Conjunctive and Boolean grammars: the true general case of the context-free grammars[☆]

Alexander Okhotin[1]

Department of Mathematics and Statistics, University of Turku, Turku FI–20014, Finland

Abstract

Conjunctive grammars extend the definition of a context-free grammar by allowing a conjunction operation in the rules; Boolean grammars are further equipped with an explicit negation. These grammars maintain the main principle of the context-free grammars, that of defining syntactically correct strings inductively from their substrings, but lift the restriction of using disjunction only. This paper surveys the results on conjunctive and Boolean grammars obtained over the last decade, comparing them to the corresponding results for ordinary context-free grammars and their main subfamilies. Much attention is given to parsing algorithms, most of which are inherited from the case of ordinary context-free grammars without increasing their computational complexity. The intended readership includes any computer scientists looking for a compact and accessible description of this formal model and its properties, as well as for a general outlook on formal grammars. The paper is also addressed to theoretical computer scientists seeking a subject for research; an account of pure theoretical research in the area presented in this paper is accompanied by a list of significant open problems, with an award offered for the first correct solution of each problem. Several directions for future investigation are proposed.

Key words: Formal languages, context-free grammars, conjunctive grammars, Boolean grammars, parsing, language equations

Contents

[☆]This paper supersedes the earlier surveys, "An overview of conjunctive grammars" (*Bulletin of the EATCS*, 2004) and "Nine open problems for conjunctive and Boolean grammars" (*Bulletin of the EATCS*, 2007).

Email addresses: alexander.okhotin@utu.fi (Alexander Okhotin)

[1]Supported by the Academy of Finland under grants 134860 and 257857.

1. Introduction

Syntax of languages, both natural and artificial, is defined inductively, in the sense that syntactic properties of strings of symbols are logically determined by the properties of their substrings. A grammar of a particular language gives names to these syntactic properties, and explains, how shorter strings with certain properties can be concatenated to obtain longer strings with another property. For instance, a rule of a hypothetical grammar for a natural language may say that a *subject* followed by a *predicate* is a *sentence*; the form of a subject and a predicate is defined by other rules of the grammar. Similarly, a grammar for a programming language may define a *loop statement* as a keyword `while` followed by an *expression* and a *statement* (where the latter may, in particular, be another loop statement). An abstract language $\{a^n b^n \,|\, n \geqslant 0\}$ over an alphabet $\{a, b\}$ is completely defined by saying that a string belongs to this language if and only if it is either an empty sequence of symbols, or a string of the form awb, where w is a string belonging to this language.

Definitions of this kind are naturally given when the syntax of any language needs to be clearly described, such as in the textbook on the English grammar by Reed and Kellogg [121] or in the first description of the Algol programming language [119]. Following Chomsky's [15] influential works, such definitions became known as *context-free grammars*, reflecting the fact that syntactic properties of a substring do not depend on the context, in which it occurs. The form of the rules was fixed to

$$A \to X_1 \dots X_\ell, \qquad (*)$$

where the symbol A represents a syntactic notion defined in the grammar, such as a *sentence* or a *loop statement* (these auxiliary notions are called "nonterminal symbols" for historical reasons), and each symbol X_i may be another nonterminal symbol or a symbol of the target alphabet. A rule (*) means that every string representable as a concatenation $X_1 \dots X_\ell$ therefore has the property A. Returning to the above examples, the hypothetical grammar for a natural language has a rule *Sentence* $\to$ *Subject Predicate*, while the grammar for the abstract language $\{a^n b^n \,|\, n \geqslant 0\}$ consists of two rules, $S \to aSb$ and $S \to \varepsilon$ (where ε denotes the empty string), and represents a complete inductive definition.

Context-free grammars may be thought of as a logic for inductive descriptions of syntax, in which the propositional connectives available for combining syntactical conditions are restricted to disjunction only. Indeed, having multiple rules for a single nonterminal A essentially represents disjunction: two rules $A \to \alpha$ and $A \to \beta$ mean that a string has the property A if and only if it is representable as α *or* as β. Any other Boolean operations, such as conjunction and negation, are not expressible using ordinary context-free grammars: that is, there is no general way for specifying all strings that satisfy a condition α *and at the same time* another condition β, and no way to denote the set of strings that do not satisfy a condition α. Furthermore, it is well-known that intersection of two context-free languages or the complement of a context-free language need not be context-free [126]; in other words, in this particular logic, conjunction or negation cannot be represented through disjunction only.

This omission in the formalism suggests the idea of defining a more general logic, which would maintain the main principles behind the context-free grammars, but at the same time extend the set of available propositional connectives. The result could then be regarded as a *completion* of the incomplete standard definition of context-free grammars. Early attempts in this direction were made by Latta and Wall [68] and by Heilbrunner and Schmitz [39], who proposed formalisms for specifying Boolean combinations of context-free languages. Latta and Wall [68], in particular, argued for the relevance of their formalism to linguistics. However, the use of conjunction and negation in these grammars was heavily restricted, and one still could not use them as freely as the disjunction.

An extension of the definition of context-free grammars featuring unrestricted conjunction, introduced by the author [83], is known as a *conjunctive grammar*. In these grammars, the conjunction of two syntactical conditions can be directly expressed in the form of a rule

$$A \to X_1 \dots X_\ell \,\&\, Y_1 \dots Y_m,$$

which asserts that every string representable both as $X_1 \dots X_\ell$ and as $Y_1 \dots Y_m$ therefore has the property A. The more general *Boolean grammars* [94] further extend the definition by allowing an explicit negation, that is, every Boolean operation is directly expressible in their formalism. For instance, the set of strings representable as $X_1 \dots X_\ell$, and at the same time not representable as $Y_1 \dots Y_m$, can be written as a rule

$$A \to X_1 \dots X_\ell \,\&\, \neg Y_1 \dots Y_m.$$

Both types of grammars remain essentially context-free, in the sense, that the deduction of the properties of a string does not depend on the context, where it occurs; the properties of a string are defined as a function of the properties of the substrings, into which the string can be split. Therefore, as rightfully noted by Kountouriotis et al. [63], "conjunctive context-free grammars" and "Boolean context-free grammars" would be appropriate names for these models. Along with most of the literature, this paper assumes the shorter names, yet refers to the fragment of Boolean grammars featuring the disjunction only as to the *ordinary context-free grammars*.

All semiformal interpretations of rules given so far show the intended meaning of grammars, but are not yet definitions in the mathematical sense. Viewing grammars as a logic, the most direct approach for defining their semantics is by introducing a deduction of elementary statements of the form "string w has property A", which are

inferred from each other according to the rules of the grammar. This general approach may be regarded as folklore; in particular, it was used by Sikkel [129] to explain computations carried out by parsers. Alternatively, the same logical dependencies can be represented by interpreting a grammar as a system of equations with formal languages as unknowns, as done by Ginsburg and Rice [30]. Finally, the most widespread definition of ordinary context-free grammars, given by Chomsky [15], is by string rewriting, when a rule $A \to \alpha$ is regarded as a production for rewriting a symbol A with a substring α, so that an abstract symbol for a sentence is eventually rewritten into an actual sentence. However, it must be stressed that even though the definition by rewriting is indeed the simplest one, it is nothing more than a convenient characterization, which leaves behind the true nature of formal grammars: that is, logical dependence between items of the form "string w has property A". Identifying formal grammars with rewriting systems is a grave error in judgment.

A conjunctive grammar can be defined in the same three ways as an ordinary context-free grammar: *by a deduction system* [95] with appropriately extended inference rules, *by language equations* [87] involving the intersection operation, and *by rewriting* [83], which is augmented to a special kind of term rewriting. These definitions are explained in detail in Section 2 of this survey, along with several characteristic examples of conjunctive grammars, which demonstrate, how conjunction can be put to use in the well-known setting of inductive definitions of syntax.

The definition of Boolean grammars is more complicated, because a grammar may express a contradiction of the form $A \to \neg A$, which states that a string has the property A if and only if it does not have this property. Thus, for Boolean grammars, the dependence of items of the form "string w has property A" has a more complicated form, which calls for being expressed by equations. The simpler approach to the definition uses a system of language equations, in which the negation is interpreted by complementation, and imposes a certain condition upon this system, which ensures that it has a unique solution; this unique solution then defines the meaning of a grammar. A more general definition was given by Kountouriotis et al. [62], who interpreted a Boolean grammar in terms of three-valued languages, so that a string may belong to a language, not belong to it, or have an undetermined membership status. Both methods are explained in Section 3. There is no known definition of Boolean grammars by rewriting.

This survey paper is aimed to present the research on conjunctive and Boolean grammars carried out over the last decade, and to justify the thesis that these grammars are the true general case of the context-free grammars. The crucial points in support of this statement are that, on the one hand, conjunctive and Boolean grammars maintain the main inductive principles behind the ordinary context-free grammars, which account for their intuitive clarity and suitability for representing syntax, and only offer additional logical connectives within the same framework; these further expressive means allow giving meaningful descriptions of quite a few syntactic constructs not representable by ordinary context-free grammars. On the other hand, this extra power does not damage the crucial properties of context-free grammars: the intuitive clarity of descriptions is preserved, the upper bounds on time complexity remain the same, and most of the parsing algorithms are directly inherited from the context-free case. In particular, the basic bottom-up parsing algorithms for ordinary context-free grammars of the general form, such as the Cocke–Kasami–Younger [59, 143] and its variants [33, 60, 138], extend to Boolean grammars so smoothly and obviously, that one can hardly see any reason for limiting logical connectives in a grammar to disjunction only. Applying some other algorithms, such as the Lang–Tomita generalized LR [65, 136] and the recursive descent, to Boolean grammars requires elaborating their flow control, but, in general, the elaboration amounts to having a parser compute conjunction and negation, wherever these operations occur in the grammar.

Although conjunctive and Boolean grammars inherit many practical properties of ordinary context-free grammars, they have some essential differences in their theoretical properties. One of these differences concerns sublinear-time parallel recognition algorithms operating on a circuit: such algorithms are known for ordinary context-free grammars [124, 12, 125], but most likely do not exist already for conjunctive grammars, since these grammars are capable of representing some P-complete languages. Another difference concerns the decision problems: for an ordinary context-free grammar, one can effectively test whether it generates a non-empty language, but it is undecidable whether a given grammar generates the set of all strings; both problems are undecidable for conjunctive grammars. Yet another difference concerns grammars over a one-symbol alphabet: while ordinary context-free grammars are limited to regular subsets of a^*, conjunctive grammars can generate a wide variety of one-symbol languages [49, 50, 51].

The last topic of this survey is summarizing the properties of conjunctive and Boolean grammars, and comparing them with the properties of other important families of formal grammars. Considering all types of grammars together, and understanding conjunctive and Boolean grammars as an essential part of the theory of formal grammars, leads to a new outlook on grammars as such. This new outlook begins with a new classification of meaningful families of formal grammars, done in terms of the amount of ambiguity and nondeterminism, various motivated restrictions on the form of rules (such as linear concatenation), and the set of allowed logical connectives (limited to disjunction alone in ordinary context-free grammars).

The proposed classification of grammars notably ignores the first and still the most well-known classification of families of formal languages: the *Chomsky hierarchy*. Why is it ignored? Chomsky's hierarchy is com-

prised of the regular languages ("type 3"), the context-free languages ("type 2"), the nondeterministic linear space ("type 1") and the recursively enumerable sets ("type 0"). These are the families of languages considered in the early days of computer science by Chomsky [15], who had formalized the intuitive notion of a formal grammar using string-rewriting systems, and then attempted to implement further linguistic ideas by altering this definition. This had a surprising outcome: though none of the modifications had anything to do with syntax, all three of them turned out to be important models of computation: "type 0" is a reformulation of a nondeterministic Turing machine, "type 3" reformulates nondeterministic finite automata, and "type 1" became the first computational complexity class to be ever considered. Putting these three models of computability, along with the basic model of syntax, within a single framework had a significant impact on the early development of the theory of computation, and Chomsky's hierarchy remains a milestone in the history of computer science. However, as far as models of syntax are being concerned, this hierarchy did not serve its purpose. Despite decades of subsequent laborious studies, the research in string-rewriting systems centered around context-free rewriting revealed no other viable model of syntax besides the context-free grammar. This leads to a conclusion that the representation of context-free grammars by string rewriting is a unique coincidence, rather than a systematic association between rewriting and grammars. Furthermore, the ungrammatical levels of the Chomsky hierarchy ("type 1" and "type 0") are useless even as a point of reference, because meaningful syntax has complexity much below NSPACE(n). In spite of its historical importance, the Chomsky hierarchy is hardly relevant anymore.

The research on formal grammars carried out over the last fifty years revealed quite a few important families of formal grammars, obtained by restricting ordinary context-free grammars: LR(1) grammars (and the deterministic context-free languages they generate), linear grammars, unambiguous grammars, etc. These families form the basis of the proposed hierarchy of formal grammars, which is then extended towards the conjunctive and Boolean grammars, as well as to their subfamilies defined by analogy with the subfamilies of ordinary context-free grammars: such as, for instance, linear conjunctive grammars or unambiguous Boolean grammars. In Section 8, all families in the hierarchy are compared in terms of their expressive power, closure properties under basic operations and the decidability and complexity of various properties. The expressive power of different grammars is furthermore related to the computational complexity classes between NC^1 and P.

The survey is concluded with some suggested directions for research on conjunctive and Boolean grammars, and on formal grammars in general. First, there is a list of significant theoretical open problems, with an award of $360 Canadian offered by the author [102] for the first correct solution of each problem. Nine problems were originally stated in 2006; since then, two problems were solved [49, 105], and, at the time of writing, seven remain open. This survey includes the statements of all problems, and briefly comments on possible approaches to them. Furthermore, some general questions worth investigation are suggested, including possible discovery of new variants of formal grammars, as well as implementation and application of conjunctive and Boolean grammars as they are.

2. Conjunctive grammars

2.1. Three equivalent definitions

A conjunctive grammar is a quadruple $G = (\Sigma, N, R, S)$, in which:

- Σ is the *alphabet* of the language being defined, that is, a finite set of symbols, from which the strings in the language are built;
- N is a finite set of auxiliary notions used in the grammar, each of them represents a syntactic property that a string in Σ^* may have or not have; for historical reasons, they are called *nonterminal symbols* or *nonterminals*, even though this name ought to have been deprecated long ago;
- R is a finite set of *grammar rules*, each of the form

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m, \tag{1}$$

  with $A \in N$, $m \geqslant 1$ and $\alpha_1, \ldots, \alpha_m \in (\Sigma \cup N)^*$;
- $S \in N$ is a distinguished nonterminal symbol representing the set of syntactically well-formed sentences in the language, and colloquially referred as a *start symbol* (although S more appropriately stands for "sentence" [15]) or as an *initial symbol*.

For every rule (1), each string α_i is called a *conjunct*, and if a grammar has a unique conjunct in every rule, it is an ordinary context-free grammar. A collection of rules for a single nonterminal can be written using the shorthand notation

$$A \to \alpha_{1,1} \,\&\, \ldots \,\&\, \alpha_{1,n_1} \mid \ldots \mid \alpha_{n,1} \,\&\, \ldots \,\&\, \alpha_{n,m_n},$$

in which the vertical lines are, in essence, the disjunction.

Informally, a rule (1) states that if a string is representable as each concatenation α_i, then it has the property A. This understanding can be formalized in three equivalent ways.

The first definition is by *rewriting*. The rewriting in conjunctive grammars is carried out generally in the same way as in the ordinary context-free case. The only difference is in the objects being transformed: while context-free rewriting operates with strings over $\Sigma \cup N$, which are terms over concatenation, rewriting in conjunctive grammars use

terms over concatenation and conjunction. Such a definition was first given in an unpublished Master's thesis by Szabari [133], which, unfortunately, was not noticed in its time.

Definition 1 (Szabari [133], Okhotin [83]). *Given a grammar G, consider terms over concatenation and conjunction with symbols from $\Sigma \cup N$ and the empty string ε as atomic terms. Assume that the symbols "(", "&" and ")" used to construct the terms are not in $\Sigma \cup N$. The relation $\Longrightarrow$ of one-step rewriting on the set of terms is defined as follows:*

- *Using a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \in R$, any atomic subterm A of any term can be rewritten by the subterm $(\alpha_1 \,\&\, \ldots \,\&\, \alpha_m)$:*

$$\ldots A \ldots \Longrightarrow \ldots (\alpha_1 \,\&\, \ldots \,\&\, \alpha_m) \ldots$$

- *A conjunction of several identical strings in Σ^* can be rewritten by one such string: for every $w \in \Sigma^*$,*

$$\ldots (w \,\&\, \ldots \,\&\, w) \ldots \Longrightarrow \ldots w \ldots$$

The relations of rewriting in zero or more steps, in one or more steps and in exactly ℓ steps are denoted by $\Longrightarrow^*$, $\Longrightarrow^+$ and $\Longrightarrow^\ell$, respectively. The language generated by a term φ is the set of all strings over Σ obtained from it in a finite number of rewriting steps:

$$L_G(\varphi) = \{w \mid w \in \Sigma^*,\ \varphi \overset{G}{\Longrightarrow}{}^* w\}.$$

The language generated by the grammar is the language generated by the term S:

$$L(G) = L_G(S) = \{w \mid w \in \Sigma^*,\ S \overset{G}{\Longrightarrow}{}^* w\}.$$

For simplicity, when a single-conjunct rule $A \to \alpha$ is applied, one can omit the parentheses, and rewrite A with α, rather than with (α).

The next definition is given by a formal deduction system, which represents the logical content of conjunctive grammars directly.

Definition 2 (Okhotin [95]). *For a conjunctive grammar $G = (\Sigma, N, R, S)$, let $\{X(w) \mid X \in \Sigma \cup N,\ w \in \Sigma^*\}$ be the set of elementary propositions (items), each representing a statement of the form "a string w has a property X". The deduction system uses the following axioms:*

$$\vdash a(a) \quad (\textit{for all } a \in \Sigma).$$

Every rule $A \to X_{1,1} \ldots X_{1,\ell_1} \,\&\, \ldots \,\&\, X_{m,1} \ldots X_{m,\ell_m} \in R$ in the grammar, with $m \geqslant 1$, $\ell_i \geqslant 0$ and $X_{i,j} \in \Sigma \cup N$, acts as a schema for deduction rules: for all strings $u_{i,j} \in \Sigma^*$ with $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant \ell_i$, that satisfy $u_{1,1} \ldots u_{1,\ell_1} = \ldots = u_{m,1} \ldots u_{m,\ell_m} = w$,

$$X_{1,1}(u_{1,1}), \ldots, X_{m,\ell_m}(u_{m,\ell_m}) \vdash A(w).$$

Whenever an item $X(w)$ can be deduced from the above axioms by the given deduction rules, this is denoted by $G \vdash X(w)$. Define $L_G(X) = \{w \mid G \vdash X(w)\}$ and $L(G) = L_G(S) = \{w \mid G \vdash S(w)\}$.

Another equivalent definition of the language generated by a conjunctive grammar is by a solution of a system of equations with languages as unknowns. Such a definition for ordinary context-free grammars, given by Ginsburg and Rice [30], is well-known: a grammar is transcribed as a system of equations

$$A = \bigcup_{A \to X_1 \ldots X_\ell \in R} X_1 \cdot \ldots \cdot X_\ell \quad (\text{for all } A \in N),$$

where nonterminal symbols $A \in N$ are unknown languages, each symbol of the alphabet $X = a \in \Sigma$ represents a constant language $\{a\}$, an empty concatenation with $\ell = 0$ is a constant language $\{\varepsilon\}$, and multiple rules for a single nonterminal are represented by union. Every such system has solutions, among them the *least solution* with respect to componentwise inclusion, and this solution consists of exactly the languages generated by the nonterminals of the grammar by rewriting [30].

This definition is directly extended to conjunctive grammars, with the conjunction in a rule represented by intersection. The resulting system is bound to have a least solution by the same basic lattice-theoretic argument as in the ordinary case with disjunction only, which is based only on the fact that the right-hand sides of the equations are monotone and continuous functions (a property shared by the intersection operation).

Definition 3 (Okhotin [87]). *For every conjunctive grammar $G = (\Sigma, N, R, S)$, the associated system of language equations is a system of equations in variables N, with each variable representing an unknown language over Σ. The system contains the following equation for every variable $A \in N$.*

$$A = \bigcup_{A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \in R} \bigcap_{i=1}^{m} \alpha_i \tag{2}$$

Each α_i in the equation is a concatenation of variables and constant languages $\{a\}$ representing symbols of the alphabet, or constant $\{\varepsilon\}$ if α_i is the empty string. Let $(\ldots, L_A, \ldots)_{A \in N}$ be the least solution of this system. Then $L_G(A)$ is defined as L_A for each $A \in N$, and $L(G) = L_S$.

This least solution can be obtained as a limit of an ascending sequence of vectors of languages, with the first element $\bot = (\varnothing, \ldots, \varnothing)$, and with every next element obtained by applying the right-hand sides of the system (2) as a vector function $\varphi : \left(2^{\Sigma^*}\right)^{|N|} \to \left(2^{\Sigma^*}\right)^{|N|}$ to the previous element. Since this function is monotone with respect to the partial ordering $\sqsubseteq$ of componentwise inclusion, the

resulting sequence $\{\varphi^k(\bot)\}_{k\to\infty}$ is ascending, and the continuity of φ implies that its limit (least upper bound)

$$\bigsqcup_{k\geqslant 0} \varphi^k(\bot) \tag{3}$$

is the least solution. This is the same argument as used for ordinary context-free grammars [30].

Definitions 1 (term rewriting), 2 (deduction) and 3 (equations) can be proved equivalent [87, 95], and the statement of their equivalence reads as follows:

Theorem 1. *Let $G = (\Sigma, N, R, S)$ be a conjunctive grammar. For every $A \in N$ and $w \in \Sigma^*$, the following statements are equivalent:*

- $A \Longrightarrow^* w$,
- $\vdash A(w)$,
- *w is in the A-component of $\bigsqcup_{k\geqslant 0} \varphi^k(\bot)$.*

A seemingly similar idea of extending Chomsky's context-free string rewriting with *alternation* (that is, alternation of existential and universal nondeterminism) was investigated by Moriya [81] and by Ibarra, Jiang and Wang [46]. This turned out to be a strange but very powerful model of computation, capable of representing at least all languages in $\mathrm{DTIME}(2^{O(n)})$, and quite unrelated to the task of defining the syntax. The difference between the implementation of conjunction in conjunctive grammars and the concept of alternation in the computation theory is that the latter applies logical conjunction to results of several computations, which need not operate on the same data, while conjunction in conjunctive grammars applies to multiple parses *of the same substring*. Aizikowitz and Kaminski [2] regarded this property as "synchronized alternation", and investigated it in the context of pushdown automata. A similar restriction on alternation was employed by Lange [66] in a model similar to a conjunctive grammar used for infinite strings.

2.2. Examples

Conjunctive grammars can obviously express everything that ordinary context-free grammars can. The new expressive means allowed by the conjunction operation will be illustrated by four representative examples.

The simplest use of conjunction is to represent intersection of separately defined ordinary context-free languages. This is what is done in the following first example, which is given mainly to illustrate the definitions.

Example 1. *The following conjunctive grammar generates the language $\{a^n b^n c^n \mid n \geqslant 0\}$:*

$$\begin{array}{lcl} S & \to & AB \,\&\, DC \\ A & \to & aA \mid \varepsilon \\ B & \to & bBc \mid \varepsilon \\ C & \to & cC \mid \varepsilon \\ D & \to & aDb \mid \varepsilon \end{array}$$

The grammar is based upon the representation of this language as an intersection of two ordinary context-free languages:

$$\underbrace{\{a^i b^j c^k \mid j = k\}}_{L(AB)} \cap \underbrace{\{a^i b^j c^k \mid i = j\}}_{L(DC)} = \underbrace{\{a^n b^n c^n \mid n \geqslant 0\}}_{L(S)}.$$

According to the definition by term rewriting, the string *abc* can be obtained by the following transformation.

$$S \Longrightarrow (AB \,\&\, DC) \Longrightarrow (aAB \,\&\, DC) \Longrightarrow (aB \,\&\, DC) \Longrightarrow (abBc \,\&\, DC) \Longrightarrow (abc \,\&\, DC) \Longrightarrow (abc \,\&\, aDbC) \Longrightarrow (abc \,\&\, abC) \Longrightarrow (abc \,\&\, abcC) \Longrightarrow (abc \,\&\, abc) \Longrightarrow abc$$

In essence, here two context-free rewritings are carried out independently of each other, and both AB and DC have to be rewritten to the same string, in order to perform the last step of the rewriting.

Turning to the definition by deduction, consider the following logical derivation of the fact that *abc* is in $L(G)$:

$$\begin{array}{rl} \vdash a(a) & (\text{axiom}) \\ \vdash b(b) & (\text{axiom}) \\ \vdash c(c) & (\text{axiom}) \\ \vdash A(\varepsilon) & (A \to \varepsilon) \\ a(a), A(\varepsilon) \vdash A(a) & (A \to aA) \\ \vdash B(\varepsilon) & (B \to \varepsilon) \\ b(b), B(\varepsilon), c(c) \vdash B(bc) & (B \to bBc) \\ \vdash D(\varepsilon) & (D \to \varepsilon) \\ a(a), D(\varepsilon), b(b) \vdash D(ab) & (D \to aDb) \\ \vdash C(\varepsilon) & (C \to \varepsilon) \\ c(c), C(\varepsilon) \vdash C(c) & (C \to cC) \\ A(a), B(bc), D(ab), C(c) \vdash S(abc) & (S \to AB \,\&\, DC) \end{array}$$

According to the definition by language equations, the system corresponding to this grammar is

$$\begin{cases} S &= AB \cap DC \\ A &= \{a\}A \cup \{\varepsilon\} \\ B &= \{b\}B\{c\} \cup \{\varepsilon\} \\ C &= \{c\}C \cup \{\varepsilon\} \\ D &= \{a\}D\{b\} \cup \{\varepsilon\}, \end{cases}$$

and this system has a unique solution with $S = \{a^n b^n c^n \mid n \geqslant 0\}$, $A = a^*$, $B = \{b^m c^m \mid m \geqslant 0\}$, $C = c^*$ and $D = \{a^m b^m \mid m \geqslant 0\}$.

An important property of conjunctive grammars is that the parse of a string according to a grammar can be represented in the form of a tree with shared leaves, which generalizes ordinary context-free parse trees. The parse tree of the string *abc* with respect to the above grammar is given in Figure 1, and one can clearly see how it combines two interpretations of the same string according to the two conjuncts in the rule for S. This tree is essentially

a proof tree corresponding to the deduction of the item $S(w)$. A formal definition of parse trees can be found in the literature [83, 103].

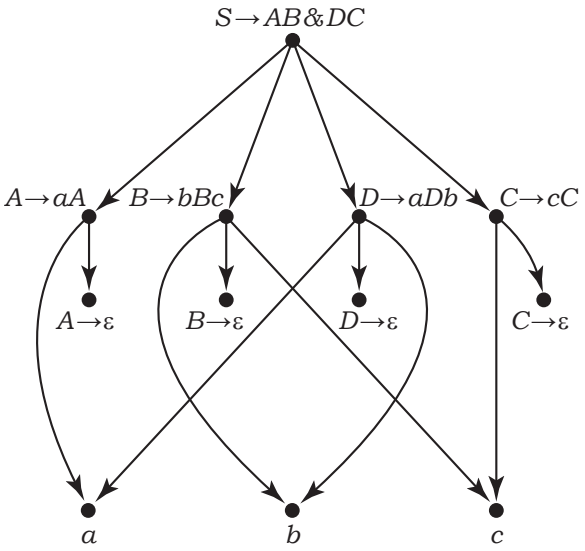

Figure 1: Parse tree of the string *abc* according to the conjunctive grammar for $\{a^n b^n c^n \mid n \geqslant 0\}$ given in Example 1.

The second example illustrates the extended possibilities for induction in conjunctive grammars. Consider another typical language that has no ordinary context-free grammar: $\{wcw \mid w \in \{a,b\}^*\}$. This language represents such syntactic constructs as identifier checking in programming languages. As proved by Wotschke [141], it is not expressible as a finite intersection of ordinary context-free languages. Constructing a conjunctive grammar for this language thus requires more than putting a conjunction on top of an ordinary context-free grammar.

Example 2 (Okhotin [83]). *The following conjunctive grammar generates the language* $\{wcw \mid w \in \{a,b\}^*\}$*:*

$$\begin{array}{lcl} S & \to & C \,\&\, D \\ C & \to & aCa \mid aCb \mid bCa \mid bCb \mid c \\ D & \to & aA \,\&\, aD \mid bB \,\&\, bD \mid cE \\ A & \to & aAa \mid aAb \mid bAa \mid bAb \mid cEa \\ B & \to & aBa \mid aBb \mid bBa \mid bBb \mid cEb \\ E & \to & aE \mid bE \mid \varepsilon \end{array}$$

The symbol C defines the language $\{xcy \mid x,y \in \{a,b\}^*; |x| = |y|\}$ in the standard way, and thus the conjunction with C in the rule for S ensures that the string consists of two parts of equal length separated by a center marker. The other conjunct D checks that the symbols in corresponding positions are the same. The actual language generated by D is $L(D) = \{uczu \mid u,z \in \{a,b\}^*\}$, and it is defined inductively as follows: a string is in $L(D)$ if and only if

- either it is in $c\{a,b\}^*$ (the base case: no symbols to compare),
- or its first symbol is the same as the corresponding symbol on the other side, *and* the string without its first symbol is in $L(D)$ (that is, the rest of the symbols in the left part correctly correspond to the symbols in the right part).

The comparison of a single symbol to the corresponding symbol on the right is done by the nonterminals A and B, which generate the languages $\{xcvay \mid x,v,y \in \{a,b\}^*,\ |x| = |y|\}$ and $\{xcvby \mid x,v,y \in \{a,b\}^*,\ |x| = |y|\}$, respectively, and the above inductive definition is directly expressed in the rules for D, which recursively refer to D in order to apply the same rule to the rest of the string. Finally, the rule for S defines the set of strings of the form xcy with $|x| = |y|$ (ensured by C) and $x = y$ (imposed by D), and

$$\{xcy \mid x,y \in \{a,b\}^*,\ |x| = |y|\} \cap \{uczu \mid u,z \in \{a,b\}^*\} = \\ = \{wcw \mid w \in \{a,b\}^*\}.$$

It is important to note that the construction essentially uses the center marker, and therefore this method cannot be applied to constructing a conjunctive grammar for the language $\{ww \mid w \in \{a,b\}^*\}$. The question of whether $\{ww \mid w \in \{a,b\}^*\}$ can be generated by any conjunctive grammar remains an open problem.

The grammar in the third example defines the requirement of *declaration before use*.

Example 3. *The following grammar generates the language* $\{u_1 \dots u_n \mid n \geqslant 0$*, and for every* i*, either* $u_i \in da^*$*, or* $u_i = ca^k$ *and* $u_j = da^k$ *for some* $j < i$ *and* $k \geqslant 0\}$*.*

$$\begin{array}{lcl} S & \to & SdA \mid ScA \,\&\, EdB \mid \varepsilon \\ A & \to & aA \mid \varepsilon \\ B & \to & aBa \mid Ec \\ E & \to & cAE \mid dAE \mid \varepsilon \end{array}$$

A substring of the form da^k *represents a declaration of* k*, and every substring of the form* ca^k *is a reference to* k*, which requires an earlier declaration* da^k*.*

The grammar applies generally the same technique of inductive definitions as in Example 2. The rule $S \to \varepsilon$ asserts that an empty sequence of declarations and references has the required property. The rule $S \to SdA$ appends a new declaration (dA) to a well-formed string with all references preceded by declarations (S). The other rule $S \to ScA \,\&\, EdB$ similarly appends a reference (cA), and at the same time ensures that this new reference has a preceding declaration (EdB). Here E defines an arbitrary sequence of declarations and references, and the concatenation EdB defines a suitable partition of the string, where the symbol d begins the appropriate declaration, and B ensures that the number of symbols a is the same in the declaration and in the reference.

The grammars in Examples 2 and 3 can be easily combined to represent declaration before use with identifiers

over a two-symbol alphabet. More constructs from programming languages are representable using these methods, and a full grammar for a model programming language can be found in the literature [96].

The fourth example shows that conjunctive grammars over a one-symbol alphabet have a non-trivial expressive power, unlike the ordinary context-free grammars, which are limited to regular unary languages [30]. Though this does not exactly pertain to defining syntax, the demonstrated technique for constructing conjunctive grammars for unary languages has numerous theoretical implications explained later in Section 7.1.

Example 4 (Jeż [49]). *The following conjunctive grammar with the initial symbol A_1 generates the language $\{a^{4^n} \mid n \geqslant 0\}$:*

$$\begin{array}{lcl} A_1 & \to & A_1A_3 \,\&\, A_2A_2 \mid a \\ A_2 & \to & A_1A_1 \,\&\, A_2A_6 \mid aa \\ A_3 & \to & A_1A_2 \,\&\, A_6A_6 \mid aaa \\ A_6 & \to & A_1A_2 \,\&\, A_3A_3 \end{array}$$

Each nonterminal A_i generates the language $\{a^{i\cdot 4^n} | n \geqslant 0\}$.

This grammar is best explained in terms of base-4 notation of the length of the strings. Then each nonterminal A_i with $i \in \{1, 2, 3\}$ represents base-4 numbers $i\mathtt{0}\ldots\mathtt{0}$, or $\mathtt{120}\ldots\mathtt{0}$ for A_6. Substituting these four languages into the equation

$$A_1 = (A_1A_3 \cap A_2A_2) \cup \{a\},$$

the first concatenation A_1A_3 produces all numbers with the notation $\mathtt{10^*30^*}$, $\mathtt{30^*10^*}$ and $\mathtt{10^+}$, of which the latter is the intended set, while the rest are regarded as garbage. The second concatenation A_2A_2 yields $\mathtt{20^*20^*}$ and $\mathtt{10^+}$. Though both concatenations contain some garbage, the garbage in the concatenations is disjoint, and is accordingly filtered out by the intersection, which produces exactly the numbers with the notation $\mathtt{10^+}$, that is, the language $\{a^{4^n} \mid n \geqslant 1\}$. Finally, the union with $\{a\}$ yields the language $\{a^{4^n} \mid n \geqslant 0\}$, and thus the first equation turns into an equality. The rest of the equations are verified similarly. Since the membership of each string in the languages $L(A_i)$ depends on the membership of strictly shorter strings, the grammar defines the correct languages by a proper induction.

Constructing conjunctive grammars for the following languages is left as an exercise to the reader:

- $\{a^m b^n c^m d^n \mid m, n \geqslant 0\}$;
- $\{w \mid w \in \{a, b, c\}^*,\ |w|_a = |w|_b = |w|_c\}$, where $|w|_s$ stands for the number of occurrences of a symbol s in w;
- $\{(wc)^{|w|} \mid w \in \{a, b\}^*\}$;
- $\{da^{k_1} \ldots da^{k_n} \mid n, k_1, \ldots, k_n \geqslant 0$, and the numbers $k_1, \ldots, k_n$ are pairwise distinct$\}$, which adopts the encoding from Example 3 and represents the condition of having no duplicate declarations;
- $\{a^{5^n} \mid n \geqslant 0\}$.

This can be done by applying the methods presented in Examples 1–4.

Another technique for constructing conjunctive grammars is illustrated in the following example:

Example 5 (Okhotin, Reitwießner [110]). *The following conjunctive grammar generates the set of palindromes of odd length over $\{a, b\}$:*

$$\begin{array}{l} S \to AB \,\&\, O \mid a \mid b \\ A \to aSa \mid \varepsilon \\ B \to bSb \mid \varepsilon \\ O \to OOO \mid a \mid b \end{array}$$

This language would normally be represented by an ordinary context-free grammar with the rules $S \to aSa \mid bSb \mid a \mid b$, which essentially uses two rules involving nonterminals—that is, a disjunction of two non-constant terms. On the contrary, the above conjunctive grammar uses disjunction only with constant terms. The construction is based on the fact that all strings with the property S are of odd length. The nonterminal O to generates all strings of odd length, and then the rule $S \to AB \,\&\, O$ simulates the disjunction of aSa and bSb as follows:

$$\begin{aligned} &(aSa \cup \{\varepsilon\})(bSb \cup \{\varepsilon\}) \cap \mathrm{Odd} = \\ &\quad = (aSabSb \cup aSa \cup bSb \cup \{\varepsilon\}) \cap \mathrm{Odd} = aSa \cup bSb. \end{aligned}$$

2.3. Normal forms

A few normal forms for conjunctive grammars are known. The first of them is a direct generalization of the *Chomsky normal form* of context-free grammars, in which every rule is either of the form $A \to BC$ with $B, C \in N$, or of the form $A \to a$ with $a \in \Sigma$, with a possible rule $S \to \varepsilon$ for the initial symbol.

Theorem 2 (Okhotin [83]). *Every conjunctive grammar can be effectively transformed to a conjunctive grammar generating the same language, which is in the* ***binary normal form****, that is, with every rule of the form*

$$\begin{array}{ll} A \to B_1C_1 \,\&\, \ldots \,\&\, B_mC_m & (m \geqslant 1,\ B_i, C_i \in N) \\ A \to a & (a \in \Sigma) \\ S \to \varepsilon, & \end{array}$$

where the last rule is allowed only if S does not appear in the right-hand sides of any rules.

The known transformation proceeds by first eliminating *epsilon conjuncts*, that is, rules of the form $A \to \varepsilon \,\&\, \ldots$ that can potentially generate the empty string. At the second step, *unit conjuncts*, that is, rules of the form $A \to B \,\&\, \ldots$ that potentially cause circularities in the definition, are eliminated. Elimination of epsilon conjuncts can be achieved with only a linear blowup in the size of

the grammar, but the known procedure for eliminating unit conjuncts leads, in the worst case, to an exponential blowup.

The other normal form theorem reflects on the use of Boolean operations in a conjunctive grammar. It turns out that it is sufficient to apply disjunction in the following restricted form:

Theorem 3 (Okhotin, Reitwießner [110]). *For every conjunctive grammar there exists and can be effectively constructed a conjunctive grammar generating the same language, in which the set of rules for every nonterminal A is of the form*

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \mid w_1 \mid \ldots \mid w_k,$$

where $m \geqslant 1$, $k \geqslant 0$, $\alpha_i \in (\Sigma \cup N)^*$ and $w_j \in \Sigma^*$.

That is, it is sufficient to have right-hand sides of the rules for every nonterminal form a logical expression, in which disjunction is always a disjunction with a singleton constant $\{w_i\}$, and disjunction of two arbitrary expressions (represented by two rules of an unrestricted form) is not allowed.

The construction elaborates on the ideas of Example 5, first ensuring that all nonterminals (except, maybe, the initial symbol) generate only strings of odd length. The latter intermediate form is of interest on its own.

Theorem 4 (Okhotin, Reitwießner [110]). *For every conjunctive grammar there exists and can be effectively constructed a conjunctive grammar $G = (\Sigma, N, R, S)$ generating the same language, which is in* ***odd normal form****, that is, with all rules of the form*

$$A \to B_1 a_1 C_1 \,\&\, \ldots \,\&\, B_m a_m C_m \quad (m \geqslant 1,\ B_i, C_i \in N,\ a_i \in \Sigma)$$
$$A \to a \quad (a \in \Sigma)$$

If S is never used in the right-hand sides of any rules, then the following two types of rules are also allowed:

$$S \to aA \quad (a \in \Sigma,\ A \in N)$$
$$S \to \varepsilon$$

For ordinary context-free grammars, there is another important normal form: the *Greibach normal form* [34], in which every rule is either $A \to a\alpha$ with $a \in \Sigma$ and $\alpha \in (\Sigma \cup N)^*$, or $A \to \varepsilon$. This definition naturally carries on to conjunctive grammars. It can be said that a conjunctive grammar $G = (\Sigma, N, R, S)$ is in Greibach normal form if every rule in R is of the form

$$A \to a\alpha_1 \,\&\, \ldots \,\&\, a\alpha_n \quad (n \geqslant 1,\ a \in \Sigma,\ \alpha_i \in N^*) \quad \text{or}$$
$$A \to \varepsilon.$$

A transformation to this form is known only for the special case of a one-symbol alphabet. It can be inferred from Theorem 4 by first transforming a grammar to the odd normal form, and then using the commutativity of concatenation of unary languages.

Corollary 4.1. *For every conjunctive grammar over an alphabet $\Sigma = \{a\}$, there exists and can be effectively constructed a conjunctive grammar in the Greibach normal form generating the same language.*

It remains unknown, whether every conjunctive grammar over an unrestricted alphabet can be transformed to that form (see Problem 5 in Section 9.1).

3. Boolean grammars

3.1. Intuitive definition

Boolean grammars are context-free grammars equipped with all propositional connectives, or, in other words, conjunctive grammars augmented with negation. Conversely, conjunctive grammars are the monotone fragment of Boolean grammars.

A Boolean grammar is a quadruple $G = (\Sigma, N, R, S)$, in which

- Σ is the alphabet;
- N is the set of nonterminal symbols;
- R is a finite set of rules of the form

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n \qquad (4)$$

  with $A \in N$, $m, n \geqslant 0$, $m + n \geqslant 1$ and $\alpha_i, \beta_j \in (\Sigma \cup N)^*$;
- $S \in N$ is the initial symbol.

The only difference from a conjunctive grammar is that some conjuncts can be negated: the conjuncts α_i and $\neg\beta_j$ are called *positive* and *negative* respectively, with the notation $\pm\alpha_i$ and $\pm\beta_j$ occasionally used to refer to conjuncts, without specifying whether they are positive or negative. A rule (4) can be read as follows: *"if a string is representable in the form $\alpha_1, \ldots, \alpha_m$, but is not representable in the form $\beta_1, \ldots, \beta_n$, then this string has the property A".* This intuitive interpretation is not yet a formal definition, but this understanding is sufficient to construct grammars.

Example 6 (cf. Example 1). *The following Boolean grammar generates the language $\{a^m b^n c^n \mid m, n \geqslant 0, m \neq n\}$:*

$$\begin{array}{rcl} S & \to & AB \,\&\, \neg DC \\ A & \to & aA \mid \varepsilon \\ B & \to & bBc \mid \varepsilon \\ C & \to & cC \mid \varepsilon \\ D & \to & aDb \mid \varepsilon \end{array}$$

The rules for the nonterminals A, B, C and D are context-free, so, according to the intuitive semantics, they should generate the same languages as in Example 1. Then the

propositional connectives in the rule for S specify the following combination of the conditions given by AB and DC:

$$\underbrace{\{a^ib^jc^k \mid j=k\}}_{L(AB)} \cap \overline{\underbrace{\{a^ib^jc^k \mid i=j\}}_{L(DC)}} =$$
$$= \{a^ib^jc^k \mid j=k \text{ and } i\neq j\} = \underbrace{\{a^mb^nc^n \mid m\neq n\}}_{L(S)}.$$

Example 7. *The following Boolean grammar generates the language* $\{ww \mid w \in \{a,b\}^*\}$*:*

$$\begin{array}{lcl} S & \to & \neg AB \,\&\, \neg BA \,\&\, C \\ A & \to & XAX \mid a \\ B & \to & XBX \mid b \\ C & \to & XXC \mid \varepsilon \\ X & \to & a \mid b \end{array}$$

Again, according to the intuitive semantics, the nonterminals A, B, C and X should generate the appropriate ordinary context-free languages, and

$$L(A) = \{uav \mid u,v \in \{a,b\}^*,\ |u| = |v|\},$$
$$L(B) = \{ubv \mid u,v \in \{a,b\}^*,\ |u| = |v|\}.$$

This implies

$$L(AB) = \{uavxby \mid u,v,x,y \in \{a,b\}^*, |u| = |x|, |v| = |y|\},$$

that is, $L(AB)$ contains all strings of even length with a mismatched a on the left and b on the right (in any position). Similarly,

$$L(BA) = \{ubvxay \mid u,v,x,y \in \{a,b\}^*, |u| = |x|, |v| = |y|\}$$

represents the mismatch formed by b on the left and a on the right. Then the rule for S defines the set of strings of even length without such mismatches:

$$L(S) = \overline{L(AB)} \cap \overline{L(BA)} \cap \{aa, ab, ba, bb\}^* = \{ww \mid w \in \{a,b\}^*\}.$$

Though such a common-sense interpretation of Boolean grammars is clear for "reasonable" grammars, the use of negation can, in general, lead to logical contradictions, such as in the grammar $S \to \neg S$. For that reason, giving a mathematically sound formal definition of Boolean grammars is far from being trivial. To be more precise, the dependence of items of the form "string w has property A" becomes *non-monotone*, where the discovery of the fact that some item is true may imply that another item is false. Thus, a correct assignment of truth-values to items is a solution of a certain infinite system of equations with Boolean unknowns. More natural and convenient formalizations are given by representing a grammar as a system of language equations, so that a particular distinguished solution of this system defines the language generated by a grammar.

There are two known definitions of Boolean grammars by equations. One of them uses equations with standard formal languages as unknowns [94], and avoids the resulting difficulties by imposing a restriction upon solutions of those equations. The other definition expands the model towards languages over three-valued logic [62] and assigns a meaning to any grammar.

3.2. Definition by language equations

According to the simplest definition, a grammar is represented by a system of language equations defined analogously to the conjunctive case, with the negation represented by complementation. However, the approach through least solutions, which worked well for conjunctive grammars, is no longer useful in this case, as illustrated in the following example.

$$\begin{array}{lcl} S & \to & \neg A \\ A & \to & A \end{array} \qquad \begin{cases} S = \overline{A} \\ A = A \end{cases}$$

Indeed, all solutions of the associated system of equations are of the form $S = \overline{L}$, $A = L$, and they are pairwise incomparable.

Thus, in the simplest definition, the system of language equations is required to have a *unique solution*, and grammars with no solutions or multiple solutions are considered ill-formed. However, this does not yet guarantee that the membership of a string in the language depends only on the membership of shorter strings, which is essential for grammars to represent inductive definitions. Consider the following grammar, along with the associated system of language equations:

$$\begin{array}{lcl} S & \to & \neg S \,\&\, aA \\ A & \to & A \end{array} \qquad \begin{cases} S = \overline{S} \cap \{a\}A \\ A = A \end{cases}$$

The system has a unique solution $S = A = \varnothing$: indeed, supposing that there is a string $w \in A$, a contradiction of the form "$aw \in S$ *if and only if* $aw \notin S$" is obtained. Thus, in order to determine that $w \notin A$, one has to consider the string aw, which contradicts the principle of inductive definition. Furthermore, there is a theoretical result, that every recursive language is represented by a unique solution of a system of language equations associated to some Boolean grammar [104].

However, once an extra restriction is imposed upon these equations, a feasible definition of Boolean grammars can be obtained. The following condition of *unique solution modulo* $\Sigma^{\leqslant \ell}$ essentially means that the properties of strings of length up to ℓ, as they depend upon each other, are completely resolved without considering any longer strings and their properties.

Definition 4 (Okhotin [94]). *Let* $G = (\Sigma, N, R, S)$ *be a Boolean grammar, and define the associated system of language equations*

$$A = \bigcup_{\substack{A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\& \\ \&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n \in R}} \left[\bigcap_{i=1}^{m} \alpha_i \ \cap \ \bigcap_{j=1}^{n} \overline{\beta_j} \right] \qquad (5)$$

Assume that for every integer $\ell \geqslant 0$ there exists a unique vector of languages $(\ldots, L_A, \ldots)_{A \in N}$ with $L_A \subseteq \Sigma^{\leqslant \ell}$, such that a substitution of L_A for A, for each $A \in N$, turns every equation (5) into an equality modulo intersection with $\Sigma^{\leqslant \ell}$. Then the system is said to have a ***strongly unique solution****, and, for every $A \in N$, the language $L_G(A)$ is defined as L_A from the unique solution of this system. The language generated by the grammar is $L(G) = L_G(S)$.*

Returning to the above grammar with the rules $S \to \neg S \,\&\, aA$ and $A \to A$, consider any number $\ell \geqslant 0$. Then the associated system (5) has two solutions modulo $\Sigma^{\leqslant \ell}$, namely, $(S = \varnothing, A = \varnothing)$ and $(S = \varnothing, A = \{a^\ell\})$. Therefore, the grammar is deemed invalid according to Definition 4.

Consider the Boolean grammar in Example 7. The corresponding system of language equations is

$$\begin{cases} S &= \overline{AB} \cap \overline{BA} \cap C \\ A &= XAX \cup \{a\} \\ B &= XBX \cup \{b\} \\ C &= XXC \cup \{\varepsilon\} \\ X &= \{a\} \cup \{b\} \end{cases}$$

and the following assignment of languages to variables is its unique solution:

$$\begin{aligned} S &= \{ww \mid w \in \{a,b\}^*\}, \\ A &= \{uav \mid u, v \in \{a,b\}^*, |u| = |v|\}, \\ B &= \{ubv \mid u, v \in \{a,b\}^*, |u| = |v|\}, \\ C &= \{aa, ab, ba, bb\}^*, \\ X &= \{a, b\}. \end{aligned}$$

Furthermore, its solution modulo every $\Sigma^{\leqslant \ell}$ with $\ell \geqslant 0$ is unique, and hence $L(G)$ is well-defined as $\{ww \mid w \in \{a,b\}^*\}$.

There is a normal form for Boolean grammars, which further generalizes the binary normal form for conjunctive grammars from Theorem 2.

Theorem 5 (Okhotin [94]). *For every Boolean grammar as in Definition 4 there exists and can be effectively constructed a Boolean grammar generating the same language, which is in the* ***binary normal form****, with all rules of the form*

$$\begin{aligned} &A \to B_1C_1 \,\&\, \ldots \,\&\, B_mC_m \,\&\, \neg D_1E_1 \,\&\, \ldots \,\&\, \neg D_nE_n \,\&\, \neg\varepsilon \\ &\qquad\qquad (m \geqslant 1,\ n \geqslant 0,\ B_i, C_i, D_j, E_j \in N) \\ &A \to a \quad (a \in \Sigma) \\ &S \to \varepsilon \end{aligned}$$

The rule of the last type is allowed only if S does not appear in the right-hand sides of any rules.

Every grammar in the binary normal form satisfies Definition 4.

Definition 4 has its limitations. First, some grammars with a clear meaning do not meet its condition. The simplest such grammar is $S \to S$, where the corresponding language equation $S = S$ has multiple solutions, with the least solution $S = \varnothing$. Even though any conjunctive grammar can be transformed, so that the system of equations has a strongly unique solution (due to Theorems 2 and 5), this restriction is still rather inconvenient.

Another complication surrounding Definition 4 is that it is undecidable whether a given Boolean grammar satisfies this definition [94, Thm. 2]. The construction in Theorem 5 is effective in the following sense: there exists an algorithm, which, given an arbitrary Boolean grammar G, not necessarily satisfying Definition 4, constructs another grammar G' in the binary normal form. If the grammar G satisfies Definition 4, then $L(G') = L(G)$. But if the original grammar does not meet the condition in Definition 4, and $L(G)$ is thus undefined, then the algorithm may not detect this, and will still return some grammar G'.

One more problem is that there exist some extremal examples of grammars that have no intuitive meaning, but which comply to Definition 4 and are deemed to generate some strange languages. Examples of the latter kind were given by Kountouriotis et al. [62], who accordingly proposed a more advanced definition of Boolean grammars given below.

3.3. Definition by a well-founded fixpoint

Another definition of Boolean grammars, inspired by the well-founded semantics in logic programming, was given by Kountouriotis et al. [62]. According to this definition, a Boolean grammar is interpreted as a system of equations with unknown *three-valued languages*, and this system is guaranteed to have a certain kind of fixpoint, which is used to assign meaning to the grammar.

Three-valued languages are mappings from Σ^* to $\{0, \frac{1}{2}, 1\}$, where 1 and 0 indicate that a string definitely is or definitely is not in the language, while $\frac{1}{2}$ stands for "undefined". Equivalently, three-valued languages can be denoted by pairs (L, L') with $L \subseteq L' \subseteq \Sigma^*$, where L and L' represent a lower bound and an upper bound on a language that is not known precisely. A string in both L and L' definitely is in the language, a string belonging to neither of them definitely is not, and if a string is in L' but not in L, then its membership is not defined. In particular, if $L = L'$, then the language is completely defined, whereas a pair $(\varnothing, \Sigma^*)$ means a language about which nothing is known. The set of such pairs shall be denoted by 3^{Σ^*}.

Boolean operations and concatenation are generalized from two-valued to three-valued languages as follows:

$$\begin{aligned} (K, K') \cup (L, L') &= \big(K \cup L,\ K' \cup L'\big), \\ (K, K') \cap (L, L') &= \big(K \cap L,\ K' \cap L'\big), \\ \overline{(L, L')} &= (\overline{L'}, \overline{L}), \\ (K, K')(L, L') &= (KL, K'L'). \end{aligned}$$

Two different partial orderings on three-valued languages are defined. First, they can be compared with respect to the *degree of truth*:

$$(K, K') \sqsubseteq_T (L, L') \quad \text{if} \quad K \subseteq L \text{ and } K' \subseteq L'.$$

This means that whenever a string belongs to the lesser language, it must be in the greater language as well, and if the membership of a string in the lesser language is uncertain, then it must be either uncertain or true for the greater language.

The other ordering is with respect to the *degree of information*:

$$(K, K') \sqsubseteq_I (L, L') \quad \text{if} \quad K \subseteq L \text{ and } L' \subseteq K'.$$

It represents the fact that (K, K') and (L, L') are approximations of the same language, and that (L, L') is more precise, in the sense of having fewer uncertain strings. If a string is definitely known to belong or not to belong to the lesser language, then it must maintain the same status in the greater language, and if a string is uncertain in the lesser language, then the greater language may have any value of this string, that is, keep it as uncertain or define it as a member or a non-member.

Both orderings are extended to vectors of three-valued languages. The truth-ordering has a least element $\bot_T = \big((\varnothing, \varnothing), \ldots, (\varnothing, \varnothing)\big)$, that is, every language is completely defined as $\varnothing$; the greatest element is $\big((\Sigma^*, \Sigma^*), \ldots, (\Sigma^*, \Sigma^*)\big)$. For the information-ordering, the least element is $\bot_I = \big((\varnothing, \Sigma^*), \ldots, (\varnothing, \Sigma^*)\big)$, in which all languages are fully undefined. Fully defined languages are pairwise incomparable maximal elements of $\sqsubseteq_I$.

Now every Boolean grammar is represented by the system of equations

$$A = \bigcup_{\substack{A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\\ \&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_{m'} \in R}} \left[\bigcap_{i=1}^{m} \alpha_i \;\cap\; \bigcap_{j=1}^{m'} \overline{\beta_j} \right], \tag{6}$$

in which the unknowns $A \in N$ are three-valued languages, each symbol $a \in \Sigma^*$ represents a completely defined constant language $(\{a\}, \{a\})$, and empty concatenations represent constants $(\{\varepsilon\}, \{\varepsilon\})$. The task is to show that this system always has solutions in three-valued languages, and to construct a particular solution used to define the language generated by the grammar.

As in the two-valued case, concatenation, union and intersection, as well as every combination thereof, are monotone and continuous with respect to the truth ordering; complementation is not monotone. With respect to the information ordering, concatenation and all Boolean operations, *including complementation*, are monotone and continuous, which extends to any combinations of these operations.

These properties allow the following two-level fixpoint iteration. In the beginning, nothing is known about the membership of any strings, which is represented by a vector of languages $K^{(0)} = \bot_I$. Then, this knowledge is substituted into *all negative conjuncts* in the right-hand sides of (6), turning each equation into

$$A = \bigcup_{\substack{A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\\ \&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_{m'} \in R}} \left[\bigcap_{i=1}^{m} \alpha_i \;\cap\; const \right],$$

where the constant depends upon $K^{(0)}$ and all β_j. The resulting system has no negation, and therefore its least solution with respect to the truth-ordering $\sqsubseteq_T$ can be obtained by a fixpoint iteration. Then, the vector of three-valued languages in the limit is denoted by $K^{(1)}$, with $K^{(0)} \sqsubseteq_I K^{(1)}$.

Repeating the same process leads to an infinite sequence $\{K^{(\ell)}\}_{\ell \geqslant 0}$, which is monotone with respect to the information-ordering $\sqsubseteq_I$. The properties of this ordering can then be used to show that the limit of this sequence is a solution of the system (6). This solution defines the meaning of the original grammar.

Definition 5 (Kountouriotis et al. [62]). *Let $G = (\Sigma, N, R, S)$ be a Boolean grammar, let $N = \{A_1, \ldots, A_n\}$. Fix any vector of three-valued languages $K = ((K_1, K_1'), \ldots, (K_n, K_n')) \in (3^{\Sigma^*})^n$ and define a function $\Theta_K : (3^{\Sigma^*})^n \to (3^{\Sigma^*})^n$ by substituting its argument into positive conjuncts and K into negative conjuncts:*

$$[\Theta_K(L)]_A = \bigcup_{\substack{A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\\ \&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_{m'} \in R}} \left[\bigcap_{i=1}^{m} \alpha_i(L) \;\cap\; \bigcap_{j=1}^{m'} \overline{\beta_j(K)} \right]$$

for each $A \in N$. Furthermore, let

$$\Omega(K) = \bigsqcup_{\ell \geqslant 0} \Theta_K^{\ell}(\bot_T),$$

and let

$$M = \bigsqcup_{k \geqslant 0} \Omega^k(\bot_I),$$

where the least upper bounds are with respect to the truth-ordering $\sqsubseteq_T$ and the information-ordering $\sqsubseteq_I$, respectively. Then, define $L_G(A) = [M]_A$.

This definition by a two-level fixpoint iteration is effective in the same sense as the one-level fixpoint iteration (3) used for conjunctive grammars. For a conjunctive grammar, if a string w is in $L_G(A)$, then it appears at some k-th step as the A-component of $\varphi^k(\bot)$, and this number k is the height of the proof that w has the property A. In the case of the well-founded semantics, whenever a string w has a certain value in M, this value is assigned at some k-th step of the I-iteration $\Omega^k(\bot_I)$, where it emerges from some ℓ-th step of the T-iteration $\Theta_K^{\ell}(\bot_T)$ with $K = \Omega^{k-1}(\bot_I)$. The pair of numbers (k, ℓ) represents the decision point for the string w, and can be used in proofs by induction similar to the simple conjunctive case.

Consider the following basic example of a Boolean grammar that is clearly ill-formed in terms of two-valued languages. In three-valued languages, this grammar generates a language undefined on every string:

Example 8. *The Boolean grammar* $S \to \neg S$ *generates the three-valued language* $(\varnothing, \Sigma^*)$.

The second example illustrates how a well-founded fixpoint generalizes the standard least fixpoint used in the definition of conjunctive grammars, and specifies a single appropriate solution among infinitely many solutions of a system of equations.

Example 9. *According to Definition 5, the following Boolean grammar*

$$\begin{array}{lcl} S & \to & \neg A \\ A & \to & A \end{array}$$

has $L_G(S) = (\Sigma^*, \Sigma^*)$ *and* $L_G(A) = (\varnothing, \varnothing)$.

The grammars in Examples 8 and 9 do not satisfy the condition in the simpler Definition 4, because each of them has multiple solutions in two-valued languages. The next example presents a grammar that complies to Definition 4, but generates a counter-intuitive language according to it, while according to the well-founded semantics, the grammar generates a fully uncertain language.

Example 10 (Kountouriotis et al. [62]). *The Boolean grammar*

$$\begin{array}{lcl} S & \to & \neg S \& \neg A \\ A & \to & A \end{array}$$

generates the three-valued language $L(G) = (\varnothing, \Sigma^*)$ *according to Definition 5, with* $L_G(A) = (\varnothing, \varnothing)$. *On the other hand, Definition 4 inexplicably assigns* $L_G(S) = \varnothing$ *and* $L_G(A) = \Sigma^*$.

Finally, note that if a grammar generates a fully defined language $L(G) = (L, L)$ according to Definition 5, and at the same time has a strongly unique solution in two-valued languages, as in Definition 4, then this strongly unique solution defines $L(G) = L$. This stems from the fact that three-valued operations on languages preserve two-valued languages, and hence every solution of the system of two-valued equations (5) associated to a grammar is a solution of the system of three-valued equations (6).

The binary normal form for conjunctive grammars and for the simpler definition of Boolean grammars has a variant for the well-founded definition.

Theorem 6 (Kountouriotis et al. [62]). *Every Boolean grammar, as in Definition 5, can be effectively transformed to a grammar in the* ***binary normal form****, in which every rule is of the form*

$$A \to B_1 C_1 \,\&\, \ldots \,\&\, B_m C_m \,\&\, \neg D_1 E_1 \,\&\, \ldots \,\&\, \neg D_{m'} E_{m'} \,\&\, \neg\varepsilon \qquad (m \geqslant 1,\ m' \geqslant 0,\ B_i, C_i, D_j, E_j \in N)$$

$$A \to a \quad (a \in \Sigma)$$

$$A \to a \,\&\, \neg A \quad (a \in \Sigma)$$

$$S \to \varepsilon$$

Here a rule $A \to a \,\&\, A$ *generates an uncertain single symbol,* $(\varnothing, \{a\})$, *while the rule* $S \to \varepsilon$ *is allowed only if* S *does not appear in the right-hand sides of any rules.*

The transformation maintains the three-valued language generated by the grammar.

Some recent contributions to the three-valued theory of Boolean grammars include an equivalent characterization of Definition 5 by an infinite combinatorial game, due to Kountouriotis et al. [63], as well as a simpler variant of Definition 5 for a subclass of Boolean grammars, proposed by Nomikos and Rondogiannis [82].

A variant of the three-valued definition of Boolean grammars, in which languages are generalized to mappings from Σ^* to any Boolean algebra $\mathcal{B}$, was introduced by Ésik and Kuich [24]. Under their definition, partially known mappings are approximated by pairs of such mappings, in the same way as in the case of $\mathcal{B} = \{0, 1\}$ studied by Kountouriotis et al. [62]. Though Ésik and Kuich [24] use only the information ordering and consider only a single fixpoint iteration with respect to this ordering, their definition can apparently be extended to a combination of two orderings, as in Definition 5.

3.4. Parse trees and ambiguity

An important property of Boolean grammars inherited from the ordinary context-free grammars is the presentation of the syntax of a generated string in the form of a *parse tree*. Parse trees for conjunctive grammars were already mentioned in Example 1, and such a tree transcribes a complete proof that a string is generated by the grammar. In the case of a Boolean grammar, the known definition of a parse tree does not account for negative conjuncts, and represents a parse of a string according to positive conjuncts in the rules.

These are, strictly speaking, finite directed acyclic graphs rather than trees. A *parse tree* of a string $w = a_1 \ldots a_{|w|}$ from a nonterminal A has $|w|$ ordered leaves labelled with $a_1, \ldots, a_{|w|}$, and the rest of the vertices are labelled with rules from R. The subtree accessible from any given vertex of the tree contains leaves in the range between $i + 1$ and j, and thus corresponds to a substring $a_{i+1} \ldots a_j$. Each vertex labelled with a rule

$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n$$

and associated to a substring $a_{i+1} \ldots a_j$ has direct descendants corresponding to the symbols in positive conjuncts, labelled according to these symbols. Negative conjuncts

are not represented in the tree. For example, if the conjunctive grammar from Example 1 had the rule for S replaced with $S \to AB \,\&\, DC \,\&\, \neg AA$, then the parse tree of the string *abc* would still have the same form as in Figure 1. For each substring of w and for each $A \in N$, it is sufficient to have at most one subtree representing a parse of w from A, and hence the entire tree needs to have at most $|N| \cdot \frac{1}{2}|w|(|w|+1) + |w|$ nodes. A mathematically precise definition of a parse tree for a Boolean grammar can be found in the literature [103].

Ambiguity in ordinary context-free grammars can be defined in two ways. A grammar can be deemed unambiguous,

- if for every string generated by the grammar there is a unique parse tree (in other words, a unique leftmost derivation);
- or if for every nonterminal A and for every string $w \in L(A)$ there exists a unique rule $A \to s_1 \dots s_\ell$ with $w \in L(s_1 \dots s_\ell)$, and a unique partition $w = u_1 \dots u_\ell$ with $u_i \in L(s_i)$.

Assuming that $L(A) \neq \varnothing$ for every nonterminal A, these definitions are equivalent. In the case of Boolean grammars, the first definition becomes useless, because negative conjuncts are not accounted for in a parse tree, and the requirement of parse tree uniqueness can be trivially satisfied by employing double negation on top of a grammar [103].

A sound definition of an unambiguous Boolean grammar follows the second approach, and takes into account partitions of strings according to both positive and negative conjuncts.

Definition 6 (Okhotin [103]). *A Boolean grammar $G = (\Sigma, N, R, S)$ is unambiguous, if*

I. *the choice of a rule for every single nonterminal A is unambiguous, in the sense that for every string w, there exists at most one rule*

$$A \to \alpha_1 \,\&\, \dots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \dots \,\&\, \neg\beta_n,$$

with $w \in L_G(\alpha_1) \cap \dots \cap L_G(\alpha_m) \cap \overline{L_G(\beta_1)} \cap \dots \cap \overline{L_G(\beta_n)}$ (in other words, different rules generate disjoint languages), and

II. *all concatenations are unambiguous, that is, for every conjunct $\pm s_1 \dots s_\ell$ that occurs in the grammar, and for every string w, there exists at most one partition $w = u_1 \dots u_\ell$ with $u_i \in L_G(s_i)$ for all i.*

*A language L can be called **inherently ambiguous** with respect to conjunctive (Boolean) grammars, if it is defined by some conjunctive (Boolean) grammar, but every conjunctive (Boolean) grammar generating L is ambiguous.*

The unambiguous concatenation requirement applies to positive and negative conjuncts alike. For a positive conjunct belonging to some rule, this means that a string that is potentially generated by this rule must be uniquely split according to this conjunct. For a negative conjunct $\neg DE$, this condition requests that a partition of $w \in L_G(DE)$ into $L_G(D) \cdot L_G(E)$ is unique, even though w is *not generated* by any rule involving this conjunct. Though the latter condition might appear unnecessary, it is in fact essential, because one can transform a given grammar to move all ambiguous concatenations into negative conjuncts [103].

Though, as mentioned above, the uniqueness of a parse tree does not guarantee that a grammar is unambiguous, the converse holds. That is, for every unambiguous Boolean grammar $G = (\Sigma, N, R, S)$, for every nonterminal $A \in N$ and for every string $w \in L_G(A)$, the parse tree of w from A is unique (assuming that only vertices labelled with the symbols of the alphabet may have multiple incoming arcs).

The ambiguity of the choice of a rule can be effectively eliminated in a Boolean grammar by supplying every rule with an additional conjunct that expresses the condition of non-representability by the rest of the rules for this nonterminal [103]. For example, two rules $A \to \alpha \mid \beta$ can be replaced with the rules $A \to \alpha \mid \beta \,\&\, \neg\alpha$. A stronger statement is given in the following normal form theorem.

Theorem 7 (Okhotin [103]). *For every Boolean grammar, as in Definition 4, there exists and can be effectively constructed a Boolean grammar generating the same language, which is in the binary normal form, as defined in Theorem 5, and in which the choice of a rule is unambiguous. Furthermore, if the original grammar had unambiguous concatenation, then so does the constructed grammar.*

For conjunctive grammars, the earlier Theorem 3 due to Okhotin and Reitwießner [110] implies that every conjunctive grammar can be transformed to a conjunctive grammar with unambiguous choice of a rule. However, the transformation introduces ambiguity of concatenation.

Consider some examples. The grammar in Example 1 is unambiguous, and so is the similar grammar in Example 6. The grammar in Example 2 is unambiguous as well. On the other hand, the grammar in Example 5 is ambiguous, because the concatenations AB and OOO are both ambiguous: for instance, $aaa \cdot babab$ and $aaaba \cdot bab$ are two partitions of a single string into $L(A)L(B)$. The grammar in Example 7 is also ambiguous, because this is basically an ambiguous ordinary context-free grammar with a negation on top; it is not known whether this language is generated by any unambiguous Boolean grammar.

The grammar in Example 4 is definitely ambiguous, because a concatenation of a language over a one-symbol alphabet with itself is always ambiguous, as long as this language has at least two elements. However, there exists a slightly more complicated grammar generating the same language using the same general method.

Example 11 (Jeż, Okhotin [57]). *The following conjunctive grammar is unambiguous, and it generates the*

language $\{a^{4^n} \mid n \geqslant 0\}$.

$$\begin{array}{rcl} A_1 & \to & A_1A_3 \,\&\, A_7A_9 \mid a \mid a^4 \\ A_2 & \to & A_1A_7 \,\&\, A_2A_6 \mid a^2 \\ A_3 & \to & A_1A_2 \,\&\, A_3A_9 \mid a^3 \\ A_6 & \to & A_1A_2 \,\&\, A_9A_{15} \mid a^6 \\ A_7 & \to & A_1A_3 \,\&\, A_1A_6 \\ A_9 & \to & A_1A_2 \,\&\, A_2A_7 \\ A_{15} & \to & A_6A_9 \,\&\, A_2A_7 \end{array}$$

Each nonterminal A_i *generates the language* $L_G(A_i) = \{a^{i \cdot 4^n} \mid n \geqslant 0\}$.

4. Grammars with linear concatenation

A special case of ordinary context-free grammars, which can express a concatenation of a nonterminal symbol only with terminal strings, is known as a *linear context-free grammar*. In such grammars, every rule $A \to \alpha$ has $\alpha \in \Sigma^* \cup \Sigma^* N \Sigma^*$. These grammars are notable for their lower computational complexity and other noteworthy properties.

Similarly to the case of grammars with disjunction only, a conjunctive grammar is called *linear conjunctive*, if every rule it contains is either of the form $A \to u_1 B_1 v_1 \,\&\, \ldots \,\&\, u_n B_n v_n$ with $n \geqslant 1$, $u_i, v_i \in \Sigma^*$ and $B_i \in N$, or of the form $A \to w$ with $w \in \Sigma^*$. The grammar in Example 2 is linear; the grammar in Example 1 is not, yet it can be straightforwardly changed to make it linear. The grammar in Example 4 essentially relies on concatenating nonterminals to each other, and has no equivalent linear conjunctive grammar.

4.1. Representation by trellis automata

The family of languages defined by linear conjunctive and linear Boolean grammars had actually been known for twenty years before these grammars were introduced. This is the family defined by one of the simplest types of cellular automata: the *one-way real-time cellular automata*, also known under the proper name of *trellis automata*, studied by Dyer [21], Čulík, Gruska and Salomaa [19, 20], Ibarra and Kim [47], and others.

A trellis automaton, defined as a quintuple $(\Sigma, Q, I, \delta, F)$, processes an input string of length $n \geqslant 1$ using a uniform triangular array of $\frac{n(n+1)}{2}$ processor nodes, connected as in Figure 2. Each node computes a value from a fixed finite set Q. The nodes in the bottom row obtain their values directly from the input symbols using a function $I : \Sigma \to Q$. The rest of the nodes compute the function $\delta : Q \times Q \to Q$ of the values in their predecessors. The string is accepted if and only if the value computed by the topmost node belongs to the set of accepting states $F \subseteq Q$.

Evidently, trellis automata are one of the simplest computational models one can imagine, and they were proved to be computationally equivalent to linear conjunctive grammars:

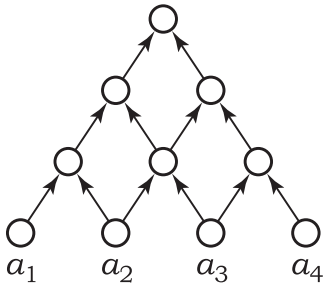

Figure 2: The form of a computation of a trellis automaton.

Theorem 8 (Okhotin [92]). *A language* $L \subseteq \Sigma^+$ *is generated by a linear conjunctive grammar if and only if* L *is recognized by a trellis automaton.*

The theorem is proved by effective constructions in both directions. Consider a linear conjunctive grammar $G = (\Sigma, N, R, S)$, in which every rule is of the form

$$A \to bB_1 \,\&\, \ldots \,\&\, bB_m \,\&\, C_1c \,\&\, \ldots \,\&\, C_nc \quad (b, c \in \Sigma,\ B_i, C_j \in N) \tag{7}$$

or $A \to a$ with $a \in \Sigma$; every linear conjunctive grammar can be converted to this form [83] by a transformation similar to the one in Theorem 2. For such a grammar, there is an equivalent trellis automaton with the set of states $Q = \Sigma \times 2^N \times \Sigma$, which, given a string w, computes the triple formed by (i) the first symbol of w, (ii) the set of nonterminals in N generating w, and (iii) the last symbol of w. Accordingly, the initial function of this automaton is defined as $I(a) = (a, \{A | A \to a \in R\}, a)$, a state (a, X, b) is set to be accepting if $S \in X$, and the transition from every pair of states (b, X, c') and (b', Y, c) leads to the following state:

$$\begin{aligned} \delta((b, X, c'), (b', Y, c)) = \big(b,\ \{A \mid &\text{there is a rule (7)} \\ &\text{with } C_1, \ldots, C_n \in X \text{ and } B_1, \ldots, B_m \in Y\},\ c\big). \end{aligned}$$

A direct construction applicable to an arbitrary linear conjunctive grammar is also known [91].

Conversely, any trellis automaton $M = (\Sigma, Q, I, \delta, F)$ can be simulated by a linear conjunctive grammar G with the set of nonterminals $N = \{A_q \mid q \in Q\} \cup \{S\}$, where $L_G(A_q)$ is the set of all strings in Σ^+, on which M computes the state q. Every transition $q = \delta(q_1, q_2)$ is then represented by the following $|\Sigma|^2$ rules:

$$A_q \to bA_{q_2} \,\&\, A_{q_1}c \quad (\text{for all } b, c \in \Sigma).$$

There are also the rules $A_{I(a)} \to a$ for each $a \in \Sigma$, and $S \to A_q$ for all $q \in F$. It is worth noting, that this grammar is unambiguous, which leads to the following small result.

Corollary 8.1 ([103, Thm. 4]). *For every linear conjunctive grammar, there exists an unambiguous linear conjunctive grammar generating the same language.*

The transformation of a trellis automaton to a grammar can be done in a different way, which minimizes the number of nonterminals, at the expense of having an enormous number of rules. This construction is described in Theorem 21 in Section 7.2.

Analogously to linear conjunctive grammars, one can define a *linear Boolean grammar*, with all rules of the form $A \to \pm u_1 B_1 v_1 \,\&\, \ldots \,\&\, \pm u_n B_n v_n$ or $A \to w$. However, linear Boolean grammars can be simulated by trellis automata [94], and therefore are equivalent in power to linear conjunctive grammars.

4.2. Examples

Trellis automata are occasionally useful for representing languages, for which no convenient grammar is known. Consider the *Dyck language* of balanced brackets, typically defined by an ordinary context-free grammar $S \to aSb \mid SS \mid \varepsilon$. It is linear conjunctive, because there exists the following automaton.

Example 12 (Dyer [21, Thm. 3]). *The Dyck language is recognized by a trellis automaton* $(\{a,b\}, Q, I, \delta, F)$*, where*

$$\begin{aligned} Q &= \{\nearrow, \nwarrow, X, -\}, \\ I(a) &= \nearrow, \\ I(b) &= \nwarrow, \\ F &= \{X\} \end{aligned}$$

δ	$\nearrow$	$\nwarrow$	X	$-$
$\nearrow$	$\nearrow$	X	$\nearrow$	$\nearrow$
$\nwarrow$	$-$	$\nwarrow$	$-$	$-$
X	$-$	$\nwarrow$	$-$	$\nearrow$
$-$	$-$	$\nwarrow$	$\nwarrow$	$-$

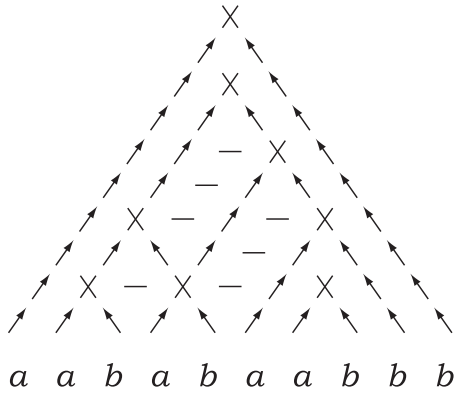

Figure 3: A sample computation of the trellis automaton for the Dyck language given in Example 12.

A linear conjunctive grammar for the Dyck language, obtained from this example by applying the construction in Theorem 8, can be found in the literature [92, Ex. 2].

The next example illustrates the ability of trellis automata to count in positional notation.

Example 13 (Ibarra, Kim [47, Ex. 2.1]). *The language* $\{a^n b^{2^n} \mid n \geqslant 1\}$ *is linear conjunctive.*

On a string $a^i b^j$ with $i, j \geqslant 1$, a trellis automaton recognizing this language computes the i-th bit of the base-2 representation of j. Some further states are used to represent the carry digit, and to keep track of whether j is less than, equal to, or greater than 2^i.

The language in the following example had once been proposed as a candidate language for having no trellis automaton [47]. Surprisingly, a sophisticated trellis automaton recognizing this language was constructed.

Example 14 (Čulík [18]). *The language* $\{a^m b^{m+n} a^n \mid m, n \geqslant 1\}$ *is linear conjunctive.*

The construction embeds a cellular automaton solving the *firing squad synchronization problem* [8] into a trellis automaton.

4.3. Limitations

A general method for proving non-representability of particular languages by trellis automata was discovered by Terrier [134]. It is based upon a special complexity function of a language, which reflects the number of cases that the trellis automaton needs to distinguish in the last few levels of its computation.

Definition 7. *Let* $L \subseteq \Sigma^*$ *be a language, let* $k \geqslant 1$ *and let* $w = a_1 \ldots a_n$ *be a string with* $n \geqslant k$*. Define a set*

$$S_{L,k,w} = \{(i,j) \mid i, j \geqslant 0,\ i + j < k,\ a_{i+1} \ldots a_{n-j} \in L\},$$

which represents the membership in L *of all substrings of* w *longer than* $|w| - k$ *symbols. Next, define the set of all sets* $S_{L,k,w}$ *for all strings* w*:*

$$\widehat{S}_{L,k} = \{S_{L,k,w} \mid w \in \Sigma^*,\ |w| \geqslant k\}.$$

Let $f_L(k) = |\widehat{S}_{L,k}|$.

Each set $S_{L,k,w}$ has between 0 and $\frac{k(k+1)}{2}$ elements, and accordingly, the cardinality of the set $\widehat{S}_{L,k}$ is between 1 (if $L \cap \Sigma^+$ is $\varnothing$ or Σ^+) and $2^{\frac{k(k+1)}{2}}$ (if $\widehat{S}_{L,k}$ contains all subsets of $\{(i,j) \mid i, j \geqslant 0,\ i + j < k\}$).

This measure exposes the following limitation of linear conjunctive languages: the growth rate of their "complexity" function $f_L(k)$ cannot get as high as $2^{\Theta(k^2)}$, for the reason that the acceptance decisions on all $\frac{k(k+1)}{2}$ long substrings of the input are determined by k states of a trellis automaton computed on k substrings of length $|w| - k + 1$, and there are only $2^{O(k)}$ combinations of these states.

Theorem 9 (Terrier [134]). *If* $L \in \Sigma^*$ *is linear conjunctive, then there exists a number* p*, such that*

$$f_L(k) \leqslant p^k.$$

Theorem 9 was followed by an example of an ordinary context-free language that maximizes this complexity measure, and hence is recognized by no trellis automaton:

Example 15 (Terrier [134]). *The language*

$$L = \{a^{i_1}b^{j_1}\dots a^{i_m}b^{j_m} \mid m \geqslant 2;\ i_t, j_t \geqslant 1, \\ \exists \ell:\ i_1 = j_\ell \text{ and } i_{\ell+1} = j_m\}$$

is generated by an ordinary context-free grammar. But it has $f_L(k) = 2^{\frac{k(k+1)}{2}}$*, and therefore is not linear conjunctive.*

The language L in Example 15 is inherently ambiguous [103], that is, every ordinary context-free grammar generating it must be ambiguous. In plain words, this is one of the more complicated context-free languages. Theorem 9 can also be used to show that some simpler languages are not linear conjunctive.

Example 16 (Okhotin [107]). *The language*

$$L = \{c^m a^{\ell_0} b \dots a^{\ell_{m-1}} b a^{\ell_m} b \dots a^{\ell_z} b d^n \mid \\ m, n, \ell_i \geqslant 0,\ z \geqslant 1,\ \ell_m = n\}$$

is generated by an LL(1) context-free grammar. However, $f_L(k) \geqslant (k+1)! = 2^{\Theta(k \log k)}$*, and therefore* L *is not linear conjunctive.*

Some further limitations of linear conjunctive languages are known for languages of a special form. Every linear conjunctive language over a one-symbol alphabet $\Sigma = \{a\}$ is regular, because all states on the same horizontal level in Figure 2 are identical. The following two results show some limitations of trellis automata operating on strings from a^*b^*.

Example 17 (Yu [144]). *The language* $\{a^n b^{i \cdot n} | n, i \geqslant 1\}$ *is not linear conjunctive.*

Theorem 10 (Buchholz, Kutrib [13]). *For every function* $f\colon \mathbb{N} \to \mathbb{N}$*, if the language* $\{a^n b^{f(n)} \mid n \geqslant 1\}$ *is linear conjunctive, then* f *is bounded by an exponential function.*

Both results are proved by showing that every trellis automaton operating on strings of this form demonstrates periodic behaviour of a certain kind.

5. Basic parsing algorithms

Parsing means decomposing a string into substrings according to a grammar, and verifying that it is a well-formed sentence. Given a string as an input, a parsing algorithm should determine whether the string belongs to the language described by a fixed (or a given) grammar, and if it does, construct a *parse tree* of the string, as it is defined by the grammar.

5.1. Cubic-time tabular parsing

The simplest parsing method applicable to all context-free languages is known as the Cocke–Kasami–Younger algorithm [59, 143]. Though not entirely practical as it is, due to the normal form requirement, this algorithm is important as the mathematical germ of other more practical algorithms, and as an easy proof that every ordinary context-free language is recognizable in time $O(n^3)$.

The Cocke–Kasami–Younger algorithm extends to Boolean grammars in a completely seemless way. It requires a Boolean grammar in binary normal form; given an input string $w = a_1 \dots a_n$, the algorithm constructs an $n \times n$ table T of sets of nonterminals, with

$$T_{i,j} = \{A \in N \mid a_{i+1} \dots a_j \in L_G(A)\}$$

for all $0 \leqslant i < j \leqslant n$. The elements of this table are calculated inductively on the length $j - i$ of the substring, beginning with the elements $T_{i,i+1}$, each depending only on the symbol a_{i+1}, and continuing with larger and larger substrings, until the element $T_{0,n}$ representing the entire string is computed. The induction step is given by the equality

$$T_{i,j} = f\Big(\bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j} \Big),$$

where the function $f\colon 2^{N \times N} \to 2^N$ is defined by $A \in f(U)$ if and only if there is a rule $A \to B_1C_1 \,\&\, \dots \,\&\, B_mC_m \,\&\, \neg D_1E_1 \,\&\, \dots \,\&\, \neg D_{m'}E_{m'} \,\&\, \neg\varepsilon$ with $(B_t, C_t) \in U$ and $(D_t, E_t) \notin U$ for all applicable t. In total, there are $\Theta(n^2)$ elements, and each of them takes $\Theta(n)$ operations to compute, which leads to an obvious cubic-time algorithm [83, 94].

Once the algorithm verifies that the input string is generated by the grammar, the table $T_{i,j}$ can be used to construct a parse tree of the string [37, Alg. 12.4.3], in time proportional to $|G| \cdot t \cdot n$, where t is the number of nodes in the resulting tree. Thus, the complexity is again asymptotically bounded by $|G| \cdot n^3$.

Several variants of this algorithm are known. Firstly, one can extend its applicability by relaxing the normal form requirement. To this end, the algorithm is changed to manipulate conjuncts with marked positions instead of ordinary nonterminals, and accordingly use more elaborate operations on sets of such items. For ordinary context-free grammars, this leads to the well-known algorithms by Earley [23] and by Graham, Harrison and Ruzzo [33]. The Graham–Harrison–Ruzzo algorithm was extended to context-free grammars with Boolean operations at the top level by Heilbrunner and Schmitz [39], and to conjunctive grammars of the general form by the author [89]. A variant of this algorithm for Boolean grammars with restricted use of negation and with weak normal form conditions was given by Wrona [142].

Another variant of the above algorithm for Boolean grammars in binary normal form defined by well-founded

fixpoints was devised by Kountouriotis et al. [62]: this algorithm computes the three-valued membership of a given string in the language.

5.2. Square-time parsing for unambiguous grammars

The basic cubic-time parsing algorithm for Boolean grammars has another variant, that works in square time on any unambiguous grammar. This performance increase is achieved by using a different data structure to represent the same table $T \in (2^N)^{n \times n}$, with

$$T_{i,j} = \{A \in N \mid a_{i+1} \dots a_j \in L_G(A)\}.$$

The new data structure is a two-dimensional table T', indexed by positions in the input and nonterminals. Each entry of this table holds a set of positions in the input string, which are stored as a list in an ascending order. The element corresponding to a position j, with $1 \leqslant j \leqslant n$, and a nonterminal $A \in N$ is denoted by $T'_j[A]$. Its intended value is

$$T'_j[A] = \{i \mid a_{i+1} \dots a_j \in L_G(A)\},$$

that is, $T'_j[A]$ should contain the initial positions of all substrings generated by A, which end at the k-th position. Then, accordingly, $i \in T'_j[A]$ if and only if $A \in T_{i,j}$. The entire string $a_1 \dots a_n$ is in $L(G)$ if and only if the position 0 is in $T'_n[S]$.

The algorithm processes the input string from left to right: after reading every next symbol a_j, it calculates the sets $T'_j[A]$ for all $A \in N$. For each j, this is done along with determining all concatenations $a_{i+1} \dots a_k \cdot a_{k+1} \dots a_j$, where $a_{i+1} \dots a_k \in L_G(B)$ and $a_{k+1} \dots a_j \in L_G(C)$ for some positions i, k and for some conjunct $\pm BC$ in the grammar. These concatenations are then stored in the variables $U_{i,j} \subseteq N \times N$, which are of the same kind as the argument of the function f in Section 5.1. Based on the conjuncts accumulated in $U_{i,j}$, the parser will eventually decide to insert the element i in the lists $T'_j[A]$, for any $A \in N$.

The key idea of the algorithm is the particular way, in which these sets $U_{i,j}$ are filled. The algorithm performs lookups of the following form: in search for a conjunct $\pm BC$, first, it traverses the list $T'_j[C]$ and reads each position k from this list; secondly, for every such position k, it traverses the list $T'_k[B]$ and considers each position i in there; and finally, for every such i, it inserts the pair (B, C) into $U_{i,j}$. In this way, if the lists for a particular input string are sparsely populated, then the algorithm will have to make only as few steps as the number of actual concatenations, rather than look for concatenations in all possible places. And if the grammar is unambiguous, one can prove that there are only $O(n^2)$ concatenations in total, and that the statement in the innermost loop of the algorithm shall be executed at most $O(n^2)$ times.

For ordinary context-free grammars, a similar algorithm was first proposed by Kasami and Torii [60]. The more well-known algorithm by Earley [23] is also known to work in square time, if the grammar is unambiguous. However, both algorithms do not use an intermediate data structure $U_{i,j}$, and write each concatenation they find directly to the equivalents of the sets T'. This would not work already for conjunctive grammars, where the conjuncts determined at a different time must be brought together to apply a rule of the grammar. Accordingly, the algorithm for Boolean grammar has to have a slightly different data flow.

Theorem 11 (Kasami, Torii [60]; Okhotin [103]). *For every Boolean grammar $G = (\Sigma, N, R, S)$ in binary normal form, there exists an algorithm, which, given an input string $w = a_1 \dots a_n$, constructs the sets $T'_j[A] = \{i \mid 0 \leqslant i < j,\ a_{i+1} \dots a_j \in L_G(A)\}$, for all $A \in N$ and $j \in \{1, \dots, n\}$. The algorithm works in time $O(n^3)$. If the concatenation in the grammar is unambiguous, it works in time $O(n^2)$.*

5.3. Generalized LR parsing

The *deterministic LR(k)* parsing method, introduced by Knuth [61], is applicable to a subclass of ordinary context-free grammars. An LR(k) parser reads the string from left to right, using a stack memory, and its configuration is a pair of the stack contents $\alpha \in (\Sigma \cup N)^*$ and the unread suffix v of the input. When the parser is in a configuration (α, v), where $w = uv$ is the entire input string, this means that it has already parsed the earlier part of the input u, and found its representation as the concatenation α, with $u \in L_G(\alpha)$.

A deterministic LR parser may use two operations: (i) shifting the next input symbol to the stack, and (ii) reducing a right-hand side of a rule $A \to \eta$ at the top of the stack to a single symbol A:

$$(\alpha, av) \xrightarrow{\text{Shift } a} (\alpha a, v)$$
$$(\alpha\eta, v) \xrightarrow{\text{Reduce } A \to \eta} (\alpha A, v)$$

In order to decide, which operation to apply in a configuration, a deterministic LR parser may use the next k symbols of the input, where k is fixed, and the state computed by a certain DFA that processes the stack from the bottom to the top. Many methods for constructing suitable DFAs are known [1, 61]. In an implementation, the parser stores $|\alpha|$ intermediate states of this DFA in the stack, instead of the symbols from α, and simulates only one transition of the DFA per each shift or reduction. If the operation to be applied cannot be deterministically decided in some configuration, then the deterministic LR parsing is not applicable to this grammar.

Generalized LR parsing, first proposed by Lang [65] and later independently discovered and developed by Tomita [136], is a polynomial-time method of simulating nondeterminism in the deterministic LR. Every time a deterministic LR parser has to choose an action to perform (to *shift* an input symbol or to *reduce* by one or another rule), a generalized LR parser performs both actions, storing all possible contents of an LR parser's stack in the

form of a graph, which contains $O(n)$ vertices and therefore fits in memory of size $O(n^2)$. If carefully implemented, the algorithm is applicable to every ordinary context-free grammar, and its complexity can be bounded by a polynomial of a degree as low as cubic. It can be straightforwardly extended to conjunctive grammars [85], while maintaining its cubic-time complexity; a further extension to Boolean grammars requires more significant modifications, and runs in time $O(n^4)$. The main advantage of this algorithm over those described in Sections 5.1–5.2 is that it works much faster on "good" grammars: for instance, in linear time on the Boolean closure of the deterministic context-free languages [85].

The Generalized LR uses a *graph-structured stack* to represent the contents of the linear stack of an ordinary LR parser in all possible branches of a nondeterministic computation. This is a directed graph with a designated *source node*, representing the bottom of the stack. Each arc of the graph is labelled with a symbol from $\Sigma \cup N$. The nodes are labelled with the states of a DFA processing the labels on the path from the source node. This would typically be one of the DFAs defined for the Deterministic LR, and it is used merely to help the parser handle at least some decisions deterministically; if a trivial one-state automaton is used, the algorithm degrades to an inconvenient variant of a tabular parser. In particular, the source node is labelled with the initial state. There is a non-empty collection of designated nodes, called *the top layer* of the stack. Every arc leaving one of these nodes has to go to another node in the top layer. The labels of these nodes should be pairwise distinct, and hence the number of top layer nodes is bounded by a constant.

Initially, the stack contains a single source node, which at the same time forms the top layer. The computation of the algorithm is an alternation of *reduction phases*, in which the arcs going to the top layer are manipulated without consuming the input, and *shift phases*, where a single input symbol is read and consumed, and a new top layer is formed as a successor of the former top layer.

The shift phase is carried out as illustrated in Figure 4. Let a be the next input symbol. For each top layer node labelled with a state q, the algorithm follows the transition of the DFA from q by the symbol a. If the transition leads to a certain state q', a node labelled with this state is created in the new top layer. If it is undefined, this branch of the graph-structured stack is removed.

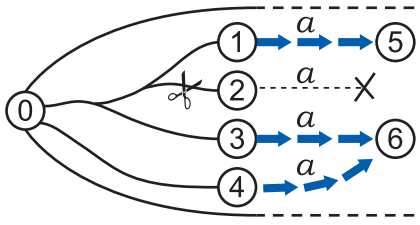

Figure 4: The shift phase in the Generalized LR parsing algorithm.

The reduction phase is a sequence of additions and removals of arcs labelled with nonterminals and leading to the top layer. Assume that the grammar has a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n$. Then, whenever the graph contains a node q, which is connected to any nodes in the top layer by the paths $\alpha_1, \ldots, \alpha_m$, and is not connected to the top layer by any of the paths $\beta_1, \ldots, \beta_n$, the parser may perform a *reduction* by this rule, adding an arc labelled A from q to a node in the top layer labelled with the appropriate state of the DFA. This is illustrated in Figure 5.

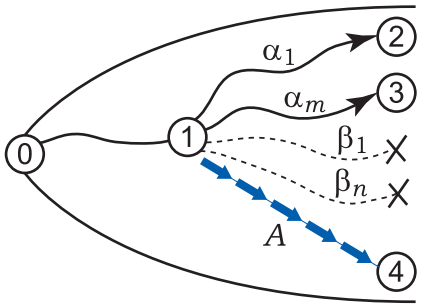

Figure 5: Reduction by a rule $A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_m \,\&\, \neg\beta_1 \,\&\, \ldots \,\&\, \neg\beta_n$ in the Generalized LR algorithm for Boolean grammars.

If the grammar is conjunctive, the condition for doing a reduction is existence of paths $\alpha_1, \ldots, \alpha_m$, and no arcs are ever removed during the reduction phase. Hence, these paths will stay in the graph. For a general Boolean grammar with negation, any of the paths $\beta_1, \ldots, \beta_n$, that were absent from the stack at the moment of the reduction, may appear later as a result of some other reductions, and then the earlier added arc labelled A might no longer be justified (unless there is another rule that warrants its existence). In such a case, the parser is entitled to *invalidate* the earlier reduction, removing the unjustified arc.

The input string is accepted, if the last reduction phase (the one after shifting the last input symbol) produces an arc labelled S from the source node to the top layer, which represents a parse of the entire string according to the grammar. The algorithm works correctly—that is, accepts if and only if the string is in the language—for every conjunctive grammar. It also works correctly for most Boolean grammars [97], although a grammar expressing a contradiction, such as $S \to \neg S$, would bring a parser into an infinite cycle of reductions and invalidations. The worst-case running time of the algorithm depends on how efficiently the operations on the graph are implemented. For conjunctive grammars, it can be made to work in time $O(n^3)$ [85], while for Boolean grammars, the best upper bound is $O(n^4)$ [97]. However, one of the main benefits of this algorithm is that it works much faster on "better" grammars: for instance, in linear time on Boolean combinations of LR(1) context-free grammars [85].

Besides recognizing the membership of an input, the algorithm can construct its parse tree in course of its computation. It is sufficient to augment the representation of

each arc in the graph with a pointer to the corresponding subtree. A larger subtree shall be constructed with each reduction, while each invalidation would require garbage collection.

Two implementations of the algorithm exist: there is an impractical reference implementation made by the author [86], which is suitable mainly for research purposes, and there is also a more useful Java-based parser generator developed by Megacz [79].

5.4. Recursive descent

The *recursive descent* is likely the simplest and the most well-known parsing method, which has been in use since the early 1960s. It is applicable to a subclass of ordinary context-free grammars, known as the *LL(k) grammars*, where k is the number of lookahead symbols [75]. This method was extended to conjunctive and Boolean grammars [84, 101], and is applicable to their appropriately defined LL(k) subclasses.

A recursive descent parser is a program containing a procedure for every terminal and nonterminal symbol used in the grammar. These procedures have access to the input string $w = a_1 a_2 \dots a_{|w|}$ and to a positive integer p pointing at the current position in this string.

The code for each procedure $a()$, with $a \in \Sigma$, simply checks that the next input symbol is a, and advances p to the next position. For each $A \in N$, the procedure $A()$ begins with deterministically choosing one of the rules for A, using the next k input symbols. Once a rule $A \to X_1 \dots X_\ell$ is chosen, the subsequent code $X_1(); \dots X_\ell();$ parses the following substring according to this rule, using the corresponding procedures to parse its substrings. Each procedure $X_i()$ advances the position in the input string, so that the next procedure deals with a subsequent substring. In total, $A()$ advances the position in ℓ steps, consuming a substring generated by A.

The extension of recursive descent parsing to conjunctive grammars [84] implements the conjunction by scanning a single substring multiple times. If a parser chooses a rule $A \to X_1 \dots X_\ell \,\&\, Y_1 \dots Y_m$, its computation begins by storing the current position in the input in a local variable. Then it invokes the code $X_1(); \dots X_\ell();$ and thus parses the substring according to the first conjunct. Next, it stores the final position in the string in another local variable, rewinds the pointer to the stored initial position and parses the string according to the second conjunct, using the code $Y_1(); \dots Y_m();$. Once the last of these procedures returns, the parser checks whether the final position is the same as the one after parsing the first conjunct. If they are different, an error is reported. This ensures that the parser has obtained two parses of a single substring.

In a recursive descent parser for a Boolean grammar [101], the negation is implemented using the mechanism of exception handling. First of all, the parser reports every error by raising an exception, and then the runtime environment of the programming language proceeds with unwinding the stack, until a frame with an exception handler is found, and passes control to the handler. Exceptions are handled by negative conjuncts. The code for a negative conjunct $\neg Y_1 \dots Y_m$ encloses the code for a positive conjunct $Y_1 \dots Y_m$ within an exception handler. If the enclosed code reports a syntax error, the exception handler regards this conjunct as successfully checked, and proceeds to checking the next conjunct of this rule (or if this was the last conjunct in the rule, the procedure sets the pointer to the stored final position and returns). However, if the enclosed code successfully terminates, and thus reports a parse of the current substring as $Y_1 \dots Y_m$, this means a syntax error in parsing this substring as A, and an exception is raised.

The algorithm can be naturally modified to construct a parse tree of the string. Whenever an instance of a procedure $A()$ successfully returns, consuming a substring u of the input, it can return a parse tree of u from A in its return value. This parse tree is constructed from the subtrees returned by the procedures invoked by $A()$.

For a grammar to be used with the recursive descent, it should, first of all, have no left recursion: that is, an attempt to parse a string w according to $A \in N$ should never require parsing a prefix of w according to the same A. Such a left recursion would require the procedure $A()$ to call itself without consuming any input symbols, thus going into an infinite loop. For example, recursive descent parsing is not applicable to the grammar $A \to Aa \mid \varepsilon$. In the general case of Boolean grammars, this condition is defined as follows.

Definition 8 ([101]). *Let $G = (\Sigma, N, R, S)$ be a Boolean grammar, and let the conjunctive grammar $G_+ = (\Sigma, N, R_+, S)$ be defined by removing all negative conjuncts from every rule in G. The grammar G is said to be* strongly non-left-recursive*, if for every recursive dependence by any rules*

$$\begin{aligned} A_0 &\to \dots \,\&\, \pm\theta_1 A_1 \eta_1 \,\&\, \dots, \\ &\vdots \\ A_{m-2} &\to \dots \,\&\, \pm\theta_{m-1} A_{m-1} \eta_{m-1} \,\&\, \dots, \\ A_{m-1} &\to \dots \,\&\, \pm\theta_m A_0 \eta_m \,\&\, \dots, \end{aligned}$$

where $\theta_i, \eta_i \in (\Sigma \cup N)^*$, it holds that $\varepsilon \notin L_{G_+}(\theta_1 \dots \theta_m)$.

If a grammar does not satisfy Definition 8, this leaves open the possibility of the procedures $A_0()$, $A_1()$, …, A_{m-1}, $A_0()$ recursively calling each other without consuming any input symbols, thus producing an infinite branch in the tree of recursive calls.

For every strongly non-left-recursive grammar, the corresponding system of language equations has a strongly unique solution [101], and therefore, Definition 8 already rules out the problematic cases in the definition of Boolean grammars. Assuming this property, it remains to ensure that the rule for a nonterminal can be unambiguously determined by examining only k next input symbols.

Definition 9 ([101]). *Let $G = (\Sigma, N, R, S)$ be a strongly non-left-recursive Boolean grammar. A string $v \in \Sigma^*$ is said to follow $\sigma \in (\Sigma \cup N)^*$, if there is a sequence of rules*

$$\begin{aligned} S &\to \dots \& \pm\theta_1 A_1 \eta_1 \& \dots, \\ A_1 &\to \dots \& \pm\theta_2 A_2 \eta_2 \& \dots, \\ &\vdots \\ A_{m-2} &\to \dots \& \pm\theta_{m-1} A_{m-1} \eta_{m-1} \& \dots, \\ A_{m-1} &\to \dots \& \pm\theta_{i+1} \sigma \eta_{i+1} \& \dots, \end{aligned}$$

with $\theta_i, \eta_i \in (\Sigma \cup N)^*$, such that $v \in L_G(\eta_m \dots \eta_2 \eta_1)$.

The grammar is said to be LL(k) for $k \geqslant 1$, if there exists such a function $T_k \colon N \times \Sigma^{\leqslant k} \to R \cup \{-\}$, that for every rule $A \to \varphi$, for every string $u \in L_G(\varphi)$ and for every string v that follows A, the value of $T_k(A, (\text{the first } k \text{ symbols of } uv))$ is $A \to \varphi$.

The correctness of the recursive descent is established as follows (the statement is slightly simplified for better understanding).

Theorem 12 ([101]). *Let $k \geqslant 1$, let $G = (\Sigma, N, R, S)$ be an LL(k) Boolean grammar, and let $T \colon N \times \Sigma^{\leqslant k} \to R \cup \{-\}$ be its parsing table. Then, for every $y, z \in \Sigma^*$ and $X_1, \dots, X_\ell \in \Sigma \cup N$ ($\ell \geqslant 0$), such that z follows $X_1 \dots X_\ell$, the code $X_1(); \dots; X_\ell()$, executed on the input yz,*

- *returns, consuming y, if $y \in L_G(X_1 \dots X_\ell)$;*
- *raises an exception, if $y \notin L_G(X_1 \dots X_\ell)$.*

The grammars from Examples 1 and 6 are both LL(1).

Example 18. *The Boolean grammar for the language $\{a^m b^n c^n \mid m, n \geqslant 0, m \neq n\}$, given in Example 6, is LL(1), and its smallest LL(1) table is given below.*

	ε	a	b	c
S	$S \to AB \,\&\, \neg DC$	$S \to AB \,\&\, \neg DC$	$S \to AB \,\&\, \neg DC$	$-$
A	$A \to \varepsilon$	$A \to aA$	$A \to \varepsilon$	$-$
B	$B \to \varepsilon$	$-$	$B \to bBc$	$B \to \varepsilon$
C	$C \to \varepsilon$	$-$	$-$	$C \to cC$
D	$D \to \varepsilon$	$D \to aDb$	$D \to \varepsilon$	$D \to \varepsilon$

On the other hand, the grammar in Example 2 is not LL(k) for any k, due to the ambiguity in the choice between the rules $E \to \varepsilon$ and $E \to a$ (or $E \to b$). It remains unknown whether the language $\{wcw \mid w \in \{a, b\}^*\}$ is generated by any LL(k) Boolean grammar.

Some limitations of LL(k) Boolean grammars have been established.

Theorem 13 (Okhotin [108]). *Every Boolean LL(k) language over a one-symbol alphabet is regular.*

Theorem 14 (Okhotin [108]). *For every Boolean LL(k) language $L \subseteq \Sigma^*$ there exist constants $d, d' \geqslant 0$ and $p \geqslant 1$, such that for all $w \in \Sigma^*$, $a \in \Sigma$, $n \geqslant d \cdot |w| + d'$ and $i \geqslant 0$,*

$$wa^n \in L \quad \text{if and only if} \quad wa^{n+ip} \in L$$

These theorems, in particular, imply that there are no Boolean LL(k) grammars for the linear conjunctive language $\{a^n b^{2^n} \mid n \geqslant 0\}$ and for the conjunctive language $\{a^{4^n} \mid n \geqslant 0\}$.

More detailed results on the limitations of LL(k) linear conjunctive and LL(k) linear Boolean grammars are known [108], and the former was proved to be strictly less powerful than the latter. Unfortunately, as far as general, non-linear LL(k) grammars are concerned, Theorems 13–14 are all tools available at present. In particular, it is not known whether LL conjunctive and LL Boolean grammars are equal in power.

For ordinary context-free grammars, LL parsing can be equally implemented using the recursive descent or using a special kind of a deterministic pushdown automaton without internal states, driven by the LL(k) table T_k. LL parsers of the latter kind can be generalized to LL(k) conjunctive grammars [84] by using a *tree-structured stack* instead of the standard linear stack. This model was studied by Aizikowitz and Kaminski [2, 3], who extended it with states and established the equivalence of its nondeterministic variant to the definition of conjunctive grammars by term rewriting. Aizikowitz and Kaminski [4] have further proposed a space-efficient implementation of such automata that represents their tree-structured stack as a graph-structured stack (similar to the one used in Generalized LR parsing). This might serve as a foundation for a new parsing technique.

6. Advanced approaches to parsing

This section describes several parsing methods, that have theoretically superior performance to the basic parsing algorithms discussed above. Even though some of them are quite unlikely to be useful in practice, they are important for understanding the theoretical complexity of formal grammars.

6.1. Parsing by matrix multiplication

Consider the basic cubic-time parsing algorithm from Section 5.1. The most time-consuming operation in the algorithm is computing the unions $U_{i,j} = \bigcup_{k=i+1}^{j-1} T_{i,k} \times T_{k,j}$, in which $U_{i,j}$ represents all concatenations BC that generate the substring $a_{i+1} \dots a_j$, and the intermediate position k is a cutting point in this substring, with B generating $a_{i+1} \dots a_k$ and with C generating $a_{k+1} \dots a_j$. If each union is computed individually, as it is done in the basic algorithm, then spending linear time for each $U_{i,j}$ is unavoidable. However, if such unions are computed for several sets $T_{i,j}$ at a time, then much of the work can be represented as Boolean matrix multiplication. This is illustrated in the following example:

Example 19 ([105]). *Let $w = a_1 a_2 a_3 a_4 a_5$ be an input string and consider the partially constructed parsing table depicted in Figure 6, with $T_{i,j}$ constructed for $0 \leqslant i < j \leqslant 3$ and for $2 \leqslant i < j \leqslant 5$, that is, for the substrings $a_1 a_2 a_3$*

and $a_3a_4a_5$, as well as for their substrings. Denote by $A_{i,j}$ the Boolean value indicating whether A is in $T_{i,j}$ or not. Then the following product of Boolean matrices

$$\begin{pmatrix} B_{0,2} & B_{0,3} \\ B_{1,2} & B_{1,3} \end{pmatrix} \times \begin{pmatrix} C_{2,4} & C_{2,5} \\ C_{3,4} & C_{3,5} \end{pmatrix} = \begin{pmatrix} X_{0,4} & X_{0,5} \\ X_{1,4} & X_{1,5} \end{pmatrix}$$

represents partial information on whether the pair (B, C) should be in the following four elements: $\begin{pmatrix} U_{0,4} & U_{0,5} \\ U_{1,4} & U_{1,5} \end{pmatrix}$. To be precise, $X_{1,4}$ computes the membership of (B, C) in $U_{1,4}$ exactly; $X_{0,4}$ does not take into account the factorization $a_1 \cdot a_2a_3a_4$, which actually requires knowing whether C is in $T_{1,4}$; the element $X_{1,5}$ is symmetrically incomplete; finally, $X_{0,5}$ misses the factorizations $a_1 \cdot a_2a_3a_4a_5$ and $a_1a_2a_3a_4 \cdot a_5$, which can be properly obtained only using $T_{0,4}$ and $T_{1,5}$. In total, this matrix product computes 8 conjunctions out of the 12 needed for these four elements of U.

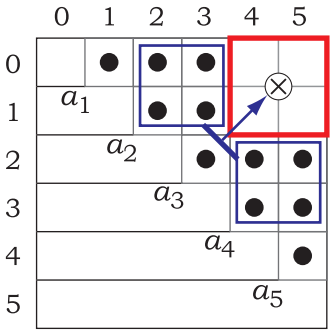

Figure 6: Using the product of two 2×2 Boolean matrices in a parsing algorithm to calculate multiple concatenations at once.

Already in this small example, using one matrix product requires changing the order of computation of the elements $\{T_{i,j}\}$: the elements $T_{0,3}$ and $T_{2,5}$ need to be calculated before $T_{1,4}$. Furthermore, the subsequent computation should be arranged to take care of the four remaining factorizations, which also must be considered in a specific order, evaluating $T_{i,j} = f(U_{i,j})$ for the appropriate entries at the appropriate time. This is achieved by a recursive partition of the matrix, discovered by Valiant [138], which arranges the computation of the products $T_{i,k} \times T_{k,j}$, so that as much work as possible is offloaded into products of the largest possible matrices. Once Valiant's [138] algorithm is simplified by removing its intermediate algebraic layer of abstraction (not discussed in this survey), and its data structures are changed accordingly, it becomes applicable to Boolean grammars.

Theorem 15 (Valiant [138]; Okhotin [105]). *For every Boolean grammar G in binary normal form, there exists an algorithm constructing the sets $T_{i,j} = \{A \in N \mid a_{i+1} \dots a_j \in L_G(A)\}$ for a given input string $w = a_1 \dots a_n$. The algorithm operates in time $O(|G| \cdot \mathrm{BMM}(n) \log n)$, where $\mathrm{BMM}(n)$ is the time needed to multiply two $n \times n$ Boolean matrices. Assuming $\mathrm{BMM}(n) = \Omega(n^{2+\varepsilon})$, the complexity is $\Theta(|G| \cdot \mathrm{BMM}(n))$.*

Besides the table $T_{i,j}$ with $0 \leqslant i < j \leqslant n$, the algorithm utilizes a second data structure, comprised of the elements $U_{i,j} \subseteq N \times N$ with $0 \leqslant i < j \leqslant n$, each corresponding to the value of R computed by the cubic-time algorithm for the appropriate substring. Their target values are $U_{i,j} = \{(B, C) \mid a_{i+1} \dots a_j \in L(B)L(C)\}$.

On an input of length $2^k - 1$, the algorithm multiplies submatrices of size 1×1, 2×2, 4×4, $\dots$, $2^{k-2} \times 2^{k-2}$. In the course of the computation, each entry $U_{i,j}$ is gradually filled with elements corresponding to different splitting points of the string $a_{i+1} \dots a_j$ into a concatenation of two strings, and for each splitting point, the exact matrix product that handles this concatenation is handled in one of the branches of recursion known in advance. Once all partitions are handled in the corresponding branches of recursion, and thus $U_{i,j}$ is constructed completely (which happens at a pre-determined point of the computation), the set $T_{i,j}$ is obtained out of $U_{i,j}$ by the function f defined above.

Corollary 15.1. *Testing whether a given Boolean grammar G generates a given string w of length n can be done in time $O(|G| \cdot n^\omega)$, for some $\omega < 2.376$, using the matrix multiplication method of Coppersmith and Winograd [17].*

The algorithm is perfectly suitable for practical implementation, as it can out-perform the basic cubic-time algorithm already on fairly short inputs. Products of Boolean matrices of a moderate size are best calculated by the algorithm by Arlazarov et al. [7], which uses $O(\frac{n^3}{\log n})$ bit operations. For very large matrices, it can be replaced by Strassen's [131] algorithm with the running time $O(n^{\log_2 7})$. An implementation recently given by Okhotin and Reitwießner [111] uses a Graphics Processing Unit for matrix multiplication.

The algorithm can be augmented to construct a parse tree of the input string as follows. The sets $U_{i,j} \subseteq N \times N$ shall be replaced with functions $U'_{i,j} \colon N \times N \to \{-, 1, \dots, n\}$, where $U'_{i,j}(B, C) = k$ indicates that $a_{i+1} \dots a_k \in L_G(B)$ and $a_{k+1} \dots a_j \in L_G(C)$, that is, the substring is split as $B \cdot C$ with the splitting position k. While the original sets $U_{i,j}$ are obtained by Boolean matrix multiplication, calculating the splitting positions for $U'_{i,j}$ is the problem of finding *witnesses for Boolean matrix multiplication*, solved by the algorithm by Alon and Naor [5], which uses $O(M(n) \log^5 n)$ operations in a finite ring, where $M(n)$ is the number of ring operations for matrix multiplication. Using this procedure, instead of the ordinary Boolean matrix multiplication in the algorithm in Theorem 15, yields an algorithm for constructing a parse tree in time $O(|G| \cdot n^\omega)$ with $\omega < 2.376$.

6.2. Parsing by convolution for unary inputs

For a Boolean grammar over a one-symbol alphabet, the basic cubic-time recognition algorithm presented in Section 5.1 can be straightforwardly modified to work in time $O(|G| \cdot n^2)$ on an input string a^n. Indeed, since

all substrings of any given length are identical, the table $T_{i,j} = \{A \mid a_{i+1} \dots a_j \in L_G(A)\}$ can be replaced with $T_\ell = \{A \mid a^\ell \in L_G(A)\}$, and the formula for calculating each entry takes the form

$$T_\ell = f(U_\ell), \quad \text{with } U_\ell = \bigcup_{i=1}^{\ell-1} T_i \times T_{\ell-i},$$

where $f\colon 2^{N\times N} \to 2^N$ represents the Boolean logic in the rules, as defined in Section 5.1.

In the case of a general alphabet, as described in Section 6.1, the whole bulk of bit operations necessary to calculate the unions $U_{i,j}$ for all $T_{i,j}$ can be split into multiple Boolean matrix multiplication subproblems, which allows using fast algorithms for this problem to achieve subcubic running time. For a one-symbol alphabet, the calculation of all U_ℓ can similarly be split into multiple instances of the *convolution of Boolean vectors*, which may be regarded as the one-dimensional analogue of Boolean matrix multiplication. The convolution of two Boolean vectors $x = (x_0, \dots, x_{n-1})$ and $y = (y_0, \dots, y_{n-1})$, denoted by $x \circ y$, is a vector $(z_0, \dots, z_{2n-2})$, with $z_i = \bigvee_{j=0}^{i} x_j \wedge y_{i-j}$, and this operations can be used in the calculation of U_ℓ as follows.

Example 20 (cf. Example 19). *Let $w = a^5$ and assume that the membership of strings of length up to 3 in the languages of each nonterminal has already been calculated and stored in $T_1, T_2, T_3 \subseteq N$. Denote by A_i the Boolean value indicating whether A is in T_i or not. Then the Boolean convolution*

$$(B_2, B_3) \circ (C_2, C_3) = (X_4, X_5, X_6)$$

represents partial information on whether the pair (B, C) should be in the elements U_4, U_5, U_6. The value $X_4 = B_2 \wedge C_2$ represents one out of three necessary conjunctions, with the conjunctions $B_1 \wedge C_3$ and $B_3 \wedge C_1$ missing; $X_5 = (B_2 \wedge C_3) \vee (B_3 \wedge C_2)$ includes two conjunctions out of four, where the remaining conjunctions, $B_1 \wedge C_4$ and $B_4 \wedge C_1$, require knowing the membership of B and C in T_4; finally, $X_6 = B_3 \wedge C_3$ is one out of five conjunctions needed for U_6.

There is a way of distributing the full set of n^2 concatenations used in the algorithm into convolutions of Boolean vectors of size $1, 2, 4, \dots, 2^{\lfloor \log_2 n \rfloor}$, so that each U_i is fully computed before the corresponding T_i is accessed. This partition, first proposed by Fischer and Stockmeyer [25] in their algorithm for online integer multiplication, can be regarded as the one-dimensional case of the similar partition used in parsing by matrix multiplication.

Theorem 16 (Okhotin, Reitwießner [112]). *For every Boolean grammar $G = (\{a\}, N, R, S)$ in binary normal form, there exists an algorithm, which, given a number $n \geqslant 1$, constructs the sets $T_\ell = \{A \in N \mid a^\ell \in L_G(A)\}$ for all $\ell \in \{1, \dots, n\}$. The algorithm works in time $O(|G| \cdot \mathrm{BC}(n) \log n)$, where $\mathrm{BC}(n)$ is the time needed to calculate convolution of two Boolean vectors of length n.*

If convolution is implemented through integer multiplication, and the latter is performed by the fastest known algorithm due to Fürer [28], this yields an algorithm with the running time $|G| \cdot n \log^3 n \cdot 2^{O(\log^* n)}$ [112]. If the grammar is unambiguous, the reduction to integer multiplication can be made more efficiently, improving the running time to $|G| \cdot n \log^2 n \cdot 2^{O(\log^* n)}$ [112].

6.3. On parallel parsing

The most efficient sequential parsing algorithm known for Boolean grammars of the general form is the one given in Section 6.1, which works on an input string of length n in time $O(\mathrm{BMM}(n) \log n)$, where $\mathrm{BMM}(n)$ is the complexity of Boolean matrix multiplication. No faster sequential algorithms are known even for ordinary context-free grammars.

A straightforward attempt to parallelize the basic cubic-time parsing algorithm leads to the following unsophisticated result.

Theorem 17. *For every Boolean grammar $G = (\Sigma, N, R, S)$ in binary normal form there exists a uniform family of Boolean circuits of depth $\Theta(n)$ and with $\Theta(n^3)$ gates, recognizing the membership of strings of each length n in the language $L(G)$.*

The circuit directly simulates the algorithm in Section 5.1. For each nonterminal $A \in N$ and for each substring $a_{i+1} \dots a_j$ of the input, with $0 \leqslant i < j \leqslant n$, the circuit contains a gate $x_{A,i,j}$, representing the membership of $a_{i+1} \dots a_j$ in $L_G(A)$. The membership of the same substring in $L_G(BC)$, for all $B, C \in N$, is calculated in another gate, $y_{BC,i,j}$. Then, as in Section 5.1,

$$y_{BC,i,j} = \bigvee_{k=i+1}^{i-1} x_{B,i,k} \wedge x_{C,k,j},$$
$$x_{A,i,j} = f_A(\dots, y_{BC,i,j}, \dots),$$

where the function $f_A\colon \mathbb{B}^{N\times N} \to \mathbb{B}$ represents the Boolean logic in the grammar. If the disjunction in the formula for $y_{BC,i,j}$ is calculated in a binary tree of height $\log_2(j-i)$, the depth of the entire circuit will be of the order $n \log n$. Overall linear depth of the circuit is achieved by calculating the conjunction for $k = \lfloor \frac{j-i}{2} \rfloor$ first, and then continuing with $k = i + t$ and $k = i - t$ for $t = \lfloor \frac{j-i}{2} \rfloor - 1, \dots, 1$.

For ordinary context-free grammars, there exists a much more efficient parallel parsing method, independently discovered by Brent and Goldschlager [12] and by Rytter [125]. The Brent–Goldschlager–Rytter parallel algorithm is implemented on a uniform family of $O(\log^2 n)$-depth Boolean circuits with $O(n^6)$ gates, and, in particular, establishes the containment of the ordinary context-free languages in the complexity class NC^2. Its main idea is best explained in terms of the definition of grammars by deduction. As always, the goal of the algorithm is to deduce elementary propositions of the form $[A, u]$, where

A is a nonterminal symbol and u is a substring of the input, which mean that $u \in L_G(A)$. The truth value of each proposition is calculated in one of the gates of the circuit. However, the tree of deduction of such an item may have linear height: this is already the case for the grammar $S \to aS \mid \varepsilon$. In order to obtain a deduction tree of logarithmic height, the algorithm further undertakes to deduce *conditional propositions* of the form $[D, v] \vdash [A, u]$, where u is a substring of the input and v is a substring of u, representing the condition that *if* $[D, v]$ *holds true, then so does* $[A, u]$. In other words, a conditional proposition represents a parse tree of u from A with a hole instead of a subtree of v from D. When this partial parse tree is of approximately the same size as the hole, this allows carrying out two deductions in parallel, thus halving the height of logical dependencies.

Unfortunately, this method does not generalize to Boolean grammars. Since every rule may have multiple conjuncts, each with its own set of dependencies, in this case, splitting a deduction of an item $[A, u]$ into multiple parallel deductions would require using conditional propositions of the form $[D_1, v_1], \ldots, [D_k, v_k] \vdash [A, u]$, where $k = \Theta(n)$. As there would be exponentially many such conditional propositions, no feasible algorithm can apparently be obtained along these lines. This applies already to linear conjunctive grammars.

In general, it is unlikely that there is any polylogarithmic-time parallel recognition algorithm for linear conjunctive grammars, for the reason that there exists a fixed grammar generating a P-complete language—which was first proved by Ibarra and Kim [47] using trellis automata—and if this single language is in NC^k for any k, this would imply that NC = P, which is generally believed to be untrue. All known examples of linear conjunctive grammars generating P-complete languages are fairly complicated [90, 106], and the possibility of defining a P-complete problem by a Boolean grammar is better explained in the following example.

Example 21 (Okhotin [106]). *The following Boolean grammar generates a* P*-complete language.*

$$\begin{array}{lcl} S & \to & \neg AbS \,\&\, \neg CS \\ A & \to & aA \mid \varepsilon \\ C & \to & aCAb \mid b \end{array}$$

The grammar can be made LL(1) by adding a new nonterminal E generating $\{a, b\}^*$, and adding a conjunct E to the rule for S.

The language generated by the grammar is the set of yes-instances for a variant of the circuit value problem. In this variant, every gate x_i computes the NOR function, $x_i = \neg x_{i-1} \wedge \neg x_{j_i}$, where the first argument is always the preceding gate, and the second argument can be any of the previous gates, $j_i < i$. Such an n-gate circuit is represented as a string $a^{n-1-j_n} b\, a^{n-2-j_{n-1}} b \ldots a^{2-j_3} b\, a^{1-j_2} b$. The grammar generates such descriptions of all circuits that evaluate to 1, and the definition, contained in the rule $S \to \neg AbS \,\&\, \neg CS$, inductively refers to the symbol S to determine the values of the previous gates. The conjunct AbS represents the circuits, in which the $(n-1)$-th gate has value 1, the conjunct CS specifies that the gate number j_n, pointed by the string a^{n-1-j_n}, has value 1. The negation of both conditions given in the rule for S implements the NOR operation.

6.4. Space complexity

All tabular parsing algorithms for Boolean grammars, presented in Sections 5.1–5.2 and 6.1, require $\Theta(n^2)$ bits of memory, where n is the length of the input string. Generalized LR uses space $C \cdot n^2$ in the worst case. The prototypes of these algorithms earlier developed for ordinary context-free grammars need the same amount of memory.

At the same time, there exist special polylogarithmic-space recognition procedures for ordinary context-free grammars. The first of them, using space $O(\log^2 n)$ on a Turing machine, was discovered by Lewis, Stearns and Hartmanis [76, Thm. 4]. The Brent–Goldschlager–Rytter parallel algorithm [12, 125] for ordinary context-free grammars, described in Section 6.3 above, actually has an similar underlying idea to the algorithm of Lewis et al. [76], and can be implemented on a $O(\log^2 n)$-space Turing machine by simulating its circuit. Compared to these results, the best currently known upper bound on the space complexity of Boolean grammars is very modest.

Theorem 18 (Okhotin [94]). *For every Boolean grammar G, there exists and can be effectively constructed a deterministic linear-space Turing machine recognizing the language $L(G)$.*

The argument given in the cited paper is by an explicit construction of such a machine, presented as a rewriting system that processes an input string. Alternatively, one can obtain such a machine by simulating the uniform circuit of depth $\Theta(n)$ presented in Theorem 17.

7. Theoretical topics

7.1. Grammars over a one-symbol alphabet

Conjunctive grammars over a unary alphabet form a special area of study. Though such grammars are completely irrelevant to the main purpose of formal grammars, that of representing syntax, they are theoretically important as a pure case of conjunctive grammars, which already shows some of their distinctive properties. Furthermore, they are crucial in the study of language equations, where their properties form the basis of the study of the more general language equations over a unary alphabet [52, 53, 55, 71, 72].

The idea of manipulating positional notation of numbers, first discovered by Jeż [49] in his grammar for the language $\{a^{4^n} \mid n \geqslant 0\}$ (Example 4), was subsequently

used to obtain the following general result. Consider a trellis automaton over an alphabet of k-ary digits, that recognizes some set of numbers, which are given to it in base-k notation. Then, there exists a conjunctive grammar that defines the same set of numbers, this time in unary notation.

Theorem 19 (Jeż, Okhotin [50]). *Let $\Sigma_k = \{0, 1, \ldots, k-1\}$ with $k \geqslant 2$ be an alphabet of k-ary digits, and let $L \subseteq \Sigma_k^*$ be a linear conjunctive language that contains no strings beginning with 0. Then, there exists a conjunctive grammar generating the language $\{a^n \mid \text{the } k\text{-ary representation of } n \text{ is in } L\}$.*

The proof is by simulating a trellis automaton $M = (\Sigma_k, Q, I, \delta, F)$ operating on positional notation of numbers, and doing so in terms of their unary representation.

Consider first a linear conjunctive grammar $G = (\Sigma_k, N, R, S)$ simulating this automaton in base-k notation, as it is given in Theorem 8. This grammar has a nonterminal A_q for each state $q \in Q$, where $L_G(A_q) \subseteq \Sigma_k^+$ is the set of all strings of digits, on which M computes the state q. Whenever there are strings of digits $iw \in L_G(A_{q'})$ and $wj \in L_G(A_{q''})$ with $i, j \in \Sigma_k$ and $w \in \Sigma_k^*$, and the automaton has a transition $\delta(q', q'') = q$, the string iwj should be in $L_G(A_q)$, and a linear conjunctive grammar over Σ_k implements this by a rule $A_q \to iA_{q'} \,\&\, A_{q''}j$.

In Theorem 19, the goal is to represent the *unary notation* of the very same numbers by a conjunctive grammar $G' = (\{a\}, N', R', S')$. For every string of digits $w \in \Sigma_k$, denote the number it represents by $(w)_k$. Ideally, one would construct a grammar with the nonterminals B_q for each $q \in Q$, which would generate all such strings a^n, that M calculates the state q on the base-k notation of n. However, such a construction does not work as it is, due to several reasons; in particular, obtaining a string $a^{(iwj)_k}$ out of the string $a^{(iw)_k}$ would require multiplying the number $(iw)_k$ by k, which is hardly possible using only concatenation of unary strings.

The actual grammar constructed in Theorem 19 indeed has a nonterminal B_q for each state of the trellis automaton, but uses the following encoding: for any string $w \in \Sigma_k^+$, on which the trellis automaton calculates the state q, the language B_q contains all strings of the form $a^{(1w'10^\ell)_k}$ with $\ell \geqslant 0$, where the string w' is obtained from w by subtracting 1 from the number it represents. For this encoding, it is possible to construct expressions $\lambda_i(L)$ and $\rho_i(L)$, which concatenate a single digit i to the left and to the right of the string of digits encoded in each element of $L \subseteq a^*$. These expressions are constructed using the operations of union, intersection and concatenation, and a number of separately expressed constant languages. Once these expressions are transcribed in the grammar, one can simulate each transition of a trellis automaton directly, using $\lambda_i(B_{q'}) \cap \rho_j(B_{q''})$ instead of $iA_{q'} \cap A_{q''}j$. The last step of the construction decodes the language $\{a^{(w)_k} \mid w \in L(M)\}$ out of the encodings given in B_q.

The unary simulation of trellis automata was subsequently reimplemented by Jeż and Okhotin [57] using an *unambiguous conjunctive grammar*, though under some restrictions on the use of digits in the original language.

A similar method was used to construct a set of numbers, whose binary representations form an EXPTIME-complete set, so that the set of unary representations of the same numbers is defined by a conjunctive grammar.

Theorem 20 (Jeż, Okhotin [51, Thms. 2,3]). *There exists an alphabet $\Sigma_k = \{0, 1, \ldots, k-1\}$ with $k \geqslant 2$ and an EXPTIME-complete language $L \subseteq \Sigma_k^*$, such that the language $\{a^n \mid \text{the } k\text{-ary notation of } n \text{ is in } L\}$ is generated by a conjunctive grammar.*

This, in particular, leads to the following result.

Corollary 20.1. *The compressed membership problem for conjunctive grammars, defined as "Given a conjunctive grammar $G_0 = (\Sigma, N, R, S)$ and a string $w \in \Sigma^*$ presented as a context-free grammar G with $L(G) = \{w\}$, determine whether $w \in L(G_0)$" is EXPTIME-complete. It remains EXPTIME-complete for a fixed G_0 (as in Theorem 20).*

The similar problem for ordinary context-free grammars is PSPACE-complete [120], while for a one-symbol alphabet it is NP-complete [44].

7.2. Descriptional complexity

It is known from Gruska [36] that the languages defined by ordinary context-free grammars form an infinite hierarchy with respect to the number of nonterminal symbols needed to define them; that is, for every $n \geqslant 2$ there exists a language representable using n nonterminals and not representable using $n-1$ nonterminals. Unfortunately, it remains unknown, whether conjunctive or Boolean grammars have a similar property (see Problem 9 in Section 9.1) However, for linear conjunctive grammars, the hierarchy of n-nonterminal languages collapses at level two, due to the following result.

Theorem 21 (Okhotin [93]). *For every trellis automaton M with n states, defined over an alphabet Σ, there exists and can be effectively constructed a linear conjunctive grammar with 2 nonterminal symbols generating the same language. The grammar has at most $2^{2^{O(n \log |\Sigma|)}}$ rules, each containing $O(n)$ conjuncts, and with $O(n)$ symbols in each conjunct.*

In fact, the entire machinery of a trellis automaton is encoded in a single language $L \subseteq \Sigma^+$, which is generated by a single nonterminal. The second nonterminal is needed to decode $L(M)$ from the language of the first nonterminal. This language L reflects the states computed by M on all strings of length 0 modulo $6n-1$. To that effect, a string wy, with $|w| \equiv 0 \pmod{6n-1}$ and $|y| = 2i-1$, belongs to L if and only if the trellis automaton computes its i-th state q_i on w, and a similar condition defines the membership

of a string xy with $|w| \equiv 0 \pmod{6n-1}$. Then, every rule of the constructed grammar uses this information to decode the states computed by the trellis automaton on $6n$ adjacent substrings of the same length, and then uses the pre-computed result of the automaton's computation on these states.

Conjunctive grammars over a unary alphabet with a unique nonterminal symbol are already non-trivial. The first example of their non-triviality, given by Okhotin and Rondogiannis [113], was actually an encoding of Example 4; this survey includes a slightly simpler encoding of the very same example:

Example 22. *The conjunctive grammar*

$$S \to a^8SS \,\&\, a^7SS \mid a^4SS \,\&\, a^{10}SS \mid a^3SS \,\&\, a^{12}SS \mid \\ \mid aSS \,\&\, a^6SS \mid a^{11} \mid a^{26} \mid a^{40}$$

generates the language $\{a^{4^n-5} \mid n \geqslant 2\} \cup \{a^{2\cdot 4^n-6} \mid n \geqslant 2\} \cup \{a^{3\cdot 4^n-8} \mid n \geqslant 2\} \cup \{a^{6\cdot 4^n-10} \mid n \geqslant 2\}$.

The example of Okhotin and Rondogiannis [113] was extended to the following general method of encoding any unary conjunctive language in a one-nonterminal conjunctive grammar.

Theorem 22 (Jeż, Okhotin [54]). *For every conjunctive grammar* $G = (\{a\}, \{A_1, \ldots, A_n\}, R, A_1)$ *in the binary normal form, there exist numbers* $0 < d_1 < \ldots < d_n < p$ *depending only on* n*, such that the language* $\{a^{pn-d_i} \mid 1 \leqslant i \leqslant n,\ a^n \in L_G(A_i)\}$ *is generated by a one-nonterminal conjunctive grammar.*

Some limitations of one-nonterminal conjunctive grammars are known.

Theorem 23 (Okhotin, Rondogiannis [113]). *Let* $L = \{a^{n_1}, a^{n_2}, \ldots, a^{n_i}, \ldots\}$ *with* $0 \leqslant n_1 < n_2 < \cdots < n_i < \cdots$ *be an infinite unary language, for which* $\liminf_{i\to\infty} \frac{n_i}{n_{i+1}} = 0$*. Then* L *is not generated by any one-nonterminal conjunctive grammar.*

In particular, no unary language with a super-exponential growth rate, such as $\{a^{2^{2^n}} \mid n \geqslant 0\}$ and $\{a^{n!} \mid n \geqslant 1\}$, can be represented by such grammars. The above theorem also applies to sets like $\{a^{n!+i} \mid n \geqslant 1,\ i \in \{0,1\}\}$.

The next theorem applies to such languages as $a^* \setminus \{a^{n^2} \mid n \geqslant 0\}$, $a^* \setminus \{a^{2^n} \mid n \geqslant 0\}$ and or $\{a^n \mid n \text{ is composite}\}$:

Theorem 24 (Okhotin, Rondogiannis [113]). *Let* $L \subseteq a^*$ *be a non-regular language that is* ***dense****, in the sense that* $\lim\limits_{n\to\infty} \frac{|L \cap \{\varepsilon, a, \ldots, a^{n-1}\}|}{n} = 1$*. Then there is no one-nonterminal conjunctive grammar generating* L*.*

The methods in Theorems 23–24 essentially use the fact that the grammar contains a single nonterminal symbol, and therefore can only concatenate this symbol to itself. Another special case of grammars demonstrating similar properties are unambiguous conjunctive grammars with 2 nonterminals: the reason is that such grammars may not concatenate any of these nonterminals to itself—any such concatenation is bound to be ambiguous—and hence are limited to concatenating these two nonterminals to each other. This was used by Jeż and Okhotin [56] to obtain results similar to Theorems 23–24 for this class of grammars.

No methods for establishing any limitations of conjunctive or Boolean grammars with three or more nonterminal symbols are known.

7.3. Undecidable properties

One of the main techniques for proving undecidability results in formal language theory, discovered by Hartmanis [38], is by expressing one or another form of the *language of computation histories* of a Turing machine. Accepting computations of a Turing machine T over an input alphabet Ω are represented as strings over an alphabet Γ, with the computation of T on $w \in \Omega^*$ represented as $C_T(w) \in \Gamma^*$. The language of valid accepting computations is

$$\mathrm{VALC}(T) = \{w \natural C_T(w) \mid w \in L(T)\},$$

defined over the alphabet $\Omega \cup \Gamma \cup \{\natural\}$, where $\natural \notin \Omega \cup \Gamma$ is a separator. Hartmanis [38] has shown that, for a certain simple encoding $C_T : \Omega^* \to \Gamma^*$, the language $\mathrm{VALC}(T)$ is an intersection of two context-free languages. The technical details of this construction can be further refined to represent $\mathrm{VALC}(T)$ as an intersection of two LL(1) linear context-free languages [98], for another suitable encoding C_T.

Therefore, for a proper choice of encoding C_T, $\mathrm{VALC}(T)$ is linear conjunctive, which directly implies the undecidability of the basic decision problems for this family, such as emptiness, finiteness, regularity and equivalence [83, 88]. Actually, there is a stronger result than just the undecidability of the emptiness problem:

Theorem 25. *For every fixed conjunctive language* $L_0 \subseteq \Sigma^*$ *over an alphabet* Σ *with* $|\Sigma| \geqslant 2$*, the problem of testing whether* $L(G) = L_0$ *for a given conjunctive grammar* G *over* Σ *is co-r.e.-complete (that is, complete for the complements of the recursively enumerable sets).*

The same result holds for unambiguous conjunctive, Boolean, unambiguous Boolean and linear conjunctive grammars, with a fixed language of the same type.

In contrast, for ordinary context-free grammars, equality to a fixed regular language is decidable if and only if that language is bounded [42], and equality to a fixed non-regular language has no known characterization of its decidable cases. For unambiguous context-free grammars, one can decide equality to a given arbitrary regular language [127].

The proof of Theorem 25 is by reducing the emptiness problem for a Turing machine T to the problem in question. There are two cases: if L_0 has *no subset of the form* $w_0\Sigma^*$, for any string $w_0 \in \Sigma^*$, then the reduction function constructs a conjunctive grammar for $L_0 \cup \mathrm{VALC}(T)\Sigma^*$, and if L_0 has a subset $w_0\Sigma^*$ for some $w_0 \in \Sigma^*$, then a grammar for the language $(L_0 \setminus w_0\Sigma^*) \cup w_0(\Sigma^* \setminus \mathrm{VALC}(T))$ is similarly constructed. In each case, the constructed grammar generates L_0 if $\mathrm{VALC}(T) = \varnothing$, and a different language if $\mathrm{VALC}(T) \neq \varnothing$. Since $\mathrm{VALC}(T) = \varnothing$ if and only if $L(T) = \varnothing$, this is a correct reduction.

Surprisingly, the above method of proving undecidability results extends to grammars over a one-symbol alphabet. Despite the apparent lack of structure in strings in a^*, a variant of the language of computation histories of a Turing machine can still be represented. Let $\mathrm{VALC}(T)$ be defined over a k-symbol alphabet, and assume that the symbols of this alphabet are digits in base-k notation. Then every string $w \natural C_T(w) \in \mathrm{VALC}(T)$ represents a certain natural number $(w \natural C_T(w))_k \in \mathbb{N}$. Since $\mathrm{VALC}(T)$ is linear conjunctive, Theorem 19 asserts that the set $(\mathrm{VALC}(T))_k$ of unary representations of these numbers is defined by a conjunctive grammar.

This, in particular, leads to an adaptation of Theorem 25 to a unary alphabet:

Theorem 26 (Jeż, Okhotin [50, 57]). *For every fixed conjunctive (unambiguous conjunctive) language $L_0 \subseteq a^*$, the problem of testing whether a given conjunctive grammar (unambiguous conjunctive grammar, respectively) over $\{a\}$ generates the language L_0 is co-r.e.-complete.*

Turning to the basic decision problems for conjunctive grammars over a unary alphabet, the representation of the unary version of $\mathrm{VALC}(T)$ is alone sufficient to prove their undecidability. With the additional help of Theorem 22, the language $(\mathrm{VALC}(T))_k \subseteq a^*$ can be further encoded using a single nonterminal symbol, which leads to the following stronger undecidability results:

Theorem 27 (Jeż, Okhotin [54]). *Testing whether a given one-nonterminal conjunctive grammar generates a finite or a co-finite language is r.e.-complete. Testing equivalence of two given one-nonterminal grammars is co-r.e.-complete.*

Summarizing the known results, the only predicate on conjunctive languages known to be decidable is the membership of a string, and, along with it, all finite Boolean combinations of such predicates (such as "$w_1 \in L$ or $w_2 \notin L$"). At the same time, all known properties of languages depending on infinitely many strings turned out to be undecidable. This suggests a hasty generalization: a conjecture that a non-trivial predicate on conjunctive languages is decidable if and only if it depends on the membership of finitely many strings. However, there is no evidence in support of this conjecture, and some property of an entirely different form may turn out to be decidable.

8. Comparison of formal grammars

8.1. Hierarchy of language families

In order to compare the expressive power of meaningful models of syntax, one should begin with compiling a list of such models. The main point of reference are, of course, the *ordinary context-free grammars* (*CF*). Many important familes of languages are defined by restricting context-free grammars in one or another way. Prohibiting syntactic ambiguity leads to the *unambiguous context-free grammars* (*UnambCF*), and to their special cases: the *LR(k) context-free grammars*, which define the deterministic context-free languages (*DetCF*), and to the *LL(k) context-free grammars* (*LLCF*). All these four classes are known to form a chain of proper inclusions (*LLCF* $\subset$ *DetCF* $\subset$ *UnambCF* $\subset$ *CF*), and their fixed membership problem belongs to the NC^2 complexity class [124, 12, 125]. The fixed membership problem for *DetCF* and *LLCF* also belongs to the class $\mathrm{SC}^2 = \mathrm{DTIMESPACE}(n^{O(1)}, \log^2 n)$ [16].

Another common restriction is to allow only concatenation of the form uAv, where A is a nonterminal and u, v are terminal strings. The corresponding *linear context-free grammars* (*LinCF*) and their *unambiguous* (*UnambLinCF*), *deterministic* (*DetLinCF*) and *LL* (*LLLinCF*) subfamilies again form a chain of proper inclusions (*LLLinCF* $\subset$ *DetLinCF* $\subset$ *UnambLinCF* $\subset$ *LinCF*), and each of them is a proper subset of its counterpart with unrestricted concatenation. These families belong to lower computational complexity classes: the family *LinCF* is a subset of the nondeterministic logarithmic space (NL) containing an NL-complete language, discovered by Sudborough [132]; *UnambLinCF* is contained in the unambiguous logspace (UL). As shown by Holzer and Lange [41], the deterministic subfamily *DetLinCF* is a subset of the logarithmic space (L), which contains an L-complete language; Ibarra et al. [45] demonstrated that *LLLinCF* is contained in NC^1.

Yet another useful restriction of the ordinary context-free languages is defined by the *input-driven pushdown automata* (IDPDA), in which the input alphabet is split into three classes, so that the type of the current symbol determines whether the automaton must push onto the stack, pop from the stack, or ignore the stack. Input-driven automata were first studied by Mehlhorn [80], followed by von Braunmühl and Verbeek [11], and later rediscovered and studied by Alur and Madhusudan [6] under the name of "visibly pushdown automata". The corresponding language family *IDCF* is a subset of *DetCF* by definition, and it is also known to be incomparable with *LLCF* [107] and with all families between *LLLinCF* and *LinCF*. Dymond [22] showed that *IDCF* is contained in NC^1.

Returning to the subject of this survey, the classification of formal grammars is now extended by another dimension, which is the set of allowed Boolean operations. The ordinary context-free grammars, as well as all their special cases mentioned so far, are restricted to disjunction only. Allowing the conjunction alongside the disjunc-

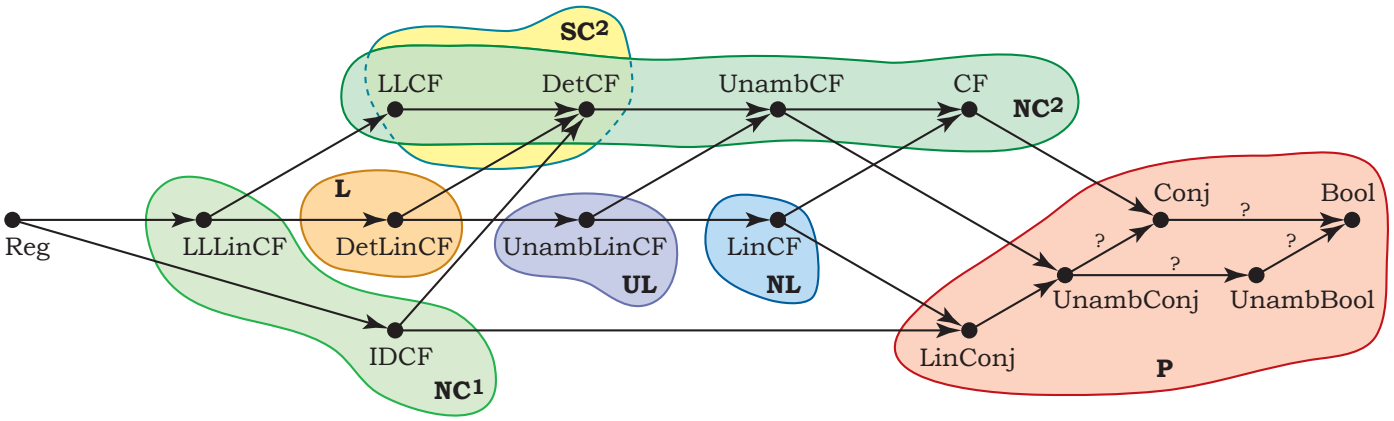

Figure 7: The hierarchy of formal grammars, and their relation to computational complexity classes.

tion leads to the conjunctive grammars (*Conj*), and further allowing the negation results in the Boolean grammars (*Bool*). An exhaustive investigation of all possible sets of Boolean operations in grammars [100] revealed only one more non-trivial case: grammars with negation only, which, although non-trivial [74], are quite unsuitable for representing syntax; their properties are of a purely theoretical interest [115, 116] and fall outside the scope of this survey.

Grammars with different sets of Boolean operations are subject to the same restrictions as ordinary grammars with disjunction only. *Unambiguous conjunctive grammars* (*UnambConj*) and *unambiguous Boolean grammars* (*UnambBool*) are defined by restricting the use of concatenation. Restricting concatenation to linear leads to *linear conjunctive grammars* and *linear Boolean grammars*, which are known to generate the same family of languages, *LinConj*. Every linear conjunctive grammar can be made unambiguous (Corollary 8.1), so there are no unambiguous subclasses in this case. For the reasons of succinctness, the hierarchy presented in this survey does not include the LL subfamilies of conjunctive and Boolean grammars, as well as of their linear cases. An analysis of these classes and their relationship to each other can be found in the literature [108].

An inclusion diagram of all these families of languages is presented in Figure 7. For each family, the figure presents the smallest computational complexity class, in which it is known to be contained.

Most of the inclusions in the figure are immediate, as they assert that a less general model defines a subset of the family defined by a more general model: for instance, *LinCF* is contained in both *CF* and *LinConj*, etc. The inclusion $\mathit{IDCF} \subset \mathit{LinConj}$ is the only one between apparently unrelated formalisms [107].

All stated inclusions are known to be strict, except the four inclusions between conjunctive and Boolean grammars, unambiguous and general. These inclusions are marked in the figure by arrows with question marks.

There are two more models, which ought to be mentioned here, even though they are not included in the comparison. One of the models is the *first-order logic over positions in the string, augmented with a least fixpoint operator*, which, as established by Immerman [48] and by Vardi [139], defines exactly the polynomial-time languages. Rounds [123] was the first to understand this logic as a general model for defining syntax, and various kinds of formal grammars as its fragments. In particular, conjunctive grammars are representable in this logic by directly employing the conjunction, while Boolean grammars can be represented by using a universal quantifier [95]. The expressive power of first-order logic with fixpoints goes much beyond conjunctive and Boolean grammars, and this logic has a lot of potential for further studies in formal grammars.

The other model are the *tree-adjoining grammars*, defined by Joshi, Levy and Takahashi [58], which can be described through the first-order logic with fixpoints, and which are likely incomparable in power with conjunctive and Boolean grammars, though there are no known methods for proving that. In the absence of any research comparing the models, there is nothing to report on in this survey; this would certainly be an interesting topic for future research.

8.2. Closure properties

A family of languages $\mathcal{L}$ is said to be *closed* under an operation $f : 2^{\Sigma^*} \times \ldots \times 2^{\Sigma^*} \to 2^{\Sigma^*}$, if, for all $L_1, \ldots, L_n \in \mathcal{L}$, the language $f(L_1, \ldots, L_n)$ also belongs to $\mathcal{L}$. The closure of a language family under the basic operations on languages, in particular, reflects the possibility for expressing the syntactical conditions within this family. This section summarizes the known closure properties of the five language families surveyed in this paper: those generated by linear conjunctive grammars (*LinConj*), conjunctive grammars (*Conj*), Boolean grammars (*Bool*), unambiguous conjunctive grammars (*UnambConj*) and unambiguous Boolean grammars (*UnambBool*).

To begin with the Boolean operations, all five families *LinConj*, *Conj*, *Bool*, *UnambConj* and *UnambBool* are

	$\cup$	$\cap$	$\sim$	$\cdot$	$*$	$\sqcup$	R	SHIFT	h	$h_{\varepsilon\text{-free}}$	h_{code}	h^{-1}
Reg	+	+	+	+	+	+	+	+ [77]	+	+	+	+
LLLinCF	− [122]	− [108]	− [122]	− [122]	− [140]	−	− [122]	− [108]	− [122]	− [122]	− [73]	− [73]
DetLinCF	−	−	+	−	−	−	− [29]	−	− [32]	− [32]	− [73]	+ [29]
UnambLinCF	−	−	?	−	−	−	+	−	− [32]	− [32]	+	+
LinCF	+	−	−	−	−	−	+	−	+	+	+	+ [37]
IDCF	+*	+*	+ [11]	+* [114]	+ [114]	−	+ [11, 114]	−	−	−	−	−
LLCF	− [122]	− [122]	− [122]	− [122]	− [140]	−	− [122]	− [108]	− [122]	− [122]	− [73]	− [73]
DetCF	− [29]	− [29]	+ [29]	− [29]	− [29]	−	− [29]	−	− [29]	−	− [73]	+ [29]
UnambCF	−	−	− [40]	− [32]	−	−	+	−	− [32]	− [32]	+ [70]	+ [32]
CF	+	− [126]	− [126]	+	+	−	+	+ [78, 118]	+	+	+	+$^\circ$ [31, 35]
LinConj	+	+	+ [88, 92]	− [134]	− [92]	−	+	− [135]	− [20, 99]	− [20]	+ [99]	+ [20, 47]
UnambConj	?	+	?	?	?	?†	+	?	−	?†	+ [70]	+ [70]
UnambBool	+	+	+	?	?	?†	+	?	−	?†	+ [70]	+ [70]
Conj	+	+	?	+	+	?†	+	?	−	?†	+ [70]	+$^\circ$ [70, 109]
Bool	+	+	+	+	+	?†	+	?	−	?†	+ [70]	+$^\circ$ [70, 109]

Table 1: Closure properties of different families of formal grammars: (*) closed, assuming the same partition of alphabets for *IDCF*; (†) closure would imply P = NP; ($^\circ$) the entire family represented as images of a single language.

closed under **intersection**, which is expressed by a direct application of conjunction. In contrast, neither the ordinary context-free languages nor any of their basic subclasses are closed under this operation. The *union* operation is directly expressed in all these families by using multiple rules for a single nonterminal symbol, though it is not known whether it preserves unambiguity of conjunctive languages. The other unambiguous class *UnambBool* is closed under union, which is proved by manipulating negation to represent any union as a *disjoint* union [103]. **Complementation** is directly expressible in Boolean grammars and preserves their unambiguity, but it is an open problem whether *Conj* and *UnambConj* are closed under it (see Problem 6 in Section 9.1). The closure of *LinConj* under complementation is easily seen on trellis automata, though a direct construction for grammars is also known [88].

Turning to concatenation-based operations, the families *Conj* and *Bool* are closed under both **concatenation** and **Kleene star**, since they are directly expressible in grammars, while the closure of the unambiguous subclasses *UnambConj* or *UnambBool* is uncertain. Linear conjunctive languages are not closed under concatenation, which was proved by Terrier [134], solving a long-standing problem on cellular automata. The latter result easily implies the non-closure of this family under Kleene star [92]. None of the families are closed under the quotient, $K^{-1} \cdot L = \{w \mid aw \in L\}$, because every recursively enumerable set can be represented as a quotient of VALC(T) with a regular language. The **right-** and **left-quotient with a symbol**, defined as $a^{-1} \cdot L = \{w \mid aw \in L\}$ and $L \cdot a^{-1} = \{w \mid aw \in L\}$, are both known to preserve *LinConj* and *Conj* [110]. Similar arguments should apply to *Bool*, and will likely extend to *UnambConj* and *UnambBool*. The operation of interleaving any two strings belonging to two given languages, known as **shuffle**, denoted by $K \sqcup L$ or $K||L$, and defined as $\{x_1 y_1 \dots x_m y_m \mid x_1 \dots x_m \in K,\ y_1 \dots y_m \in L\}$, does not preserve ordinary context-free languages or any of their basic subfamilies. The non-closure of the linear conjunctive languages under shuffle can be proved by considering the language $\left(\{a^n b a^{2^n} \mid n \geqslant 0\} \sqcup b^*\right) \cap (ab)^*$, which is left as an exercise to the reader. It remains unknown whether conjunctive and Boolean grammars with unrestricted concatenation, or their unambiguous subclasses, are closed under shuffle. If any of these families is closed under shuffle, this would imply P = NP, which can be proved by shuffling a known P-complete linear conjunctive language [90] with an appropriate regular language.

Consider the following two operations that rearrange symbols in all strings of their language argument: **reversal** $L^R = \{a_n \dots a_1 \mid a_1 \dots a_n \in L\}$ and **cyclic shift** $\text{SHIFT}(L) = \{a_i \dots a_n a_1 \dots a_{i-1} | a_1 \dots a_{i-1} a_i \dots a_n \in L,\ 1 \leqslant i \leqslant n\}$. For each of the families in question, the closure under reversal is obtained simply by reversing the right-hand sides of all rules in a grammar. Several proofs of the closure of the context-free languages under cyclic shift are known [78, 118]; in particular, Hopcroft and Ullman [43, Exercise 3.4(c)] present a transformation of a context-free grammar to a grammar for its cyclic shift. Whether anything similar holds for Boolean grammars or any of their variants, is unknown. As noted by Terrier [135, Fact 6], linear conjunctive languages are not closed under cyclic shift.

The last group of operations to be considered are homomorphisms and some related operations. None of these families are closed under general (erasing) **homomorphisms**, because every recursively enumerable language is a homomorphic image of the language VALC(T) of computation histories of the Turing machine defining the language in question. For **non-erasing homomorphisms**, linear conjunctive languages are known to be not closed,

while the closure of the other four families is an open problem; however, it is worth mentioning that the closure would imply P = NP [90], which makes it unlikely. All five families are closed under **codes** (that is, injective homomorphisms), and, more generally, under injective mappings computed by generalized sequential machines (GSM) [70]. For linear conjunctive languages, it is furthermore known that they are closed under a homomorphism h if and only if either h is a code, or h erases all symbols [99]. All five families are also closed under **inverse homomorphisms** and inverse GSM mappings [70].

Concerning inverse homomorphisms, for ordinary context-free grammars, there is a famous theorem by Greibach [35], which states that there exists a single ordinary context-free language L_0 ("the hardest context-free language"), so that every ordinary context-free language L over any alphabet can be represented as an inverse homomorphic image of L_0. The conjunctive grammars are known to have their own "hardest language", and so do the Boolean grammars [109]. Whether a similar theorem exists for linear conjunctive languages, is unknown.

All the mentioned closure properties are summarized in Table 1. These results are compared to the (mostly) well-known properties of ordinary context-free grammars and their subfamilies. Discussing the closure properties of the latter is beyond the scope of this survey. The references to the literature given in the table attempt to point to the most relevant work: in most cases, this is the paper, in which the corresponding result was first established; and occasionally, the table lists one or more papers providing a further insight into applying the given operation to languages from the given class.

8.3. Decision problems

Let $\mathcal{G}$ be a family of grammars or automata. The **fixed membership problem** for $\mathcal{G}$ is stated as follows: *"For a fixed grammar $G \in \mathcal{G}$ over an alphabet Σ, given a string $w \in \Sigma^*$, is w in $L(G)$?"*. As explained in Section 6.3, this problem is P-complete for all five families of grammars studied in this paper (*LinConj*, *UnambConj*, *UnambBool*, *Conj* and *Bool*), because it is decidable in polynomial time, and is P-hard for a fixed unambiguous linear conjunctive grammar. The best running time among the known algorithms for solving this problem is $O(n^2)$ for linear and unambiguous grammars, and $O(n^\omega)$ with $\omega < 2.376$ for general conjunctive and Boolean grammars (see Section 6.1).

The more general **uniform membership problem** includes the grammar as a part of the input: *"Given a grammar $G \in \mathcal{G}$ over an alphabet Σ, and a string $w \in \Sigma^*$, is w in $L(G)$?"*. The problem is again P-complete: it is P-hard, because already the fixed membership problem is P-hard, and polynomial-time solutions for this problem are known: for Boolean grammars, a solution is given, for instance, by the Generalized LR parsing algorithm described in Section 5.3.

Other typical decision problems involve testing one or another property of the language generated by a given grammar. For instance, the emptiness problem is stated as follows: *"Given a grammar $G \in \mathcal{G}$, determine whether $L(G)$ is empty"*. As shown in Section 7.3, it is undecidable already for linear conjunctive grammars; to be more precise, it is complete for the class of the complements of recursively enumerable sets (co-r.e.). The more general **equivalence problem**, stated as *"Given two grammars $G, G' \in \mathcal{G}$, determine whether $L(G) = L(G')$"*, is co-r.e.-complete as well, and so is the **inclusion problem**, which asks for determining whether $L(G) \subseteq L(G')$. Both problems are undecidable already for ordinary context-free grammars. For some subfamilies of ordinary context-free grammars, equivalence is known to be decidable; its decidability status for unambiguous context-free grammars and for their linear subclass remains undetermined.

Decidability and complexity of main decision problems for Boolean grammars and their subfamilies is compared in Table 2.

9. Research directions

9.1. Nine theoretical problems

The previous survey of Boolean grammars [102] introduced nine open problems, each concerned with some theoretical property of Boolean grammars. Since then, two problems have been solved, and seven others remain open[2]. An award is offered for the first correct solution of each of the remaining problems[3].

The first and the most important problem concerns the *limitations of Boolean grammars*. The limitations of the expressive power of the ordinary context-free grammars are known quite well. Besides the complexity upper bounds, there exist direct techniques of proving non-representability of particular languages, such as the pumping lemma and its variants, as well as Parikh's theorem, which show that some computationally very easy languages cannot be generated by any context-free grammar. In contrast, no methods of proving non-representability of languages by Boolean grammars are currently known, and this is the foremost gap in the knowledge on these grammars. No such methods are known already for conjunctive grammars.

Of course, the known upper bounds on the complexity of parsing for Boolean grammars (see Sections 6.1 and 6.4) already imply that all computationally harder languages are beyond their scope. The question is, whether there

[2]The current status of these problems is displayed at the author's web page, `http://users.utu.fi/aleokh/boolean/nine_open_problems.html`.

[3]In most cases, a solution must be published in a recognized journal or presented at a recognized conference to qualify for the award. The award is $360 Canadian per problem, which is equally distributed between the authors of the solution. If two papers solving the same question appear simultaneously, the award is split between them. Each author receives a handwritten certificate and an award cheque (which may be replaced by another financial instrument). Every lady among the authors additionally receives a flower.

	Membership		Properties of a language			
	fixed	uniform	emptiness	equality to $L_0 \in$ *Reg*	equality	inclusion
DFA	regular	L	NL	NL	NL	NL
NFA	regular	NL	NL	PSPACE	PSPACE	PSPACE
LLLinCF	in NC^1 [45, 41]	L	NL	in P	decidable [137]	co-r.e. [98]
DetLinCF	L [41]	L [41]	NL	in P	decidable [137]	co-r.e. [98]
UnambLinCF	in UL [14]	in UL	NL	decidable [127, 130]	?	co-r.e. [98]
LinCF	NL [132]	NL	NL	co-r.e.	co-r.e.	co-r.e.
DIDPDA	in NC^1 [22]	in P	P [6, 67]	P	P	P
NIDPDA	in NC^1 [22]	in P	P [6, 67]	EXP [6]	EXP [6]	EXP [6]
LLCF	in $\mathrm{NC}^2 \cap \mathrm{SC}^2$	P	P	P [29]	decidable [122, 117]	co-r.e. [27]
DetCF	in $\mathrm{NC}^2 \cap \mathrm{SC}^2$ [16]	P	P	P [29]	decidable [128]	co-r.e. [29]
UnambCF	in NC^2	P	P	decidable [127]	?	co-r.e. [29]
CF	in NC^2 [124, 12, 125]	P	P	co-r.e. [10, 42]	co-r.e.	co-r.e. [10]
LinConj	P [47, 106]	P	co-r.e.	co-r.e.	co-r.e.	co-r.e.
UnambConj	P	P	co-r.e.	co-r.e.	co-r.e.	co-r.e.
UnambBool	P	P	co-r.e.	co-r.e.	co-r.e.	co-r.e.
Conj	P [106]	P [89]	co-r.e. [83]	co-r.e. [83]	co-r.e. [83]	co-r.e. [83]
Bool	P [94]	P [97]	co-r.e.	co-r.e.	co-r.e.	co-r.e.

Table 2: Decidability and computational complexity of decision problems for formal grammars (an entry "$\mathcal{C}$" indicates completeness for a complexity class $\mathcal{C}$; "in $\mathcal{C}$" means a problem belonging to $\mathcal{C}$, but not known to be complete for $\mathcal{C}$; "?" marks problems with unknown decidability status).

exist any *computationally easy* languages that cannot be defined by Boolean grammars, and how can one generally prove assertions of this kind? The following statement was designed to rule out trivialized answers based on complexity:

Problem 1. *Is there any language recognized by an algorithm working in time $O(n^2)$ and using space $O(n)$, which cannot be defined by a Boolean grammar?*

The second problem proposed in the 2006 survey [102] was concerned with the expressive power of conjunctive grammars over a one-symbol alphabet, asking whether they are trivial or not.

Problem 2 (Solved negatively by Jeż [49] in 2007). *Do conjunctive grammars over a one-symbol alphabet generate only regular languages?*

It was suggested in the previous survey, that *"If they can generate any non-regular language, this would be a surprise"* [102]. The surprise took form of a conjunctive grammar for the language $\{a^{4^n} \mid n \geqslant 0\}$ [49], given in Example 4. The ideas of that example led to a whole direction of theoretical research on conjunctive grammars over a one-symbol alphabet [50, 51, 54, 57, 56, 113], presented in Sections 7.1–7.3. It also had a significant impact on understanding the power of language equations of a more general form [52, 53, 55, 69, 71, 72]

The next problem asked about the time complexity of Boolean grammars, whether one can test membership faster than in cubic time.

Problem 3 (Solved positively by Okhotin [105] in 2009). *Are the languages generated by Boolean grammars contained in deterministic time $O(n^{3-\varepsilon})$, for any $\varepsilon > 0$?*

Though, at the first glance, Valiant's [138] algorithm for ordinary context-free grammars looked as if it essentially uses the encoding of a grammar in a semiring, which could hardly be achieved for a conjunctive grammar, a straightforward refactoring, presented in Theorem 15, led to a simpler variant of Valiant's algorithm, which is naturally applicable to the general case of Boolean grammars. Accordingly, the family of languages generated by Boolean grammars is contained in deterministic time $O(n^\omega)$ with $\omega < 2.376$.

A similar problem about the space complexity of Boolean grammars still remains open.

Problem 4. *Are the languages generated by Boolean grammars contained in deterministic space $O(n^{1-\varepsilon})$, for any $\varepsilon > 0$?*

If this were proved for any $\varepsilon > 0$, this would, in particular, separate Boolean grammars from $\mathrm{DSPACE}(n)$, and hence from Chomsky's "type 1" rewriting systems.

The next problem concerns a possible analogue of the Greibach normal form [34] for Boolean grammars. A Boolean grammar is in Greibach normal form, if all its rules are of the form

$$A \to a\alpha_1 \,\&\, \ldots \,\&\, a\alpha_m \,\&\, \neg a\beta_1 \,\&\, \ldots \,\&\, \neg a\beta_n, \qquad (8)$$

where $a \in \Sigma$, $m+n \geqslant 1$ and $\alpha_i, \beta_i \in (\Sigma \cup N)^*$. However, it is not known whether the family of languages generated by Boolean grammars in Greibach normal form is the same

as the entire family generated by Boolean grammars. This is proposed as a research problem:

Problem 5. *Is it true that for every Boolean grammar there exists a Boolean grammar in Greibach normal form generating the same language?*

Though the answer is uncertain, there are some problems with devising a transformation of a grammar to this normal form. Consider the language $\{a^n b^{2^n} \mid n \geqslant 0\}$, which has a linear conjunctive grammar constructed along the lines of Example 13, but this grammar is definitely not in Greibach normal form. The only known grammar in Greibach normal form for this language involves representing an auxiliary language $\{b^{2^m} \mid m \geqslant 0\}$, generally as in Example 4 (the construction is left as an exercise for the reader). Therefore, if a transformation to the Greibach normal form exists, then this transformation might have to discover Example 4 on the basis of Example 13, which appears unlikely. If such a transformation exists, then finding out how to transform Example 4 is a good starting point.

The family of languages generated by Boolean grammars is closed under all Boolean operations and concatenation, simply by virtue of having the corresponding operators as a part of the formalism. However, conjunctive grammars do not have an explicit negation operator, and the question of whether negation can still be somehow expressed—that is, whether for every conjunctive grammar G there exists a grammar for the complement of $L(G)$—is open.

Problem 6. *Is the family of conjunctive languages closed under complementation?*

If the answer is negative, a possible witness language is the one from Example 7: the language $\overline{\{ww \mid w \in \{a,b\}^*\}}$ is known to be context-free, while its complement might be non-representable by conjunctive grammars. The same problem could be separately considered for conjunctive languages over a unary alphabet. In fact, for every unary language found to be conjunctive in the literature, its complement could be proved conjunctive by the same methods [50, 51], yet there are no methods of transforming a given conjunctive grammar to a grammar for the complement of the generated language. One can try approaching this problem by considering whether the complement of every unambiguous context-free language is necessarily conjunctive.

The definition of unambiguous Boolean grammars suggests the question of whether there exist any inherently ambiguous languages with respect to Boolean grammars.

Problem 7. *Does there exist a Boolean grammar, which has no equivalent unambiguous Boolean grammar?*

The language $\{ww \mid w \in \{a,b\}^*\}$ is a possible candidate for being inherently ambiguous, because the only known way of representing this language is by putting a negation on top of an ambiguous ordinary context-free grammar, as in Example 7. Some limitations of unambiguous context-free grammars were established by Flajolet [26] using the methods of complex analysis. At the first glance, it appears that nothing of this kind could be applicable already to conjunctive grammars, but perhaps there is more to investigate.

Much more is known about the family of Boolean LL(k) languages, for which some limitations have already been established [108], see Theorems 13–14. What remains completely unknown, is whether there exists an infinite hierarchy of languages with respect to the length of lookahead k.

Problem 8. *Does there exist a number $k_0 > 0$, such that, for all $k \geqslant k_0$, Boolean LL(k) grammars generate the same family of languages as Boolean LL(k_0) grammars?*

To compare, for LL(k) context-free grammars, an infinite hierarchy with respect to k was established by Kurki-Suonio [64].

The last of the nine problems in the 2006 survey [102] concerns with the families of languages generated by n-nonterminal Boolean grammars. The question is, whether these families form an infinite hierarchy with respect to n, or does this hierarchy collapse at some point?

Problem 9. *Does there exist a number $k > 0$, such that every language generated by any Boolean grammar can be generated by a k-nonterminal Boolean grammar?*

For conjunctive grammars, the answer to this question is also unknown. To compare, for ordinary context-free grammars, an infinite hierarchy was established by Gruska [36], while for linear conjunctive grammars, two nonterminals are sufficient to represent every linear conjunctive language (Theorem 21). Perhaps, some progress could be made by investigating any limitations of 2-nonterminal grammars.

Besides the announced theoretical problems, one can formulate and solve many related questions. For instance, analogues of Problems 1 (limitations of Boolean grammars), 4 (space complexity), 5 (Greibach normal form), 8 (LL(k) hierarchy) and 9 (nonterminal hierarchy) can be stated for conjunctive grammars or for unambiguous conjunctive grammars. A variant of Problem 3 (time complexity) for unambiguous Boolean grammars would ask about parsing in time $O(n^{2-\varepsilon})$: the answer is unknown already for unambiguous linear context-free grammars. Problem 6 (complementation) is no less interesting for unambiguous conjunctive grammars. All the same questions could be studied for grammars restricted to a one-symbol alphabet. In relation to Problem 4 (space complexity), one could consider at least separating unambiguous conjunctive grammars from NSPACE(n).

9.2. Further topics

Besides solving specific theoretical problems about formal grammars, one can consider many possible directions for general investigation.

For instance, one can try to obtain **an attractive subclass** of Boolean grammars, which would preserve their essential expressive power, and at the same time allow more efficient parsing than in the general case. What of the expressive power of Boolean grammars should be regarded as essential? Of what is known, the most useful is, perhaps, the ability to compare identifiers, demonstrated in Example 2, and the ability to check declaration before use, as in Example 3. If one could find a subclass of Boolean grammars, that is still able to generate these two languages, and which would have a parsing method with a running time $o(n^2)$, perhaps $n \log^{O(1)} n$, then this would make a significant step towards the practical use of Boolean grammars. This is mostly a matter of finding an entirely new efficient parsing algorithm; once such an algorithm is discovered, it may redefine the understanding of what is essential in Boolean grammars.

Thinking in the opposite direction, perhaps it could be possible to invent **a meaningful superclass** of conjunctive or Boolean grammars, which would add some useful expressive means, and yet maintain the crucial properties of context-free grammars. Defining the properties of strings in some sense inductively on the length of the strings is proposed as a necessary condition of meaningfulness. The crucial important properties include, for instance, the existence of an analogue of the Cocke–Kasami–Younger algorithm, running in polynomial time. If such an algorithm exists, there is a chance that the resulting grammars inherit some other good properties; in particular, a few other parsing algorithms. An effort in that direction has recently been made by Barash and Okhotin [9], who extended conjunctive grammars with quantifiers for specifying contexts.

Besides looking for logically different models, one can consider a **stochastic variant** of the very same conjunctive and Boolean grammars. The general way of defining such a variant was already set by Ésik and Kuich [24], who developed related ideas in the context of least fixpoints of systems of language equations. Though Ésik and Kuich [24] restricted the underlying model of values assigned to strings to Boolean algebras, exactly the same approach should work for probabilities, and generally for semirings. Recently, Zier-Vogel and Domaratzki [145] defined *stochastic conjunctive grammars* and used them, along with the appropriately extended cubic-time parsing algorithm, to detect pseudo-knots in RNA. With these pioneering studies carried out, much work remains to be done in order to develop a conclusive theory of stochastic conjunctive and Boolean grammars. In particular, one would have to investigate learning algorithms for these families.

Another proposed subject of research is **implementation** of conjunctive and Boolean grammars. The goal is to produce a software tool to handle these grammars, which could be used by practitioners for parsing data in their programs. The parsing algorithms have been defined and theoretically analyzed, so it remains to adapt them to the needs of the users, to invent an appropriate input notation and internal representation of data and methods, and finally to develop the actual software. So far there have been a purely academic implementation of basic algorithms by the author [86], a GPU implementation of parsing by matrix multiplication by Okhotin and Reitwießner [111], and a Java-based GLR parser generator by Megacz [79].

And perhaps the most important direction for general investigation is identifying **applications** for conjunctive and Boolean grammars. This is a sphere of action for experts in applied areas, such as linguistics, software engineering or bioinformatics. The mathematical theory of formal grammars has now been developed towards a more expressive logic, which can still be conveniently and efficiently implemented. It is up to these experts to find ways of using this new theory, and thus motivate future research on formal grammars by more than a pure mathematical interest.

References

[1] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers: principles, techniques and tools*, Addison-Wesley, Reading, Mass., 1986.

[2] T. Aizikowitz, M. Kaminski, "Conjunctive grammars and alternating pushdown automata", *Acta Informatica*, 50:3 (2013), 175–197.

[3] T. Aizikowitz, M. Kaminski, "Linear conjunctive grammars and one-turn synchronized alternating pushdown automata", *Formal Grammar* (Bordeaux, France, July 25–26, 2009), LNAI 5591, 1–16.

[4] T. Aizikowitz, M. Kaminski, "LR(0) conjunctive grammars and deterministic synchronized alternating pushdown automata", *Computer Science in Russia* (CSR 2011, St. Petersburg, Russia, 14–18 June 2011), LNCS 6651, 345–358.

[5] N. Alon, M. Naor, "Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions", *Algorithmica*, 16:4–5 (1996), 434–449.

[6] R. Alur, P. Madhusudan, "Visibly pushdown languages", *ACM Symposium on Theory of Computing* (STOC 2004, Chicago, USA, 13–16 June 2004), 202–211.

[7] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, I. A. Faradzhev, "On economical construction of the transitive closure of an oriented graph", *Soviet Mathematics Doklady*, 11 (1970), 1209–1210.

[8] R. Balzer, "An 8-state minimal time solution to the firing squad synchronization problem", *Information and Control*, 10:1 (1967), 22–42.

[9] M. Barash, A. Okhotin, "Defining contexts in context-free grammars", *Language and Automata Theory and Applications* (LATA 2012, A Coruña, Spain, 5–9 March 2012), LNCS 7183, 106–118.

[10] Y. Bar-Hillel, M. Perles, E. Shamir, "On formal properties of simple phrase-structure grammars", *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14 (1961), 143–177.

[11] B. von Braunmühl, R. Verbeek, "Input driven languages are recognized in $\log n$ space", *Annals of Discrete Mathematics*, 24 (1985), 1–20.

[12] R. P. Brent, L. M. Goldschlager, "A parallel algorithm for context-free parsing", *Australian Computer Science Communications*, 6:7 (1984), 7.1–7.10.

[13] T. Buchholz, M. Kutrib, "On time computability of functions in one-way cellular automata", *Acta Informatica*, 35:4 (1998), 329–352.

[14] G. Buntrock, B. Jenner, K.-J. Lange, P. Rossmanith, "Unambiguity and fewness for logarithmic space", *Fundamentals of Computation Theory* (FCT 1991, Gosen, Germany, September 9–13, 1991), LNCS 529, 168–179.

[15] N. Chomsky, "On certain formal properties of grammars", *Information and Control*, 2:2 (1959), 137–167.

[16] S. A. Cook, "Deterministic CFL's are accepted simultaneously in polynomial time and log squared space", *11th Annual ACM Symposium on Theory of Computing* (STOC 1979, April 30–May 2, 1979, Atlanta, Georgia, USA), 338–345.

[17] D. Coppersmith, S. Winograd, "Matrix multiplication via arithmetic progressions", *Journal of Symbolic Computation*, 9:3 (1990), 251–280.

[18] K. Čulík II, "Variations of the firing squad problem and applications", *Information Processing Letters*, 30:3 (1989), 152–157.

[19] K. Čulík II, J. Gruska, A. Salomaa, "Systolic trellis automata", I and II, *International Journal of Computer Mathematics*, 15 (1984), 195–212, and 16 (1984), 3–22.

[20] K. Čulík II, J. Gruska, A. Salomaa, "Systolic trellis automata: stability, decidability and complexity", *Information and Control*, 71 (1986) 218–230.

[21] C. Dyer, "One-way bounded cellular automata", *Information and Control*, 44:3 (1980), 261–281.

[22] P. W. Dymond, "Input-driven languages are in $\log n$ depth", *Information Processing Letters*, 26 (1988), 247–250.

[23] J. Earley, "An efficient context-free parsing algorithm", *Communications of the ACM*, 13:2 (1970), 94–102.

[24] Z. Ésik, W. Kuich, "Boolean fuzzy sets", *International Journal of Foundations of Computer Science*, 18:6 (2007), 1197–1207.

[25] M. J. Fischer, L. J. Stockmeyer, "Fast on-line integer multiplication", *Journal of Computer and System Sciences*, 9:3 (1974), 317–331.

[26] Ph. Flajolet, "Analytic models and ambiguity of context-free languages", *Theoretical Computer Science*, 49 (1987), 283–309.

[27] E. P. Friedman, "The inclusion problem for simple languages", *Theoretical Computer Science*, 1:4 (1976), 297–316.

[28] M. Fürer, "Faster integer multiplication", *SIAM Journal on Computing*, 39:3 (2009), 979–1005.

[29] S. Ginsburg, S. A. Greibach, "Deterministic context-free languages", *Information and Control*, 9:6 (1966), 620–648.

[30] S. Ginsburg, H. G. Rice, "Two families of languages related to ALGOL", *Journal of the ACM*, 9 (1962), 350–371.

[31] S. Ginsburg, G. Rose, "Operations which preserve definability in languages", *Journal of the ACM*, 10:2 (1963), 175–195.

[32] S. Ginsburg, J. Ullian, "Preservation of unambiguity and inherent ambiguity in context-free languages", *Journal of the ACM*, 13:3 (1966), 364–368.

[33] S. L. Graham, M. A. Harrison, W. L. Ruzzo, "An improved context-free recognizer", *ACM Transactions of Programming Languages and Systems*, 2:3 (1980), 415–462.

[34] S. A. Greibach, "A new normal-form theorem for context-free phrase structure grammars", *Journal of the ACM*, 12 (1965), 42–52.

[35] S. A. Greibach, "The hardest context-free language", *SIAM Journal on Computing*, 2:4 (1973), 304–310.

[36] J. Gruska, "Descriptional complexity of context-free languages", *Mathematical Foundations of Computer Science* (MFCS 1973, Strbské Pleso, High Tatras, Czechoslovakia, 3-8 September 1973), 71–83.

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

[38] J. Hartmanis, "Context-free languages and Turing machine computations", *Proceedings of Symposia in Applied Mathematics*, Vol. 19, AMS, 1967, 42–51.

[39] S. Heilbrunner, L. Schmitz, "An efficient recognizer for the Boolean closure of context-free languages", *Theoretical Computer Science*, 80 (1991), 53–75.

[40] T. N. Hibbard, J. Ullian, "The independence of inherent ambiguity from complementedness among context-free languages", *Journal of the ACM*, 13:4 (1966), 588–593.

[41] M. Holzer, K.-J. Lange, "On the complexities of linear LL(1) and LR(1) grammars", *Fundamentals of Computation Theory* (FCT 1993, Hungary, August 23–27, 1993), LNCS 710, 299–308.

[42] J. E. Hopcroft, "On the equivalence and containment problems for context-free languages", *Theory of Computing Systems*, 3:2 (1969), 119–124.

[43] J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[44] D. T. Huynh, "Commutative grammars: the complexity of uniform word problems", *Information and Control*, 57:1 (1983), 21–39.

[45] O. H. Ibarra, T. Jiang, B. Ravikumar, "Some subclasses of context-free languages in NC^1", *Information Processing Letters*, 29:3 (1988), 111–117.

[46] O. H. Ibarra, T. Jiang, H. Wang, "A characterization of exponential-time languages by alternating context-free grammars", *Theoretical Computer Science*, 99:2 (1992), 301–313.

[47] O. H. Ibarra, S. M. Kim, "Characterizations and computational complexity of systolic trellis automata", *Theoretical Computer Science*, 29 (1984), 123–153.

[48] N. Immerman, "Relational queries computable in polynomial time", *Information and Control*, 68:1–3 (1986), 86–104.

[49] A. Jeż, "Conjunctive grammars can generate non-regular unary languages", *International Journal of Foundations of Computer Science*, 19:3 (2008), 597–615.

[50] A. Jeż, A. Okhotin, "Conjunctive grammars over a unary alphabet: undecidability and unbounded growth", *Theory of Computing Systems*, 46:1 (2010), 27–58.

[51] A. Jeż, A. Okhotin, "Complexity of equations over sets of natural numbers", *Theory of Computing Systems*, 48:2 (2011), 319–342.

[52] A. Jeż, A. Okhotin, "On the computational completeness of equations over sets of natural numbers", *35th International Colloquium on Automata, Languages and Programming* (ICALP 2008, Reykjavik, Iceland, July 7–11, 2008), 63–74.

[53] A. Jeż, A. Okhotin, "Equations over sets of natural numbers with addition only", *STACS 2009* (Freiburg, Germany, 26–28 February, 2009), 577–588.

[54] A. Jeż, A. Okhotin, "One-nonterminal conjunctive grammars over a unary alphabet", *Theory of Computing Systems*, 49:2 (2011), 319–342.

[55] A. Jeż, A. Okhotin, "Representing hyper-arithmetical sets by equations over sets of integers", *Theory of Computing Systems*, 51:2 (2012), 196–228.

[56] A. Jeż, A. Okhotin, "On the number of nonterminal symbols in unambiguous conjunctive grammars", *Descriptional Complexity of Formal Systems* (DCFS 2012, Braga, Portugal, 23–25 July 2012), LNCS 7386, 183–195.

[57] A. Jeż, A. Okhotin, "Unambiguous conjunctive grammars over a one-letter alphabet", *Developments in Language Theory* (DLT 2013, Paris, France, 18–21 June 2013), LNCS 7907, to appear.

[58] A. K. Joshi, L. S. Levy, M. Takahashi, "Tree adjunct grammars", *Journal of Computer and System Sciences*, 10:1 (1975), 136–163.

[59] T. Kasami, "An efficient recognition and syntax-analysis algorithm for context-free languages", Report AF CRL-65-758, Air Force Cambridge Research Laboratory, USA, 1965.

[60] T. Kasami, K. Torii, "A syntax-analysis procedure for unambiguous context-free grammars", *Journal of the ACM*, 16:3 (1969), 423–431.

[61] D. E. Knuth, "On the translation of languages from left to right", *Information and Control*, 8:6 (1965), 607–639.

[62] V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, "Well-founded semantics for Boolean grammars", *Information and Computation*, 207:9 (2009), 945–967.

[63] V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, "A game-theoretic characterization of Boolean grammars", *Theoretical Computer Science*, 412:12–14 (2011), 1169–1183.

[64] R. Kurki-Suonio, "Notes on top-down languages", *BIT Numerical Mathematics*, 9 (1969), 225–238.

[65] B. Lang, "Deterministic techniques for efficient non-deterministic parsers", *Automata, Languages and Programming* (ICALP 1974, Saarbrücken, July 29–August 2, 1974), LNCS 14, 255-269.

[66] M. Lange, "Alternating context-free languages and linear time μ-calculus with sequential composition", *Electronic Notes on Theoretical Computer Science*, 68:2 (2002), 70–86.

[67] M. Lange, "P-hardness of the emptiness problem for visibly pushdown languages", *Information Processing Letters*, 111:7 (2011), 338–341.

[68] M. Latta, R. Wall, "Intersective context-free languages", *Lenguajes Naturales y Lenguajes Formales IX*, Barcelona, 1993, 15–43.

[69] T. Lehtinen, "On equations $X+A = B$ and $(X+X)+C = (X-X) + D$ over sets of numbers", *Mathematical Foundations of Computer Science* (MFCS 2012, Bratislava, Slovakia, August 26–31 2012), LNCS 7464, 615–629.

[70] T. Lehtinen, A. Okhotin, "Boolean grammars and GSM mappings", *International Journal of Foundations of Computer Science*, 21:5 (2010), 799–815.

[71] T. Lehtinen, A. Okhotin, "On equations over sets of numbers and their limitations", *International Journal of Foundations of Computer Science*, 22:2 (2011), 377–393.

[72] T. Lehtinen, A. Okhotin, "On language equations $XXK = XXL$ and $XM = N$ over a unary alphabet", *Developments in Language Theory* (DLT 2010, London, Ontario, Canada, 17–20 August 2010), LNCS 6224, 291–302.

[73] T. Lehtinen, A. Okhotin, "Homomorphisms preserving deterministic context-free languages", *Developments in Language Theory* (DLT 2012, Taipei, Taiwan, 14–17 August 2012), LNCS 7410, 154–165.

[74] E. L. Leiss, "Unrestricted complementation in language equations over a one-letter alphabet", *Theoretical Computer Science*, 132 (1994), 71–93.

[75] P. M. Lewis II, R. E. Stearns, "Syntax-directed transduction", *Journal of the ACM*, 15:3 (1968), 465–488.

[76] P. M. Lewis II, R. E. Stearns, J. Hartmanis, "Memory bounds for recognition of context-free and context-sensitive languages", *IEEE Conference Record on Switching Circuit Theory and Logical Design*, 191–202, 1965.

[77] A. N. Maslov, "Estimates of the number of states of finite automata", *Soviet Mathematics Doklady*, 11 (1970), 1373–1375.

[78] A. N. Maslov, "Cyclic shift operation for languages", *Problems of Information Transmission*, 9 (1973), 333–338.

[79] A. Megacz, "Scannerless Boolean parsing", LDTA 2006, *Electronic Notes in Theoretical Computer Science*, 164:2 (2006), 97–102.

[80] K. Mehlhorn, "Pebbling mountain ranges and its application to DCFL-recognition", *Automata, Languages and Programming* (ICALP 1980, Noordweijkerhout, The Netherlands, 14–18 July 1980), LNCS 85, 422-435.

[81] E. Moriya, "A grammatical characterization of alternating pushdown automata", *Theoretical Computer Science*, 67:1 (1989), 75–85.

[82] Ch. Nomikos, P. Rondogiannis, "Locally stratified Boolean grammars", *Information and Computation*, 206:9–10 (2008), 1219–1233.

[83] A. Okhotin, "Conjunctive grammars", *Journal of Automata, Languages and Combinatorics*, 6:4 (2001), 519–535.

[84] A. Okhotin, "Top-down parsing of conjunctive languages", *Grammars*, 5:1 (2002), 21–40.

[85] A. Okhotin, "LR parsing for conjunctive grammars", *Grammars*, 5:2 (2002), 81–124.

[86] A. Okhotin, "Whale Calf, a parser generator for conjunctive grammars", *Implementation and Application of Automata* (CIAA 2002, Tours, France, July 3–5, 2002), LNCS 2608, 213–220.

[87] A. Okhotin, "Conjunctive grammars and systems of language equations", *Programming and Computer Software*, 28:5 (2002), 243–249.

[88] A. Okhotin, "On the closure properties of linear conjunctive languages", *Theoretical Computer Science*, 299 (2003), 663–685.

[89] A. Okhotin, "A recognition and parsing algorithm for arbitrary conjunctive grammars", *Theoretical Computer Science*, 302 (2003), 365–399.

[90] A. Okhotin, "The hardest linear conjunctive language", *Information Processing Letters*, 86:5 (2003), 247–253.

[91] A. Okhotin, "Efficient automaton-based recognition for linear conjunctive languages", *International Journal of Foundations of Computer Science*, 14:6 (2003), 1103–1116.

[92] A. Okhotin, "On the equivalence of linear conjunctive grammars to trellis automata", *RAIRO Informatique Théorique et Applications*, 38:1 (2004), 69–88.

[93] A. Okhotin, "On the number of nonterminals in linear conjunctive grammars", *Theoretical Computer Science*, 320:2–3 (2004), 419–448.

[94] A. Okhotin, "Boolean grammars", *Information and Computation*, 194:1 (2004), 19–48.

[95] A. Okhotin, "The dual of concatenation", *Theoretical Computer Science*, 345:2–3 (2005), 425–447.

[96] A. Okhotin, "On the existence of a Boolean grammar for a simple programming language", *Proceedings of AFL 2005* (May 17–20, 2005, Dobogókő, Hungary).

[97] A. Okhotin, "Generalized LR parsing algorithm for Boolean grammars", *International Journal of Foundations of Computer Science*, 17:3 (2006), 629–664.

[98] A. Okhotin, "Language equations with symmetric difference", *Fundamenta Informaticae*, 116:1–4 (2012), 205–222.

[99] A. Okhotin, "Homomorphisms preserving linear conjunctive languages", *Journal of Automata, Languages and Combinatorics*, 13:3–4 (2008), 299–305.

[100] A. Okhotin, "Seven families of language equations", *AutoMathA 2007*, Palermo, Italy, June 18–22, 2007.

[101] A. Okhotin, "Recursive descent parsing for Boolean grammars", *Acta Informatica*, 44:3–4 (2007), 167–189.

[102] A. Okhotin, "Nine open problems for conjunctive and Boolean grammars", *Bulletin of the EATCS*, 91 (2007), 96–119.

[103] A. Okhotin, "Unambiguous Boolean grammars", *Information and Computation*, 206 (2008), 1234–1247.

[104] A. Okhotin, "Decision problems for language equations", *Journal of Computer and System Sciences*, 76:3–4 (2010), 251–266.

[105] A. Okhotin, "Fast parsing for Boolean grammars: a generalization of Valiant's algorithm", *Developments in Language Theory* (DLT 2010, London, Ontario, Canada, August 17–20, 2010), LNCS 6224, 340–351.

[106] A. Okhotin, "A simple P-complete problem and its language-theoretic representations", *Theoretical Computer Science*, 412:1–2 (2011), 68–82.

[107] A. Okhotin, "Comparing linear conjunctive languages to subfamilies of the context-free languages", *SOFSEM 2011: Theory and Practice of Computer Science* (Nový Smokovec, Slovakia, 22–28 January 2011), LNCS 6543, 431–443.

[108] A. Okhotin, "Expressive power of LL(k) Boolean grammars", *Theoretical Computer Science*, 412:39 (2011), 5132–5155.

[109] A. Okhotin, "Inverse homomorphic characterizations of conjunctive and Boolean grammars", TUCS Technical Report 1080, Turku Centre for Computer Science, May 2013.

[110] A. Okhotin, C. Reitwießner, "Conjunctive grammars with restricted disjunction", *Theoretical Computer Science*, 411:26–28 (2010), 2559–2571.

[111] A. Okhotin, C. Reitwießner, "Parsing by matrix multiplication implemented on a CPU–GPU system", manuscript, April 2010.

[112] A. Okhotin, C. Reitwießner, "Parsing Boolean grammars over a one-letter alphabet using online convolution", *Theoretical Computer Science*, 457 (2012), 149–157.

[113] A. Okhotin, P. Rondogiannis, "On the expressive power of univariate equations over sets of natural numbers", *Information and Computation*, 212 (2012), 1–14.

[114] A. Okhotin, K. Salomaa, "State complexity of operations on input-driven pushdown automata", *Mathematical Foundations of Computer Science* (MFCS 2011, Warsaw, Poland, 22–26 August 2011), LNCS 6907, 485–496.

[115] A. Okhotin, O. Yakimova, "Language equations with complementation: Decision problems", *Theoretical Computer Science*, 376:1–2 (2007), 112–126.

[116] A. Okhotin, O. Yakimova, "Language equations with complementation: Expressive power", *Theoretical Computer Science*, 416 (2012), 71–86.

[117] T. Olshansky, A. Pnueli, "A direct algorithm for checking equivalence of LL(k) grammars" *Theoretical Computer Science*, 4:3 (1977), 321–349.

[118] T. Oshiba, "Closure property of the family of context-free languages under the cyclic shift operation", *Transactions of IECE*, 55D (1972), 119–122.

[119] A. J. Perlis, K. Samelson, ''Preliminary report: international algebraic language", *Communications of the ACM* 1:12 (1958), 8–22.

[120] W. Plandowski, W. Rytter, "Complexity of language recognition problems for compressed words", in: J. Karhumäki, H. A. Maurer, G. Păun, G. Rozenberg (Eds.), *Jewels are Forever*, Springer, 1999, 262–272.

[121] A. Reed, B. Kellogg, *Higher Lessons in English*, revised edition, 1896.

[122] D. J. Rosenkrantz, R. E. Stearns, "Properties of deterministic top-down grammars", *Information and Control*, 17 (1970), 226–256.

[123] W. C. Rounds, "LFP: A logic for linguistic descriptions and an analysis of its complexity", *Computational Linguistics*, 14:4 (1988), 1–9.

[124] W. L. Ruzzo, "On uniform circuit complexity", *Journal of Computer and System Sciences*, 22:3 (1981), 365–383.

[125] W. Rytter, "On the recognition of context-free languages", *Fundamentals of Computation Theory* (FCT 1985, Cottbus, Germany), LNCS 208, 315–322.

[126] S. Scheinberg, "Note on the boolean properties of context free languages", *Information and Control*, 3 (1960), 372-375.

[127] A. L. Semenov, "Algorithmic problems for power series and for context-free grammars", *Doklady Akademii Nauk SSSR*, 212 (1973), 50–52.

[128] G. Sénizergues, "$L(A) = L(B)$? decidability results from complete formal systems", *Theoretical Computer Science*, 251:1–2 (2001), 1–166.

[129] K. Sikkel, *Parsing Schemata*, Springer-Verlag, 1997.

[130] R. E. Stearns, H. B. Hunt III, "On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata", *SIAM Journal on Computing*, 14 (1985), 598–611.

[131] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik*, 13 (1969), 354–356.

[132] I. H. Sudborough, "A note on tape-bounded complexity classes and linear context-free languages", *Journal of the ACM*, 22:4 (1975), 499–500.

[133] A. Szabari, *Alternujúce Zásobníkové Automaty* (Alternating Pushdown Automata), in Slovak, diploma work (M.Sc. thesis), University of Košice (Czechoslovakia), 1991, 45 pp.

[134] V. Terrier, "On real-time one-way cellular array", *Theoretical Computer Science*, 141:1–2 (1995), 331–335.

[135] V. Terrier, "Closure properties of cellular automata", *Theoretical Computer Science*, 352:1–3 (2006), 97–107.

[136] M. Tomita, "An efficient augmented context-free parsing algorithm", *Computational Linguistics*, 13:1 (1987), 31–46.

[137] L. G. Valiant, "The equivalence problem for deterministic finite-turn pushdown automata", *Information and Control*, 25:2 (1974), 123–133.

[138] L. G. Valiant, "General context-free recognition in less than cubic time", *Journal of Computer and System Sciences*, 10:2 (1975), 308–314.

[139] M. Y. Vardi, "The complexity of relational query languages", *STOC 1982*, 137–146.

[140] D. Wood, "A further note on top-down deterministic languages", *Computer Journal*, 14:4 (1971), 396–403.

[141] D. Wotschke, "The Boolean closures of deterministic and nondeterministic context-free languages", In: W. Brauer (ed.), *Gesellschaft für Informatik e. V., 3. Jahrestagung 1973*, LNCS 1, 113–121.

[142] M. Wrona, "Stratified Boolean grammars", *Mathematical Foundations of Computer Science* (MFCS 2005, Gdansk, Poland, August 29–September 2, 2005), LNCS 3618, 801–812.

[143] D. H. Younger, "Recognition and parsing of context-free languages in time n^3", *Information and Control*, 10 (1967), 189–208.

[144] S. Yu, "A property of real-time trellis automata", *Discrete Applied Mathematics*, 15:1 (1986), 117–119.

[145] R. Zier-Vogel, M. Domaratzki, "RNA pseudoknot prediction through stochastic conjunctive grammars", CiE 2013, to appear.