increasing amount of information that needs to be analyzed consists in leveraging the power of modern multicore computing platforms to process large volumes of data in parallel.

OPLs provide a natural approach to address this issue, since the OP parallel parsing algorithm can be quite easily adapted to support semantic processing of an XML document, besides the mere reconstruction of its syntactic structure. The approach relies on the fact that the syntax of an XML document, specified by a DTD (Document Type Definition), can be formalized with an OPG, for which an OP parallel parser can be automatically built. Given an XPath expression that a user wants to query against a document, the OP parsing algorithm can be augmented with semantic actions that, upon reduction of sentential forms with the rules of the OPG, progressively check the segments of the path expression until a positive answer can be provided or no corresponding match is found. The parsing (and associated semantic) actions are performed in parallel by independent processors on different chunks of the XML document. The partial parsing trees generated by the different processors are, as usual, pairwise combined and reduced into the final tree, and the semantic actions for the reductions involved in these transformations are performed likewise.

The extension of the parallel OP parsing algorithm to execute XPath queries on XML documents is illustrated in Section 6.3 of this chapter, after the basic definitions and notation on DTDs and XPath are recalled in Section 6.2. Some related works on parallel XML processing are then discussed in Section 6.4.

6.2 PRELIMINARIES

In this section we remind preliminary notation on DTDs and XPath expressions.

6.2.1 *DTD*

A DTD [101] describes the structure of a class of XML documents and is composed by a list of declarations of the elements and attributes that are allowed within the documents. An element type declaration defines an element and its content. An element's content can be:

- EMPTY: the element has no content, i.e., it cannot have children elements nor text elements;
- ANY: the element has any content, i.e., it may have any number (even none) and type of children elements (including text elements). As a simplifying restriction, we forbid this type of content.
- an expression, which specifies the elements allowed as direct children. It can be:
  1. an *element content*, meaning that there cannot be text elements as children. The element content consists of a *content particle*, which can be either the name of an element declared in the DTD, or a sequence list or choice list.
     - a *sequence list* is an ordered list of one or more content particles (specified between parentheses and separated by a "," character) that appear successively as direct children in the content of the defined element;
     - a *choice list* is a mutually exclusive list of two or more content particles (specified between parentheses and separated by a "|" character).

     The element content may be followed by a quantifier ($+$, $*$ or ?), which specifies the number of successive occurrences of an item at the given position in the content of the element.
  2. a *mixed content*, meaning that the content may include at least one text element and zero or more declared elements; differently than in an element content, their order and number of occurrences are not restricted:

- (#PCDATA): the content consists exactly of one text element (PCDATA means parsed character data);
- (#PCDATA | element name | ...)∗: the content consists of a choice (in an exclusive list between parentheses and separated by "|" characters and terminated by the "*" quantifier) of two or more child elements (including only text elements or also the specified declared elements).

Example 6.1. The following list is an example of element type declarations:

$$\begin{array}{lll}
\langle !ELEMENT & note & (to, from, heading, body)\rangle \\
\langle !ELEMENT & to & (\#PCDATA)\rangle \\
\langle !ELEMENT & from & (\#PCDATA)\rangle \\
\langle !ELEMENT & heading & (\#PCDATA)\rangle \\
\langle !ELEMENT & body & (\#PCDATA)\rangle
\end{array}$$

An attribute list declaration defines the set of legal attributes for a given element, and for each attribute specifies its name, its type (or an enumeration of its possible values) and its default value.

Example 6.2. For example, an attribute list declaration for the element *note* is:

$$\begin{array}{lllll}
\langle !ATTLIST & note & & & \\
 & priority & CDATA & \#REQUIRED & \\
 & id & ID & \#IMPLIED & \\
 & forward & (yes \mid no) & & \text{“}yes''\rangle
\end{array}$$

Finally, the root element of an XML document complying with a DTD is specified by a *DOCTYPE* definition (e.g., ⟨!DOCTYPE e⟩).

The element declaration part of a DTD can be expressed as an OPG in a straightforward way. The terminal alphabet of the grammar includes the textual content $STRING$ and symbols $\langle a\rangle$, $\langle /a\rangle$, $\langle \backslash a\rangle$ for every element a of the DTD ($\langle a\rangle$ and $\langle /a\rangle$ are matching open and closed tags respectively; $\langle \backslash a\rangle$ denotes an element with empty content). Nonterminals represent the elements and their contents; the axiom is defined as the root element specified by the *DOCTYPE* definition; the productions comply with one of the following structures:

$$\begin{aligned}
&M \to STRING \\
&M \to \langle a \rangle\, R\, \langle /a \rangle \\
&M \to \langle \backslash a \rangle \\
&M \to T\ STRING \\
&M \to T\ \langle a \rangle\, R\, \langle /a \rangle \\
&M \to T\ \langle \backslash a \rangle
\end{aligned}$$

where M, R, T are nonterminals, a denotes a DTD element.

6.2.2 *XPath*

XPath (the XML Path Language) is a query language defined by the World Wide Web Consortium (W3C) [100]. The XPath language is based on a tree representation of the elements of an XML document, and provides the ability to select nodes (elements or attributes) of the document by specifying the way to navigate the tree to reach them.

A basic *XPath query* (or *expression*) is syntactically represented as a sequence of *location steps* (a path) and is evaluated with respect to a context node. Each step consists of three components: an axis, a node test and zero or more predicates. The axis specifier (such as “child” or “descendant”) denotes the direction along which the tree must be traversed from the context node. The node test and the predicates filter the nodes denoted by the axis specifier: the node test prescribes which is the label that all nodes navigated to must have, while a predicate consists of XPath expressions themselves that state some properties on these nodes.

Formally, we consider XPath expressions, P, specified by the following grammar:

$$\begin{aligned}
&P ::= /N \mid //N \mid PP \\
&N ::= E \mid @A \mid * \mid text() \mid text() = S
\end{aligned}$$

where E and A denote elements and attributes respectively and S is a string constant. $/$ denotes the child axis, $//$ denotes the descendant axis, $*$ is the wild card ($*$ matches any element node and $@*$ matches any attribute node).

As in e.g. [80], we transform XPath queries with predicates into a set of subqueries without predicates, one for each element referenced in the predicate and the parent element. As an example, the query $/a[b]/c$ is rewritten into the subqueries: $/a$, $/a/b$ and $/a/c$: the answers corresponding to the subqueries must be therefore aggregated to obtain the matches satisfying the initial predicate.

6.3 PARALLELIZATION OF SEMANTIC ANALYSIS

The OP parsing algorithm described in Chapter 5 can be naturally adapted to support parallel processing of XPath expressions against an XML document.

The algorithm is extended so that, during the OP parsing of the document, each time a reduction is performed it checks whether the tokens that have been reduced correspond to nodes that might satisfy the pattern of the XPath expression. Since the semantic processing is paired with the bottom-up OP parsing algorithm, the pattern matching against the expression is performed bottom-up too; thus initially it checks whether the reduced tokens correspond to the location step at the end of the path of the expression. If this is the case, it computes which is the rest of the path that the upper part of the parsing tree must satisfy so that the reduced tokens are actually a correct match for the query. This information is propagated to the nonterminal that is the l.h.s. of the rule used in the reduction, which will replace the r.h.s. on the stack of the parser, and the pattern matching continues thereon.

Each nonterminal that is pushed on the stack is thus associated with a set of patterns of paths, with a corresponding set of pointers to the element node or the attribute value that will be returned as output answer to the query. Note also that each (open or empty) token representing an XML tag has an associated set of attribute values.

The extension of the OP parallel parsing Algorithm 2 for XML semantic processing is presented in Algorithm 5, which illustrates the semantic actions to evaluate whenever a reduction of Algorithm 2 is performed. In the description of the algorithm, as a notational convention, let $G = (N, \Sigma, P, S)$ be the OPG derived from the DTD after it has been put in Fischer normal form, and let *exp* be the XPath expression to query. Although we have not yet evaluated our approach for parallel semantic processing through an experimental campaign, we reasonably expect to achieve performances on throughput and speedup comparable to those obtained for parallel syntactic and lexical analysis.

In the following, we detail the main phases of the algorithm.

CHECK LOCATION STEP AT THE END OF PATH (PHASE 2 AND 3) Initially the procedure checks whether the current reduction includes an XML element corresponding to the end of the path expression. If this is the case, it propagates the remaining path that need to be checked to the nonterminal that represents the l.h.s. of the rule used in the reduction (the path is conventionally ended by $//$ if the location step has a "descendant" axis specifier). Otherwise, no new path is propagated.

CHECK INTERNAL LOCATION STEPS (PHASE 4) Each time the r.h.s. of a rule is reduced, the procedure examines the paths that have been propagated up to the nonterminals of this r.h.s, so that it can verify if their descendants in the parsing tree correctly match the full path of the query. For each nonterminal X in the r.h.s the new path that will be associated to the l.h.s. is computed as follows. If the path is empty, a match for the whole XPath expression has been found. If the previous location step was specified by a "child" axis or was the end of the Xpath expression *exp* (cases 4b and 4e), then if the nonterminal X denotes an XML element b corresponding to the current location step, the path is obtained from the current one by removing this last location step; if instead the nonterminal is not surrounded by a pair of matching tags (and thus it correspond to the contents of an XML element that is still unknown at this point of the parsing), the path is propagated unchanged; otherwise this part of the tree does not match the query and no path is propagated. If the previous location step was specified by a "descendant" axis, and thus the current path by convention ends with a $//$ symbol (cases 4c and 4d), the algorithm must take care of the fact that the current node that represents the contents of a b element may not have ancestors in the tree that match the remaining path of the query, but one of its ancestors also corresponding to a b element can satisfy it. Thus, we associate to the nonterminal at the l.h.s. the path with the last step removed and the path unchanged (the unchanged path is still terminated by $//$ to remind that the same check must be done for the ancestor nodes). Note that in cases 4b and 4e, it is not necessary to propagate two paths, since we are sure that the suffix of the XPath expression *exp* from the current location step labeled b onward is satisfied.

Finally, note that, in every step 2 and 4 of the algorithm, if the path *exp* or P starts with axis $/$, we check also whether the current node is the root of the document. Also, we can check predicates specifying conditions on attributes: e.g., $q/b[@attribute\ \theta\ value]$ (or $q//b[@attribute\ \theta\ value]$, resp.), where θ is a comparison operator, is handled as q/b (or $q//b$, resp.), where the algorithm also checks that the comparison *attribute* θ *value* holds true.

Remark 6.1. Algorithm 5 can be slightly optimized. Note that if we can store the children/descendants of an element, then if the last segment of the XPath query is $(/*)^+$, we might consider the query without this suffix and return as output the children/descendants of the matching elements found for this reduced expression. The same can be done if the last segment is $//*$ or $//text()$. Furthermore, we might also possibly reduce the number of failed computations along the parsing tree by checking beforehand, given just the OPG and the XPath expression, which are the r.h.s.s of the rules that are reachable (top-down) along the path defined by the expression.

Example 6.3. As an example, consider the following DTD:

$$\begin{array}{lll}
\langle !DOCTYPE & a\rangle & \\
\langle !ELEMENT & a & (b \mid c)+\rangle \\
\langle !ELEMENT & b & (c \mid d)+\rangle \\
\langle !ELEMENT & c & (\#PCDATA)\rangle \\
\langle !ELEMENT & d & (a)+\rangle
\end{array}$$

A corresponding OPG is:

$$\begin{aligned}
&S \rightarrow \langle a \rangle\, A\, \langle /a \rangle \\
&A \rightarrow A\, \langle b \rangle\, B\, \langle /b \rangle \mid A \langle c \rangle\, C\, \langle /c \rangle \mid \langle b \rangle\, B\, \langle /b \rangle \mid \langle c \rangle\, C\, \langle /c \rangle \\
&B \rightarrow B\, \langle c \rangle\, C\, \langle /c \rangle \mid B \langle d \rangle\, D\, \langle /d \rangle \mid \langle c \rangle\, C\, \langle /c \rangle \mid \langle d \rangle\, D\, \langle /d \rangle \\
&C \rightarrow STRING \\
&D \rightarrow D\, \langle a \rangle\, A\, \langle /a \rangle \mid \langle a \rangle\, A\, \langle /a \rangle
\end{aligned}$$

An equivalent OPG in Fischer normal form has rules:

$$\begin{aligned}
N_{DS} &\rightarrow && \langle a \rangle\, A\, \langle /a \rangle \mid \\
& && \langle a \rangle\, N_{AB}\, \langle /a \rangle \\
A &\rightarrow && \langle a \rangle\, A\, \langle /a \rangle \mid \\
& && A\, \langle b \rangle\, N_{AB}\, \langle /b \rangle \mid \\
& && N_{AB}\, \langle b \rangle\, N_{AB}\, \langle /b \rangle \mid \\
& && \langle b \rangle\, B\, \langle /b \rangle \mid \\
& && A\, \langle b \rangle\, B\, \langle /b \rangle \mid \\
& && N_{AB}\, \langle b \rangle\, B\, \langle /b \rangle \mid \\
& && A\, \langle c \rangle\, C\, \langle /c \rangle \\
B &\rightarrow && B\, \langle c \rangle\, C\, \langle /c \rangle \mid \\
& && \langle d \rangle\, D\, \langle /d \rangle \mid \\
& && N_{AB}\, \langle d \rangle\, D\, \langle /d \rangle \mid \\
& && B\, \langle d \rangle\, D\, \langle /d \rangle \mid \\
& && \langle d \rangle\ \langle /d \rangle \mid \\
& && \langle d \rangle\, N_{DS}\, \langle /d \rangle \mid \\
& && N_{AB}\, \langle d \rangle\, N_{DS}\, \langle /d \rangle \mid \\
& && B\, \langle d \rangle\, N_{DS}\, \langle /d \rangle
\end{aligned}$$

$$\begin{aligned}
N_{AB} &\rightarrow \langle c \rangle\, C\, \langle /c \rangle \mid \\
&\quad N_{AB}\, \langle c \rangle\, C\, \langle /c \rangle \\
C &\rightarrow STRING \\
D &\rightarrow D\, \langle a \rangle\, A\, \langle /a \rangle \mid \\
&\quad N_{DS}\, \langle a \rangle\, A\, \langle /a \rangle \mid \\
&\quad D\, \langle a \rangle\, N_{AB}\, \langle /a \rangle \mid \\
&\quad N_{DS}\, \langle a \rangle\, N_{AB}\, \langle /a \rangle
\end{aligned}$$

Consider now the following XML document and the XPath expression $//a/b/c$:

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<a>
    <b>
        <d> </d>
    </b>
    <b>
        <c>text</c>
        <d>
            <a>
                <b>
                    <c>text</c>
                    <c>text</c>
                </b>
            </a>
        </d>
    </b>
    <c>text</c>
</a>
```

By applying the extended OP parsing algorithm, we obtain that the nodes that match the path of the query are the first, second and third element c.

6.4 RELATED WORK

XML parallel processing has been the object of various research works: a quite detailed bibliography on related works on XML parallel parsing and querying is presented, e.g., in [80].

A key property of the parallel OP parsing algorithm consists in the possibility of arbitrarily splitting an input string into equal-sized chunks, avoiding to cause load imbalance on CPU cores in the presence of documents with an irregular structure. Most of the early literature on XML parallel processing, instead, relies on the availability of well-formed fragments of the XML document, which are queried in parallel. E.g., [67] proposes a parallel structured join algorithm for evaluating XPath expressions, which partitions the elements of the XML document into buckets and examines each bucket in parallel, joining the results afterwards. A downside of the approach, however, is that, although the evaluation is parallelizable, the partition must be obtained through a sequential preprocessing step, which introduces a bottleneck in the execution. [44] exploits data parallelism for query processing using the map-reduce model, but assumes well-formed XML fragments as input to the map operation, leading to a sequential bottleneck likewise.

The same drawback affects some other techniques, which parallelize the evaluation of an XPath expression after building a parse tree in memory. A simple approach is presented in [21], where the expression is rewritten into several, easier to process, subqueries that are evaluated in parallel. To exploit data parallelism efficiently, the parse tree must be partitioned, again with a sequential preprocessing step. Since all these approaches assume that the parse tree is represented in memory, they are furthermore unfeasible to support processing of streaming XML data.

A different approach w.r.t. these early works is presented in [80]. They introduce a data-parallel approach for processing streaming XPath expressions where the execution of the query is modeled by a Visibly Pushdown Automaton (VPA): the XML document to be examined can be split into arbitrarily-sized chunks, with each chunk being processed by a VPA in parallel. To cope with the possible presence of not necessarily well-formed chunks, the automata consider all possible starting states for XML elements and construct corresponding mappings from starting to ending states. The state mappings for each chunk are built in parallel on independent CPU cores, thanks to the fact that the mappings do not depend on the state preceding the starting one; the mappings constructed for the chunks are then unified. The approach presented in [80] is thus analogous, in this respect, with processing based on OPLs: they can both operate on arbitrarily framed XML chunks, thus avoiding the cost of a sequential preprocessing phase to split the data into well formed units. We have not yet validated our approach with an experimental campaign; thus, we cannot compare the performance of the two approaches, although we reasonably expect to achieve results analogous to those showed in Chapter 5 for simple parsing and lexing. However, we note that the automata-based XML processing of [80] shows some redundancy by considering complete VPAs and taking into account unnecessary computations while building the state mapping. Furthermore, XML processing based on OPLs naturally allows for both the evaluation of the XPath expression and schema validation on the XML document. We also emphasize that the approach in [80] is supported by a C++ prototype implementation that, as the authors claim, is able to achieve near linear scaling up to 64 CPU cores while processing XPath queries taken from standard benchmarks on XML stream data. However, the implementation is not available, whereas our tool PAPAGENO is an open source project available under GNU General Public License and we plan to make available likewise also the extension for XML parallel processing. Moreover, being based on the model of VPAs, it presents some limitations as regards the syntax of the data representation languages it can deal with; we plan, instead, to define algorithms for efficient parallel query processing that exploit the properties of OPLs to support more complex standard data format to represent data (e.g., besides XML and JSON, also YAML [2] or domain specific languages).

Algorithm 5 : pathMatching$(rule, exp)$

```
1. Let $M \rightarrow \gamma$ be the production rule used in the current reduction, with $\gamma \in V^*$.
2. If $\gamma = \alpha\langle b\rangle X\langle /b\rangle$ or $\gamma = \alpha\langle\backslash b\rangle$ with $\alpha \in N \cup \{\varepsilon\}$, $X \in N$
    • If $exp = q/b$ (or $q/*$ or $q//*$) where $q$ is any path, possibly empty, then associate with nonterminal $M$ on the stack the path $q$
    • If $exp = q//b$ where $q$ is any path, possibly empty, then associate with nonterminal $M$ on the stack the path $q//$
    • If $exp = q/b/@attribute$ (or $exp = q//b/@attribute$, resp.) where $q$ is any path, possibly empty, and the tag $b$ on the stack has an attribute $attribute$, then associate with nonterminal $M$ on the stack the path $q$ (or $q//$, resp.).
3. Else if $\gamma = \alpha\ STRING$ with $\alpha \in N \cup \{\varepsilon\}$
    • If $exp = q/text()$ (or $exp = q//text()$ resp.) where $q$ is any path, then associate with nonterminal $M$ on the stack the path $q$ (or $q//$, resp.). If $exp = q/text(S)$ or $exp = q//text(S)$, check also that $STRING = S$.
4. For each nonterminal $X$ occurring in $\gamma$ do
    • For each path pattern $P$ associated with $X$ do
        a) If $P = \varepsilon$ or $P = \varepsilon//$, then return as output that the nodes matched the full path of $exp$. Furthermore remove, from all other path patterns of $X$, the node pointers already in the set associated with $P$ (note that, if a path pattern has an empty set of pointers, it is removed too).
        b) If $P = q/b$ (or $P = q//b$, resp.) where $q$ is any path
            – If $\gamma = \alpha\langle b\rangle X\langle /b\rangle$ then associate with nonterminal $M$ on the stack the path $q$ (or $q//$, resp.)
            – else if $\gamma = X\beta$ with $\beta \in V^*$, i.e., $X$ is not surrounded by a pair of matching tags $\langle b\rangle\langle /b\rangle$ nor by a pair of matching tags $\langle c\rangle X\langle /c\rangle$ with $c \neq b$, then associate with $M$ the path $q/b$ (or $q//b$, resp.). Thus, if $\gamma = \alpha\langle c\rangle X\langle /c\rangle$ with $c \neq b$, no path is propagated to $M$.
        c) If $P = q/b//$
            – If $\gamma = \alpha\langle b\rangle X\langle /b\rangle$ then associate with nonterminal $M$ on the stack both the path $q$ and the path $q/b//$
            – otherwise, associate with $M$ the path $q/b//$
        d) If $P = q//b//$
            – If $\gamma = \alpha\langle b\rangle X\langle /b\rangle$ then associate with nonterminal $M$ on the stack the path $q//$
            – otherwise, associate with $M$ the path $q//b//$
        e) If $P = q/b/@attribute$ (or $q//b/@attribute$) idem as in case 4b, but we also check that the tag $b$ has an attribute $attribute$
        f) If $P = q/*$ (or $q//*$, resp.) If $\gamma = \alpha\langle b\rangle X\langle /b\rangle$ for a tag $b$ in $\Sigma$ then associate with nonterminal $M$ on the stack the path $q$ ($q//$, resp.) otherwise, associate with $M$ the path $q/*$ ($q//*$, resp.)
```

Part IV.

Conclusions

7 CONCLUSIONS

7.1 CONCLUSIONS AND FUTURE RESEARCH

This thesis has investigated the formalism of Operator Precedence Languages (OPLs), which has been introduced in the 1960s by Robert Floyd and soon abandoned due to the advent of other language families well suited for deterministic parsing and with greater expressive power. Study on OPLs has been recently resumed with novel motivation with the discovery of some distinguishing properties they enjoy, which naturally foster their application in contexts as automatic verification (model-checking) and efficient parallel and incremental analysis.

The first part of this dissertation follows this renewed path of investigation with the aim of providing a "general theory of OPLs", which completes the results on this class of languages that have been proved in the last half a century, concerning their characterization by logical formalisms or by families of generating or recognizing devices and their algebraic properties, which qualify them as the largest class of deterministic context-free languages enjoying closure under all main language operations (Boolean ones, concatenation, Kleene * and others). Herein we extended research on OPLs to the field of ω-languages, i.e., languages consisting of infinite-length strings: we introduced a corresponding automata and logic-based characterization and we proved that all the above properties continue to hold, with the noticeable and typical exception of the lack of equivalence between deterministic and nondeterministic automata – under the Büchi acceptance condition.

As a result, OPLs are now completely characterized in terms of their algebraic properties, their recognizing automata, both on finite and infinite strings, and the logical expression of their properties in terms of a suitable MSO logic. In particular, the closure properties and the decidability of the emptiness problem for this class of languages enable the definition of model-checking algorithms outside the traditional scope of finite-state machines, allowing thus for more expressiveness in the specification of a system of interest (with finite or infinite computations) or in stating requirements on its behavior w.r.t. renowned classes of formalisms as, e.g., VPLs, which represent a strict subclass of OPLs. Furthermore, we investigated logic formalisms simpler than MSO logic to characterize suitable subclasses of general OPLs; a first result on this respect is that free languages, a subclass of OPLs originally motivated by grammar inference [35, 36] can be defined in terms of a first-order (FO) logic rather than a second-order one.

This fairly complete foundational characterization of OPLs can ignite further research along various directions: for instance, a subject for future work is the definition of a new subclass of OPLs that perfectly matches the FO logic formulation; also, moving from traditional FO or MSO logics to temporal logics that avoid explicit quantification –a widely adopted approach in various model-checking-based applications [31, 8]– and the study of corresponding model-checking algorithms are topics that belong to future research.

The second part of this dissertation deals with a further relevant property exhibited by OPLs, namely their local parsability. Local parsability means that parsing of any substring of a string according to a grammar depends only on information that can be obtained from a local analysis of the portion of the substring under processing and is thus not influenced by parsing of other substrings.

Local parsability is the key property to exploit effectively non-speculative parallelism in parsing, which was up to now a considerable exception to the present tendency to exploit parallelism in practically any application [14]. The classical parsing algorithms used for deterministic context-free (DCF) languages such as LR and LL, in fact, do not enjoy this property and, although they can be efficiently implemented (in linear-time) on sequential machines, do not achieve speedups on multicore architectures due to their inherent sequential nature: if an input string is split into several parts, handled by different processors, the parsing actions may require communication among the different processing nodes, with considerable additional overhead.

This thesis resumes the approach for OP parallel parsing, which has been introduced in [14] by exploiting the local parsability property, and it complements this approach with a schema for parallelizing also the lexical analysis phase. The algorithms for parallel parsing [15] and lexing have been implemented in a prototype tool (PAPAGENO), which we show, through an extensive experimental campaign, to achieve significant speed-up compared with state of the art sequential parsers and lexers generated by, e.g., Bison and Flex, as regards the analysis of both general programming languages and standard data representation languages. Notably, PAPAGENO is available for free downloading (under GNU GPL license) for further application, experimentation, and possible extensions. To validate the approach we chose practical test-benches spanning from the JavaScript Object Notation (JSON) data description language to the Lua programming language, showing how the minor theoretical limitations in terms of generative power of OPGs (w.r.t. DCF languages) do not significantly affect their applicability; also, the changes needed to adapt the original BNF of the source language to OPG constraints are obtained in an original way by augmenting the parallel initial phase of lexical analysis.

This dissertation exploits the local parsability property enjoyed by OPLs also for efficient parallel querying of large structured and semistructured documents. As a case study, we augmented the parallel OP parsing algorithm to allow for processing of XPath queries on XML documents in parallel on multicore machines. As a future research, we plan to validate the approach by considering, besides JSON, more complex data description languages as, e.g., YAML or domain specific languages that can be expressed as an OPL. We also plan to extend the algorithm to deal with more general users' expressions, which require to specify additional record-selection criteria or to transform the result set.

In the following, we mention various research directions to fully exploit the local parsability property; we first give a couple of hints to make our tool PAPAGENO even more efficient and more widely applicable to a larger set of languages. Then, we outline other research directions aimed at exploiting local parsability, both of theoretical nature and with the purpose of applying it in other application fields than "pure" compilation.

7.1.1 *Optimizing performances and effectiveness of the present tool*

The general parallel parsing algorithmic schema defined in Section 5.2 is conceived in such a way that it can be iterated through several passes until the obtained result is short enough to make it convenient to apply a final sequential parsing. As a matter of fact, our present tool obtained quite satisfactory performances even with one only parallel pass immediately followed by a final sequential one. As we already noticed, however, there could be cases where splitting the input in chunks of equal length does not correspond to the overall structure of the source, e.g., if the chunks consist of the frontiers of two adjacent and large sub-trees. In such cases the further passes described in the general Algorithm 2 may produce substantial benefits. Some further experiments are planned to validate our conjecture that in most cases two passes are all that is needed to exploit at best parallel parsing.

Deterministic parsing based on OPGs has been applied to many programming languages in the past [47, 39] and we added a couple of more recent ones used in our benchmark; purposely, JSON and Lua have been selected with sharply different features and, not surprisingly, adapting Lua to our OP-based approach was a considerably tougher job than the former one. Our further investigations, e.g., on JavaScript, seems to hint at some more difficulties with other modern languages: in general, in fact, as pointed out in Section 5.4, many modern languages offer (too?) much freedom to programmers which generates overloading of various symbols and hampers deterministic analysis, what often results into precedence conflicts. We are confident that such difficulties can be overcome but some more work is needed to make our approach applicable to most practical languages in a generalized way. As a first step we plan to widen the heuristic techniques developed in the case of Lua so as to apply them to other widely used languages. To be useful in practice, such techniques must be supported by automatic tools to obtain an OPG grammar equivalent to the original one and/or to produce an intermediate text –after parallel lexical analysis– that can be supplied to an OP parallel parser. In the longer term we wish to investigate also the theoretical aspects of this issue as hinted in the following subsection.

On the other hand we notice that recent recommended best practices tend to limit the excessive freedom allowed by the grammars of modern languages. For instance

[38] proposes a series of disciplined ways to write cleaner and more understandable JavaScript programs; we verified that, if applied rigorously, they produce a subset language almost ready for OP-parsing with no need for a heavy preprocessing. Another case of well disciplined, and easily analyzable source is provided by modern compiler back-ends, which target a restriction of the JavaScript language as their assembly output such as Emscripten [4] and asm.js [93]. In both cases, the language to be analyzed has all the features to be efficiently parsed in parallel: it is characterized by very large compilation units, and has a grammar free from unusual quirks.

7.1.2 *Research directions on the local parsability property*

The local parsability property proposes several intriguing questions which further widen the spectrum of potential applications both within and beyond parallelism.

From a theoretical point of view, OPLs are just an example of locally parsable languages with bound 1 but many more could be worth investigating: as mentioned in Section 5.7, Floyd himself proposed a generalization of the family by adopting larger bounds of the context necessary to disambiguate the r.h.s. to be reduced. At the time it was concluded that the approach was unpractical for complexity reasons, but the computational power available nowadays could question that early decision. Given that most parsability properties of CF languages are in general undecidable (see e.g. [62, 53] for a summary of such results) and that it is even undecidable whether the language generated by a CF grammar is OPL [57], a few natural questions arise.

- To which languages is it possible to apply a preprocessing in the same style as we did for Lua in Section 5.4, so that the obtained intermediate text is an OPL?
- Furthermore, is it possible and useful to extend our proposed approach to locally parsable languages beyond the OPLs?

The local parsability property can be exploited also in further ways besides parallel parsing.

As stated in Section 5.7, a promising field of study is represented by incremental parsing, to deal with continuous changes and evolution of software artifacts. The local parsability property can be exploited in conjunction for *incremental and parallel* parsing in the non-infrequent case of multiple, scattered changes to large pieces of software.

Finally, (syntactic) error management can also take advantage of the local parsability property. In many cases, in fact, a syntax error may affect an unpredictable portion of code, and it is often the case that, at their early occurrence, parsing is stopped or becomes meaningless (e.g., the standard parsers generated by Bison stop their processing at the first error). The local parsability property instead, allowing for (re)starting parsing at any position, may produce, possibly by acting in parallel, large portions of syntax tree associated with correct code, even if such code is preceded by serious errors. Thus, breaking the code into many chunks can help not only to locate the source of the problem but also to fix it without redoing much useless work, by exploiting both parallelism and incrementality. Similar, though less relevant, benefits could be obtained by exploiting parallel lexical analysis.

BIBLIOGRAPHY

[1] Flup. `https://github.com/bzoto/flup/`.

[2] YAML. `http://www.yaml.org/`.

[3] A. Demaille, et al . GNU Bison. `http://www.gnu.org/software/bison/`, 2014.

[4] A. Zakai. Emscripten. `https://github.com/kripken/emscripten/wiki`, 2014.

[5] L. Afanasiev, P. Blackburn, I. Dimitriou, B. Gaiffe, E. Goris, M. Marx, and M. de Rijke. PDL for ordered trees. *Journal of Applied Non-Classical Logic*, 15(2):115–135, 2005.

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, second edition, 2007.

[7] H. Alblas, R. op den Akker, P. O. Luttighuis, and K. Sikkel. A bibliography on parallel parsing. *SIGPLAN Notices*, 29(1):54–65, 1994.

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

[9] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer Berlin Heidelberg, 2004.

[10] R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 2004.

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

[12] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, F. Panella, and M. Pradella. The PAPAGENO parallel-parser generator. In *Compiler Construction - 23rd International Conference, CC*, pages 192–196, 2014.

[13] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, F. Panella, and M. Pradella. Parallel parsing made practical. *Science of Computer Programming*, 112, Part 3:195 – 226, 2015.

[14] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. Parallel parsing of operator precedence grammars. *Information Processing Letters*, 2013. DOI:10.1016/j.ipl.2013.01.008.

[15] A. Barenghi, E. Viviani, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. PAPAGENO: a parallel parser generator for operator precedence grammars. In *5th International Conference on Software Language Engineering (SLE)*, 2012.

[16] J. Berstel and L. Boasson. Balanced Grammars and Their Languages. In W. B. et al., editor, *Formal and Natural Computing*, volume 2300 of *LNCS*, pages 3–25. Springer, 2002.

[17] J. Berstel and L. Boasson. Balanced grammars and their languages. In W. B. et al., editor, *Formal and Natural Computing*, volume 2300 of *LNCS*, pages 3–25. Springer, 2002.

[18] D. Bianculli, A. Filieri, C. Ghezzi, and D. Mandrioli. Syntactic-semantic incrementality for agile verification. *Science of Computer Programming*, 2013. DOI:10.1016/j.scico.2013.11.026.

[19] L. Boasson and M. Nivat. Adherences of languages. *Journal of Computer and System Sciences*, 20(3):285 – 309, 1980.

[20] A. Boral and S. Schmitz. Model-checking parse trees. In *LICS*, 2013.

[21] R. Bordawekar, L. Lim, A. Kementsietsidis, and B. W. Kok. Statistics-based parallelization of XPath queries in shared memory systems. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 159–170. ACM, 2010.

[22] W. S. Brainerd. The minimalization of tree automata. *Information and Control*, 13(5):484–491, 1968.

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

[24] J. R. Büchi. On a decision method in restricted second order arithmetic. In E. Nagel, P. Suppes, and A. Tarski, editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, pages 1–11. Stanford University Press, 1962.

[25] O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR '92*, volume 630 of *LNCS*, pages 123–137. 1992.

[26] D. Caucal and S. Hassen. Synchronization of Grammars. In E. A. Hirsch, A. A. Razborov, A. L. Semenov, and A. Slissenko, editors, *CSR*, volume 5010 of *LNCS*, pages 110–121. Springer, 2008.

[27] Celestia Development Team. The Celestia Space Simulation. `http://sourceforge.net/projects/celestia/`, 2014.

[28] C. Choffrut, A. Malcher, C. Mereghetti, and B. Palano. On the Expressive Power of FO[+]. In *LATA*, pages 190–201, 2010.

[29] C. Choffrut, A. Malcher, C. Mereghetti, and B. Palano. First-order logics: some characterizations and closure properties. *Acta Inf.*, 49(4):225–248, 2012.

[30] M. Chytil, M. Crochemore, B. Monien, and W. Rytter. On the parallel recognition of unambiguous context-free languages. *Theoretical Computer Science*, 81(2):311–316, 30 Apr. 1991. Note.

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

[32] J. Cohen, T. Hickey, and J. Katcoff. Upper bounds for speedup in parallel parsing. *Journal of the ACM*, 29(2):408–428, Apr. 1982.

[33] S. Crespi Reghizzi and D. Mandrioli. A Class of Grammar Generating Non-Counting Languages. *Inf. Process. Lett.*, 7(1):24–26, 1978.

[34] S. Crespi Reghizzi and D. Mandrioli. Operator Precedence and the Visibly Pushdown Property. *Journal of Computer and System Science*, 78(6):1837–1867, 2012.

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

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

[37] D. Crockford. RFC4267 - The application/json Media Type for JavaScript Object Notation (JSON). `http://www.ietf.org/rfc/rfc4627.txt`, 2006.

[38] D. Crockford. *JavaScript - the good parts: unearthing the excellence in JavaScript*. O'Reilly, 2008.

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

[40] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[41] P. Degano, S. Mannucci, and B. Mojana. Efficient incremental LR parsing for syntax-directed editors. *ACM Trans. Program. Lang. Syst.*, 10(3):345–373, July 1988.

[42] M. DeLoura. The engine survey: General results. `http://www.satori.org/2009/03/the-engine-survey-general-results/`, March 2009. [Online; accessed 5 December 2013].

[43] A. D'Ulizia, F. Ferri, and P. Grifoni. A survey of grammatical inference methods for natural language learning. *Artif. Intell. Rev.*, 36(1):1–27, 2011.

[44] L. Fegaras, C. Li, U. Gupta, and J. Philip. Xml query optimization in map-reduce. In *WebDB*, 2011.

[45] C. N. Fischer. On parsing context free languages in parallel environments. Technical report, Cornell University, Apr. 1975.

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

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

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

[49] Game Developer. 14th Annual Front Line Awards, January 2012.

[50] U. Germann, E. Joanis, and S. Larkin. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 31–39, 2009.

[51] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, 1979.

[52] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Information and Computation*, 81(1):32–45, Apr. 1989.

[53] S. Greibach. A note on undecidable properties of formal languages. *Mathematical systems theory*, 2(1):1–6, 1968.

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

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

[56] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. of the ACM*, 29(12):1170–1183, 1986.

[57] H. B. Hunt and D. J. Rosenkrantz. Computational parallels between the regular and context-free languages. *SIAM J. Comput.*, 7(1):99–114, 1978.

[58] H. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, Los Angeles (California), 1968.

[59] A. H. Karp and H. P. Flatt. Measuring parallel processor performance. *Comm. ACM*, 33(5):539–543, 1990.

[60] J. Kegler. Perl and undecidability. *The Perl Review*, 2008.

[61] P. N. Klein and J. H. Reif. Parallel time O(log n) acceptance of deterministic CFLs on an exclusive-write P-RAM. *SICOMP: SIAM Journal on Computing*, 17, 1988.

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

[63] J. Lampe. MATE - A metasystem with concurrent attribute evaluation (abstract). In D. K. Hammer, editor, *CC*, volume 477 of *Lecture Notes in Computer Science*, pages 222–223. Springer, 1990.

[64] J. Larchevêque. Optimal incremental parsing. *ACM TOPLAS*, 17(1):1–15, Jan. 1995.

[65] C. Lautemann, T. Schwentick, and D. Thérien. Logics for context-free languages. In *Selected Papers from the 8th International Workshop on Computer Science Logic*, CSL '94, pages 205–216, London, UK, 1995. Springer-Verlag.

[66] Layer7 Technology. XML Accelerator. `http://www.layer7tech.com/products/xml-accelerator`, 2014.

[67] L. Liu, J. Feng, G. Li, Q. Qian, and J. Li. Parallel structural join algorithm on shared-memory multi-core systems. In *Web-Age Information Management, 2008. WAIM'08. The Ninth International Conference on*, pages 70–77. IEEE, 2008.

[68] V. Lonati, D. Mandrioli, F. Panella, and M. Pradella. First-order Logic Definition of Free Languages. In *10th Int. Computer Science Symposium in Russia (CSR)*, volume 9139 of *LNCS*. Springer, 2015.

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

[70] V. Lonati, D. Mandrioli, and M. Pradella. Precedence Automata and Languages. In *6th Int. Computer Science Symposium in Russia (CSR)*, volume 6651 of *LNCS*, pages 291–304. Springer, 2011.

[71] V. Lonati, D. Mandrioli, and M. Pradella. Logic Characterization of Invisibly Structured Languages: the Case of Floyd Languages. In *39th Int. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 7741 of *LNCS*, pages 307–318. Springer, 2013.

[72] W. Lu, K. Chiu, and Y. Pan. A parallel approach to XML parsing. In *7th IEEE/ACM International Conference on Grid Computing (GRID 2006), September 28-29, 2006, Barcelona, Spain, Proceedings*, pages 223–230, 2006.

[73] R. McCloskey, J. Wang, and J. Belanger. Parallel parsing of languages generated by ambiguous bounded context grammars, Mar. 18 1994.

[74] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9(6):521–530, 1966.

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

[76] R. McNaughton and S. Papert. *Counter-free Automata*. MIT Press, Cambridge, USA, 1971.

[77] M. D. Mickunas and R. M. Schell. Parallel compilation in A multiprocessor environment (extended abstract). In *Proceedings 1978 ACM Annual Conference, Washington, DC, USA, December 4-6, 1978, Volume I*, pages 241–246, 1978.

[78] D. E. Muller. Infinite sequences and finite machines. In *Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '63, pages 3–16, Washington, DC, USA, 1963. IEEE Computer Society.

[79] D. Nowotka and J. Srba. Height-Deterministic Pushdown Automata. In L. Kucera and A. Kucera, editors, *MFCS 2007, Ceský Krumlov, Czech Republic, August 26-31, 2007, Proceedings*, volume 4708 of *LNCS*, pages 125–134. Springer, 2007.

[80] P. Ogden, D. Thomas, and P. Pietzuch. Scalable xml query processing using parallel pushdown transducers. *Proceedings of the VLDB Endowment*, 6(14):1738–1749, 2013.

[81] F. Panella, M. Pradella, V. Lonati, and D. Mandrioli. Operator precedence ω-languages. In *17th International Conference on Developments in Language Theory (DLT)*, volume 7907 of *LNCS*, pages 396–408. 2013.

[82] F. Panella, M. Pradella, V. Lonati, and D. Mandrioli. Operator precedence ω-languages. *CoRR*, abs/1301.2476, 2013. http://arxiv.org/abs/1301.2476.

[83] Papageno Developers. PAPAGENO: the parallel parser generator for operator precedence grammars. `https://github.com/PAPAGENO-devels/papageno`, 2014.

[84] D. Perrin and J.-E. Pin. *Infinite words*, volume 141 of *Pure and Applied Mathematics*. Elsevier, Amsterdam, 2004.

[85] Pontifical Catholic University of Rio de Janeiro. Lua official reference manual. `http://www.lua.org/manual/5.2/`, 2014.

[86] M. Rabin. *Automata on infinite objects and Church's problem*. Regional conference series in mathematics. Published for the Conference Board of the Mathematical Sciences by the American Mathematical Society, 1972.

[87] W. Rytter. On the complexity of parallel parsing of general context-free languages. *Theoretical Computer Science*, 47(3):315–321, 1986. Note.

[88] A. K. Salomaa. *Formal Languages*. Academic Press, New York, NY, 1973.

[89] D. Sarkar and N. Deo. Estimating the speedup in parallel parsing. *IEEE Trans. on Softw. Eng.*, 16(7):677, 1990.

[90] R. Sin'ya, K. Matsuzaki, and M. Sassa. Simultaneous finite automata: An efficient data-parallel model for regular expression matching. In *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, pages 220–229, 2013.

[91] G. U. Srikanth. Parallel lexical analyzer on the cell processor. In *SSIRI (Companion)*, pages 28–29, 2010.

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

[93] The Mozilla Foundation. Asm.js. `http://asmjs.org/`, 2014.

[94] The Wikimedia Foundation. Wikipedia adopts Lua as its default scripting language. `http://en.wikipedia.org/wiki/Wikipedia:Wikipedia_Signpost/2012-01-30/Technology_report`, 2012.

[95] W. Thomas. A combinatorial approach to the theory of omega-automata. *Information and Control*, 48(3):261 – 283, 1981.

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

[97] L. Vagner and B. Melichar. Parallel LL parsing. *Acta Inf*, 44(1):1–21, 2007.

[98] B. von Braunmühl and 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*, pages 40–51. Springer, 1983.

[99] W. Thomas. Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25(3):360 – 376, 1982.

[100] World Wide Web Consortium (W3C. XML Path Language (XPath) 2.0 (Second Edition) W3C Recommendation. `http://www.w3.org/TR/2010/REC-xpath20-20101214`, 14 December 2010.

[101] World Wide Web Consortium (W3C. Extensible Markup Language (XML) 1.1 (Second Edition) W3C Recommendation. `http://www.w3.org/TR/2006/REC-xml11-20060816`, 16 August 2006.

[102] Z. Zhao, M. Bebenita, D. Herman, J. Sun, and X. Shen. HPar: a practical parallel parser for HTML — taming HTML complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization*, 10(4), Dec. 2013.

APPENDIX

Herein we report Lua's syntactic grammar in operator precedence form.

chunk	→	block | ENDFILE
block	→	statList |
		retStat |
		statList RETURN SEMI |
		statList RETURN exprList SEMI |
		statList RETURN |
		statList RETURN exprList
statList	→	stat |
		SEMI |
		stat SEMI |
		statList SEMI stat |
		statList SEMI
retStat	→	RETURN SEMI |
		RETURN exprList SEMI |
		RETURN |
		RETURN exprList
stat	→	varList XEQ exprList |
		functionCall |
		label |
		BREAK |
		GOTO NAME |
		DO block END |
		DO END |
		WHILE expr DO block END |
		WHILE expr DO END |

```
REPEAT block UNTIL expr |
REPEAT UNTIL expr |
IF exprThen END  |
IF exprThen ELSE block END |
IF exprThen ELSE END |
IF exprThenElseIfB END |
IF exprThenElseIfB ELSE block END |
IF exprThenElseIfB ELSE END  |
FOR name XEQ eCe DO block END |
FOR name XEQ eCeCe DO block END |
FOR nameList IN exprList DO block END |
FUNCTION funcName LPARENFUNC parList RPARENFUNC block END |
FUNCTION funcName LPARENFUNC RPARENFUNC block END |
FOR name XEQ eCe DO END |
FOR name XEQ eCeCe DO END |
FOR nameList IN exprList DO END |
FUNCTION funcName LPARENFUNC parList RPARENFUNC END |
FUNCTION funcName LPARENFUNC RPARENFUNC END |
LOCAL FUNCTION name LPARENFUNC parList RPARENFUNC block END |
LOCAL FUNCTION name LPARENFUNC RPARENFUNC block END |
LOCAL FUNCTION name LPARENFUNC parList RPARENFUNC END |
LOCAL FUNCTION name LPARENFUNC RPARENFUNC END  |
LOCAL nameList |
LOCAL nameList XEQ exprList
```

elseIfBlock	→	block ELSEIF expr THEN block |
		block ELSEIF expr THEN elseIfBlock |
		ELSEIF expr THEN block |
		block ELSEIF expr THEN |
		ELSEIF expr THEN |
		ELSEIF expr THEN elseIfBlock
exprThenElseIfB	→	expr THEN elseIfBlock
exprThen	→	expr THEN block |
		expr THEN
name	→	NAME
eCe	→	expr COMMA expr
eCeCe	→	eCe COMMA expr
dot3	→	DOT3
label	→	COLON2 NAME COLON2
funcName	→	nameDotList |
		nameDotList COLON name
nameDotList	→	NAME |
		nameDotList DOT NAME
varList	→	var |
		varList COMMA var
var	→	NAME |
		prefixExp LBRACK expr RBRACK |
		prefixExp DOT NAME
nameList	→	NAME |
		nameList COMMA name
exprList	→	expr |
		exprList COMMA expr
expr	→	logicalOrExp
logicalOrExp	→	logicalAndExp |
		logicalOrExp OR logicalAndExp

logicalAndExp	→	relationalExp |
		logicalAndExp AND relationalExp
relationalExp	→	concatExp |
		relationalExp LT concatExp |
		relationalExp GT concatExp |
		relationalExp LTEQ concatExp |
		relationalExp GTEQ concatExp |
		relationalExp NEQ concatExp |
		relationalExp EQ2 concatExp
concatExp	→	additiveExp |
		additiveExp DOT2 concatExp
additiveExp	→	multiplicativeExp |
		additiveExp PLUS multiplicativeExp |
		additiveExp MINUS multiplicativeExp
multiplicativeExp	→	unaryExp |
		multiplicativeExp ASTERISK unaryExp |
		multiplicativeExp DIVIDE unaryExp |
		multiplicativeExp PERCENT unaryExp
unaryExp	→	caretExp |
		NOT unaryExp |
		SHARP unaryExp |
		UMINUS unaryExp
caretExp	→	baseExp |
		baseExp CARET caretExp
baseExp	→	NIL | FALSE | TRUE | NUMBER |
		STRING |
		DOT3 |
		functionDef |
		prefixExp |
		tableConstructor
prefixExp	→	var |
		functionCall |
		LPAREN expr RPAREN

```
functionCall      →  prefixExp LPAREN exprList RPAREN |
                     prefixExp LPAREN RPAREN |
                     prefixExp LBRACE fieldList RBRACE |
                     prefixExp LBRACE RBRACE |
                     prefixExp STRING |
                     prefixExp COLON name LPAREN exprList RPAREN |
                     prefixExp COLON name LPAREN RPAREN |
                     prefixExp COLON name LBRACE fieldList RBRACE |
                     prefixExp COLON name LBRACE RBRACE |
                     prefixExp COLON name STRING
functionDef       →  FUNCTION LPARENFUNC parList RPARENFUNC block END |
                     FUNCTION LPARENFUNC RPARENFUNC block END |
                     FUNCTION LPARENFUNC parList RPARENFUNC END |
                     FUNCTION LPARENFUNC RPARENFUNC END
parList           →  nameList | nameList COMMA dot3 | DOT3
tableConstructor  →  LBRACE fieldList RBRACE |
                     LBRACE RBRACE
fieldList         →  fieldListBody | fieldListBody COMMA | fieldListBody SEMIFIELD
fieldListBody     →  field | fieldListBody COMMA field | fieldListBody SEMIFIELD field
field             →  bracketedExp EQ expr | name EQ expr | expr
bracketedExp      →  LBRACK expr RBRACK
```