Theoretical Computer Science 618 (2016) 52–71

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Input-driven languages are linear conjunctive ☆

Alexander Okhotin [1]

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

ARTICLE INFO

Article history:
Received 7 May 2015
Received in revised form 5 January 2016
Accepted 7 January 2016
Available online 12 January 2016
Communicated by D. Perrin

Keywords:
Context-free grammars
Conjunctive grammars
Cellular automata
Trellis automata
Pushdown automata
Input-driven automata
Visibly pushdown automata

ABSTRACT

Linear conjunctive grammars define the same family of languages as one-way real-time cellular automata (A. Okhotin, "On the equivalence of linear conjunctive grammars to trellis automata", *RAIRO ITA*, 2004), and this family is known to be incomparable with the context-free languages (V. Terrier, "On real-time one-way cellular array", *Theoret. Comput. Sci.*, 1995). This paper demonstrates the containment of the languages accepted by input-driven pushdown automata (a.k.a. visibly pushdown automata) in the family of linear conjunctive languages, which is established by a direct simulation of an input-driven automaton by a one-way real-time cellular automaton. On the other hand, it is shown that the language families defined by the unambiguous grammars, the LR(k) grammars and the LL(k) grammars are incomparable with the linear conjunctive languages.

1. Introduction

The theory of formal grammars is centred around a simple fundamental model, which is historically known under the name of a *context-free grammar*. Whereas other "context-sensitive" models proposed by Chomsky in his early papers were quickly found to be unrelated to syntactic descriptions of any kind, the context-free model was universally accepted as the standard mathematical model of syntax, that is, the ordinary kind of formal grammars. Many other practically important subfamilies of these grammars were quickly identified, such as the unambiguous grammars, the linear grammars, the LL(k) grammars [31], the LR(k) grammars [18], the bracketed grammars [12] and their variants. These models formed the core of formal language theory.

The following years saw many attempts at investigating formal grammars beyond the ordinary grammars. At first, there was no general theory on what kind of models would be suitable for representing syntax, and some successful definitions were initially based upon rather unobvious ideas, such as inserting subtrees into parse trees [17]. An understanding of these extensions came only in the 1980s, with the idea of parsing as deduction [30] leading to a remarkable paper by Rounds [32], who finally explained various kinds of grammars as fragments of the first-order logic over positions in a string, with recursive definitions under least fixed point semantics.

In light of this understanding, formal grammars are regarded as a specialized logic for describing the syntax of languages, where properties of strings (or their *syntactic categories*) are defined inductively by combining substrings with known properties. Then, ordinary grammars (Chomsky's "context-free") use substrings as basic objects, concatenation as the only function

☆ A preliminary version of this paper, entitled "Comparing linear conjunctive languages to subfamilies of the context-free languages" was presented at the SOFSEM 2011 conference held in Nový Smokovec, Slovakia, on 22–28 January 2011, and its extended abstract appeared in the conference proceedings.

E-mail address: alexander.okhotin@utu.fi.

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

http://dx.doi.org/10.1016/j.tcs.2016.01.007
0304-3975/

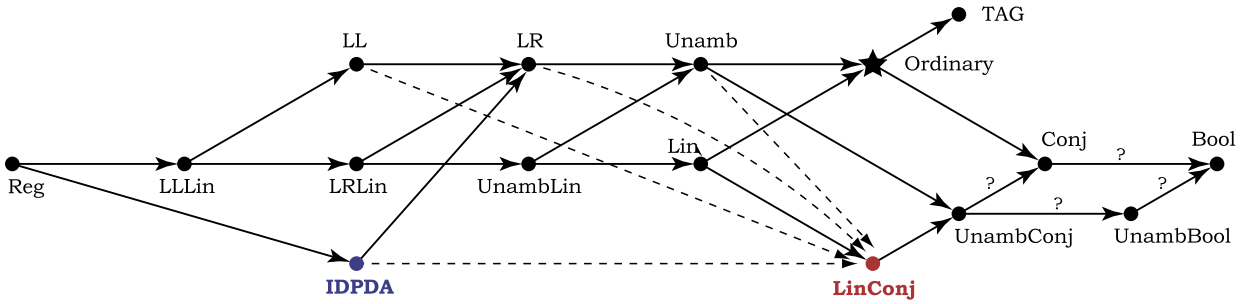

Fig. 1. The known hierarchy of formal grammars (solid lines) and possible inclusions to be investigated in this paper (dotted lines). The families depicted are those defined by finite automata (*Reg*), by (linear) LL(k) grammars (*LLLin*, *LL*), by (linear) LR(1) grammars (*LRLin*, *LR*), by (linear) unambiguous grammars (*UnambLin*, *Unamb*), by ordinary context-free grammars (*Ordinary*), by linear grammars (*Lin*), by tree-adjoining grammars (*TAG*), by input-driven pushdown automata (*IDPDA*), by (linear) conjunctive grammars (*LinConj*, *Conj*), by unambiguous conjunctive and Boolean grammars (*UnambConj*, *UnambBool*), and by Boolean grammars (*Bool*).

and disjunction as the only logical connective. Other kinds of grammars are variants of this definition, and are organized into the hierarchy depicted in Fig. 1, where each solid arrow represents containment of one family in another. These inclusions are known to be proper in all cases, except for those labelled with question marks.

The hierarchy is centred around the ordinary formal grammars (*Ordinary*). Their simplest extensions are those featuring extra logical connectives in the rules, namely, *conjunctive grammars* [22] (*Conj*) with conjunction, and *Boolean grammars* [25, 20] (*Bool*) further equipped with negation. Extensions of another kind maintain disjunction as the only logical connective, but extend the basic objects and the operations on them: the simplest of these grammars are the *tree-adjoining grammars* [17] (*TAG*), which deal with pairs of strings instead of strings, and use wrapping of one pair around another as the only function. The latter model and its further extensions received much attention in computational linguistics.

Other families presented in Fig. 1 restrict these families in one or another way. First, there are the unambiguous subclasses of these grammars (*Unamb*, *UnambConj*, *UnambBool*), which define a unique parse for every string they generate; these grammars are notable for having square-time parsing algorithms [26]. Further below, there are left-to-right deterministic variants of grammars that admit linear-time parsing: the *LR*(k) *grammars* [18] (*LR*), also known for their equivalent representation by deterministic pushdown automata [11], and the *LL*(k) *grammars* [31] (*LL*), which can be parsed by recursive descent with k symbols of lookahead. Another small subfamily of the LR(k) grammars are the *bracketed grammars* [12], which later evolved into *input-driven pushdown automata* [21] (*IDPDA*), also known under the name of *visibly pushdown automata* [2]. The hierarchy also features subclasses of some of these families, in which the concatenation is restricted to linear (*LLLin*, *LRLin*, *UnambLin*, *LinConj*). The largest of these is the family of linear conjunctive languages.[2]

This purpose of this paper is to investigate the hierarchy of formal grammars presented in Fig. 1, and to settle four previously unresolved relations marked in the figure by dotted arrows. Each of these relations might turn out to be a proper containment, although incomparability would appear more likely, because those grammar families are defined by incomparable restrictions to the underlying logic.

As shown in Fig. 1, all results in this paper are concerned with possible containment of other language families in the family of linear conjunctive languages (*LinConj*). Consider first the whole family of *conjunctive grammars* [22], which extend the ordinary grammars with an additional conjunction operation that expresses a substring satisfying multiple conditions simultaneously. These grammars have greater expressive power than the ordinary grammars, and yet inherit their main practical properties, such as subcubic-time parsing [28], and offer a new field for theoretical studies, which is elaborated in a recent survey paper [27]. Their special case, in which concatenation can be taken only with terminal strings, is known as *linear conjunctive grammars* [22].

Linear conjunctive grammars are notable for having an equivalent representation by one of the simplest types of cellular automata. These are the *one-way real-time cellular automata*, also known as *trellis automata*, studied by Dyer [9], Čulík, Gruska and Salomaa [7,8], Ibarra and Kim [13], Terrier [34], and others. These automata work in real time, making $n-1$ parallel steps on an input of length n, and the next value of each cell is determined only by its own value and the value of its right neighbour. Precise definitions of linear conjunctive grammars and trellis automata are recalled in Section 2, along with the computational equivalence result [23].

The main result of this paper is that every language recognized by an input-driven pushdown automaton (*IDPDA*) is linear conjunctive (*LinConj*), so that one of the dotted lines in Fig. 1 becomes a solid line. An *input-driven pushdown automaton* (IDPDA) is a special case of a deterministic pushdown automaton, in which the input alphabet Σ is split into three disjoint subsets, Σ_{+1}, Σ_{-1} and Σ_0, and the type of the input symbol determines the type of the operation with the stack (push, pop, or ignore, respectively). These automata develop the idea behind *bracketed grammars* [12]; they were first studied by

[2] It is interesting to note that none of these models allow context-dependent rules of any kind. This makes Chomsky's term "context-free grammar" redundant, as each grammar family in Fig. 1 could be nicknamed "context-free". For that reason, this paper refers to Chomsky's "context-free grammars" as the ordinary grammars.

Mehlhorn [21], von Braunmühl and Verbeek [5], Dymond [10] and Rytter [33], who showed that the languages they recognize have very low computational complexity (to be precise, are in NC^1). Von Braunmühl and Verbeek [5] also demonstrated that the nondeterministic variant of the model is equal in power to the deterministic one. Later, input-driven automata were rediscovered and further studied by Alur and Madhusudan [2], whose work inspired a renewed interest in the model (often under the names "visibly pushdown automata" and "nested word automata").

The definition of input-driven automata is given in Section 3. Before showing how any input-driven automaton can be simulated by a trellis automaton, the first to be presented is a model trellis automaton that parses bracketed structures, which is constructed and explained in Section 4. Then, in Section 5, this model automaton is augmented to simulate the computation of any given input-driven automaton on that bracketed structure. The main difficulty in this simulation is that IDPDA are deterministic only when executed sequentially, as per their definition, whereas, on the other hand, a trellis automaton attains its full power on parallel computations. Hence, the deterministic computation of an IDPDA has to be simulated as if it were nondeterministic: the trellis automaton does not know the state, in which the IDPDA begins processing each substring, and so it calculates the result of the IDPDA's computation beginning with every possible state. These computed *behaviours on substrings* are gradually combined, until the behaviour on the entire string is obtained.

Having found a family of grammars that can be simulated by trellis automata, it is natural to ask whether any larger grammar families can be simulated as well. For the full family of ordinary grammars, it is known from Terrier [34] that it is incomparable with the linear conjunctive grammars. However, the witness language given by Terrier is an inherently ambiguous language [26], and therefore the three families down the hierarchy—that is, the unambiguous grammars, the LL grammars and the LR grammars—could still potentially be simulated by linear conjunctive grammars, as indicated by dotted arrows in Fig. 1. The second result of this paper, established in Section 6, is that this is not the case, and all three language families are actually incomparable with the linear conjunctive languages. This is done by presenting an LL(1) language that is not linear conjunctive; the proof of the latter fact is carried out using a lemma by Terrier [34].

The paper is concluded with a revised hierarchy of formal grammars, and with an analysis of incomparability between all pairs of grammar families not connected by a directed path in the hierarchy.

2. Linear conjunctive grammars and trellis automata

All families of formal languages considered in this paper are subsets of the family defined by *conjunctive grammars*, which extend ordinary ("context-free") grammars by allowing a conjunction of syntactical conditions in any rules.

Definition 1. (See Okhotin [22].) A conjunctive grammar is a quadruple $G = (\Sigma, N, R, S)$, in which:

- a finite set Σ is the alphabet of the language being defined;
- another finite set N contains the symbols for syntactic categories defined in the grammar, which are historically called *nonterminal symbols;*
- R is a finite set of grammar rules, each of the form
$$A \to \alpha_1 \,\&\, \ldots \,\&\, \alpha_n, \tag{1}$$
with $A \in N$, $n \geqslant 1$ and $\alpha_1, \ldots, \alpha_n \in (\Sigma \cup N)^*$;
- $S \in N$ is the *initial symbol* of the grammar, representing the set of well-formed sentences of the language.

A rule (1) expresses that every string that is described by each α_i is hence described by A. This understanding can be formalized in two equivalent ways. One definition employs term rewriting that generalizes Chomsky's string rewriting: according to it, a rule (1) allows rewriting A with a term $(\alpha_1 \& \ldots \& \alpha_n)$ over concatenation and conjunction, and furthermore, one can rewrite a conjunction $(w \& \ldots \& w)$ of identical terminal strings with a single such string w [22]. The other definition is based on language equations with nonterminal symbols as unknown languages, and it uses the least solution of the following system to define the languages generated by these nonterminals.

$$A = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_n \in R} \bigcap_{i=1}^{n} \alpha_i \qquad \text{(for all } A \in N\text{)}$$

These definitions are illustrated in the below example.

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

$$\begin{aligned} S &\to A \& C \\ A &\to aA \mid B \\ B &\to bBc \mid \varepsilon \\ C &\to Cc \mid D \\ D &\to aDb \mid \varepsilon \end{aligned}$$

According to the definition by term rewriting, the string $w = abc$ can be generated as follows.

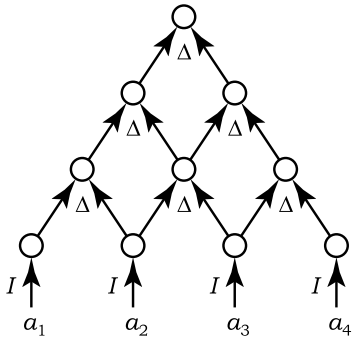

Fig. 2. Computation carried out by a trellis automaton.

$S \Longrightarrow (A\,\&\,C) \Longrightarrow (aA\,\&\,C) \Longrightarrow (aB\,\&\,C) \Longrightarrow (abBc\,\&\,C) \Longrightarrow (abc\,\&\,C) \Longrightarrow (abc\,\&\,Cc) \Longrightarrow (abc\,\&\,Dc) \Longrightarrow (abc\,\&\,aDbc) \Longrightarrow (abc\,\&\,abc) \Longrightarrow abc$

The definition by language equations represents the grammar as the following system of equations with five unknowns.

$$S = A \cap C$$
$$A = (\{a\}A) \cup B$$
$$B = (\{b\}B\{c\}) \cup \{\varepsilon\}$$
$$C = (C\{c\}) \cup D$$
$$D = (\{a\}D\{b\}) \cup \{\varepsilon\}$$

Its least solution is $S = \{a^n b^n c^n \mid n \geqslant 0\}$, $A = \{a^i b^n c^n \mid i, n \geqslant 0\}$, $B = \{b^n c^n \mid \ell \geqslant 0\}$, $C = \{a^n b^n c^j \mid n, j \geqslant 0\}$, $D = \{a^n b^n \mid n \geqslant 0\}$, which is exactly what one would obtain by term rewriting.

The family of conjunctive grammars has two important special cases. One of them is the case of *ordinary grammars* (Chomsky's "context-free"), in which every rule (1) has one conjunct, that is, $n = 1$. The other case are the *linear conjunctive grammars*, in which every conjunct α in every rule (1) may contain at most one nonterminal symbol; the grammar in the above Example 1 is actually linear conjunctive. Grammars satisfying both restrictions are known as *linear*. Ordinary grammars and linear conjunctive grammars are known to be incomparable in power [34], and this result shall be elaborated in Section 6 of this paper. Although the more general Boolean grammars [25,20] are not considered in this paper, it is worth noting that linear Boolean grammars define the same class of languages as linear conjunctive grammars [23].

In Example 1, conjunction is only used on top of ordinary grammars to define intersection of languages. By *iterating the conjunction*, one can obtain much more interesting definitions of quite a few non-trivial languages not described by ordinary grammars, such as $\{wcw \mid w \in \{a, b\}^*\}$ [22], the language of checking declarations before their use [27, Ex. 3], the unary language $\{a^{4^n} \mid n \geqslant 0\}$ [14] and other unary languages [15], as well as some P-complete languages [27, Ex. 21]. At the same time, conjunctive grammars retain the important practical properties of ordinary grammars, in particular, parse trees and efficient parsing algorithms [26,28]. For more information, an interested reader is directed to a recent survey paper [27].

The family of languages described by linear conjunctive grammars is remarkable for being the same as the family recognized by one of the most basic kinds of cellular automata. These are the *one-way real-time cellular automata* [9], also known as *trellis automata* [7,8]. A trellis automaton processes an input string of length $n \geqslant 1$ in a uniform triangular array of $\frac{n(n+1)}{2}$ processor nodes, connected as in Fig. 2. Each node computes a value from a fixed finite set $\mathcal{S}$. The nodes in the bottom row obtain their values directly from the input symbols, whereas the rest of the nodes compute the same function Δ mapping the states of their two predecessors to a new state. The string is accepted if and only if the value computed at the pinnacle belongs to the set of accepting states.

Definition 2. (See Dyer [9]; Čulík, Gruska and Salomaa [7].) A trellis automaton is a quintuple $M = (\Sigma, \mathcal{S}, \mathcal{I}, \Delta, \mathcal{F})$, where:

- Σ is the input alphabet,
- $\mathcal{S}$ is a finite non-empty set of states,
- $\mathcal{I}\colon \Sigma \to \mathcal{S}$ is a function that sets the states in the bottom row,
- $\Delta\colon \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ is the transition function, and
- $\mathcal{F} \subseteq \mathcal{S}$ is the set of accepting states.

In order to describe the behaviour of the automaton, the function Δ is extended to transform a string of states $\alpha \in \mathcal{S}^+$ forming the bottom of a triangle to a single state at its pinnacle, formalizing the connections in Fig. 2. The value of this extended function $\Delta\colon \Sigma^+ \to \mathcal{S}$ is defined inductively on the length of a string of states, as follows.

$$\Delta(q) = q$$
$$\Delta(p\alpha q) = \Delta\big(\Delta(p\alpha), \Delta(\alpha q)\big)$$

Similarly, the function $\mathcal{I}$ is extended to transform an input string to a string of states, as $\mathcal{I}(a_1 \ldots a_\ell) = \mathcal{I}(a_1) \ldots \mathcal{I}(a_\ell)$. Then the language recognized by the automaton is defined as $L(M) = \{w \in \Sigma^+ \mid \Delta(\mathcal{I}(w)) \in \mathcal{F}\}$.

Theorem A. *(See Okhotin [23,24].) A language $L \subseteq \Sigma^+$ is described by a linear conjunctive grammar if and only if L is recognized by a trellis automaton. Furthermore, every such language is described by a linear conjunctive grammar with two nonterminal symbols.*

The proof of Theorem A is by an effective construction of a trellis automaton out of a grammar, and vice versa.

The main language-theoretic properties of linear conjunctive languages were established using trellis automata by Čulík et al. [7,8], by Ibarra and Kim [13] and by Terrier [34]. For more information, an interested reader is referred to a survey of conjunctive grammars [27, Sect. 4]. One small result that is particularly helpful in this paper is the closure of the linear conjunctive languages under quotient with one-symbol strings.

Lemma B. *(See Čulík et al. [7,8].) For every linear conjunctive language $L \subseteq \Sigma^*$ and for every symbol $a \in \Sigma$, the languages $a^{-1} \cdot L = \{w \mid aw \in L\}$ and $L \cdot a^{-1} = \{w \mid wa \in L\}$ are linear conjunctive as well.*

3. Input-driven automata

In an input-driven pushdown automaton [21,2], the input alphabet Σ is split into three disjoint subsets, Σ_{+1}, Σ_{-1} and Σ_0, and the type of the input symbol determines the type of the operation with the stack that the automaton must perform. On each symbol from Σ_{+1}, the automaton always pushes one symbol onto the stack. If the input symbol is in Σ_{-1}, the automaton pops one symbol. Finally, for a neutral symbol in Σ_0, the automaton may not use the stack: that is, neither modify it, nor even examine its contents.

Symbols from Σ_{+1} and from Σ_{-1} are known as *left brackets* and *right brackets*, respectively, and are denoted by angled bracket symbols ($<$, $>$). Symbols from Σ_0 are called *neutral symbols*. When an input-driven automaton pushes a symbol onto the stack upon reading a left bracket, it is guaranteed to pop that symbol from the stack upon reading the matching right bracket. A string over this alphabet is called *well-nested*, if each bracket has its matching counterpart. The set of all well-nested strings is defined by a grammar with the following rules.

$$S \to \varepsilon$$
$$S \to <S> \qquad (< \in \Sigma_{+1},\ > \in \Sigma_{+1})$$
$$S \to cS \qquad (c \in \Sigma_0)$$
$$S \to SS$$

In their standard form, input-driven automata operate only on well-nested strings. In order to handle ill-nested strings as well, Alur and Madhusudan [2] adapted the definition as follows. On an unmatched left bracket, an input-driven automaton pushes a symbol that it shall never pop, and which accordingly shall not affect its acceptance decision. An unmatched right bracket is encountered with an empty stack, and for such a case the automaton is equipped with a special type of transitions that detect the emptiness of the stack but do not modify it. This extended definition is assumed in this paper.

Definition 3. (See Mehlhorn [21]; Alur and Madhusudan [2].) An input-driven pushdown automaton (IDPDA) is an undecuple $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0, Q, q_0, \Gamma, (\delta_<)_{<\in\Sigma_{+1}}, (\gamma_<)_{<\in\Sigma_{+1}}, (\delta_>)_{>\in\Sigma_{-1}}, (\delta_c)_{c\in\Sigma_0}, F)$, where:

- $\Sigma = \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ is an input alphabet split into three disjoint classes;
- Q is the set of states of the automaton, with an initial state $q_0 \in Q$ and a subset of accepting states $F \subseteq Q$;
- Γ is a finite pushdown alphabet, and a special symbol $\bot \notin \Gamma$ denotes an empty stack;
- for each left bracket symbol $< \in \Sigma_{+1}$, the behaviour of the automaton is described by a pair of complete functions $\delta_<\colon Q \to Q$ and $\gamma_<\colon Q \to \Gamma$, which provide the next state and the symbol to be pushed onto the stack, respectively;
- for every right bracket symbol $> \in \Sigma_{-1}$, a complete function $\delta_>\colon Q \times (\Gamma \cup \{\bot\}) \to Q$ specifies the next state, assuming that the given stack symbol is popped from the stack or the stack is empty ($\bot$);
- for each neutral symbol $c \in \Sigma_0$, the state change is described by a complete function $\delta_c\colon Q \to Q$.

A configuration of A is a triple (q, w, x), where $q \in Q$ is the current state, $w \in \Sigma^*$ is the remaining input and $x \in \Gamma^*$ is the stack contents. The initial configuration on an input string $w_0 \in \Sigma^*$ is (q_0, w_0, ε). For each configuration with at least one remaining input symbol, the next configuration is uniquely determined as follows, for any $q \in Q$, $w \in \Sigma^*$ and $x \in \Gamma^*$.

$$\begin{aligned}
(q, {<}w, x) &\vdash_A (\delta_{<}(q), w, \gamma_{<}(q)x), && \text{for } < \in \Sigma_{+1}\\
(q, {>}w, \gamma x) &\vdash_A (\delta_{>}(q, \gamma), w, x), && \text{for } > \in \Sigma_{-1} \text{ and } \gamma \in \Gamma\\
(q, {>}w, \varepsilon) &\vdash_A (\delta_{>}(q, \bot), w, \varepsilon), && \text{for } > \in \Sigma_{-1}\\
(q, cw, x) &\vdash_A (\delta_c(q), w, x), && \text{for } c \in \Sigma_0
\end{aligned}$$

Once the input string is exhausted, the last configuration (q, ε, x) is accepting if $q \in F$, regardless of the stack contents. The language $L(A)$ recognized by the automaton is the set of such all strings $w \in \Sigma^*$, that the computation beginning from (q_0, w, ε) is accepting.

For example, one can construct an input-driven automaton for the language $\{a^n b^n \mid n \geqslant 0\} \cup \{a^n c^n \mid n \geqslant 0\}$, under the partition of the alphabet $\{a, b, c\}$ into $\Sigma_{+1} = \{a\}$, $\Sigma_{-1} = \{b, c\}$ and $\Sigma_0 = \varnothing$. For any other partition of this alphabet, recognizing this language by an IDPDA is impossible. As another example, consider that the language $\{a^n b^n \mid n \geqslant 0\} \cup \{b^n a^n \mid n \geqslant 0\}$ over the alphabet $\Sigma = \{a, b\}$ is not an input-driven language for any partition of the alphabet.

For more information about input-driven automata, an interested reader is directed to a recent survey by Okhotin and Salomaa [29].

4. A trellis automaton for parsing well-nested strings

Before approaching the construction of a trellis automaton simulating any given IDPDA, its main idea shall be illustrated on a simpler example: a trellis automaton that parses every well-nested string over an alphabet $\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ according to its structure.

Note that the set of well-nested strings is similar to the Dyck language augmented with neutral symbols occurring anywhere in a string. Trellis automata recognizing the Dyck language are known from Dyer [9], and the following example adapts Dyer's construction to handle neutral symbols.

Example 2. (Adapted from Dyer [9, Thm. 3].) Let $\Sigma = \{<, >, c\}$ be an alphabet, with a partition $\Sigma_{+1} = \{<\}$, $\Sigma_{-1} = \{>\}$ and $\Sigma_0 = \{c\}$. Then a trellis automaton with the set of states $\mathcal{S} = \{\nearrow, \nwarrow, \square, -\}$ and with the following transition function Δ carries out computations sufficient to parse the structure of every well-nested string enclosed within a pair of brackets.

Δ	$\nearrow$	$\nwarrow$	$\square$	$-$
$\nearrow$	$\nearrow$	$\square$	$\nearrow$	
$\nwarrow$	$-$	$\nwarrow$	$-$	$-$
$\square$	$\square$	$\nwarrow$	$\square$	$\square$
$-$	$-$	$\nwarrow$	$-$	$-$

Its initial function is defined by $\mathcal{I}(<) = \nearrow$, $\mathcal{I}(>) = \nwarrow$, $\mathcal{I}(c) = \square$.

Fig. 3 illustrates the computation of this automaton on a well-nested input string $w = {<}w_1 w_2 w_3 w_4{>}$, where $w_1 = {<}{<}{>}{>}$, $w_2 = {<}{>}$, $w_3 = c$ and $w_4 = {<}c{<}{>}{>}$ are its well-nested substrings.

Note that accepting states are not defined for this automaton, as it is not made to recognize any particular language: its only purpose is to arrange the information transfer illustrated in Fig. 3.

This automaton has to be explained in all detail, because later it shall be augmented to simulate any given trellis automaton. Its states have the following meaning. The state denoted by a *right arrow* ($\nearrow$) represents a left bracket ($<$) looking for a matching right bracket ($>$) to the right, and similarly, the *left arrow* ($\nwarrow$) represents a right bracket ($>$) looking for a matching right bracket ($<$) to the left. Once these arrows meet, they produce a *box* ($\square$), which indicates a substring from L_0. Boxes are propagated to the right, until they meet a diagonal of left arrows ($\nwarrow$), into which they join. Finally, the *blank state* ($-$) is used in all other places.

Let L_0 be the language of all non-empty well-nested strings over Σ that are not representable as concatenations of two non-empty well-nested strings. This language is described by the following grammar.

$$\begin{aligned}
S &\to {<}A{>} \mid c\\
A &\to SA \mid \varepsilon
\end{aligned}$$

The states reached by the trellis automaton constructed above on different substrings are described in the following lemma.

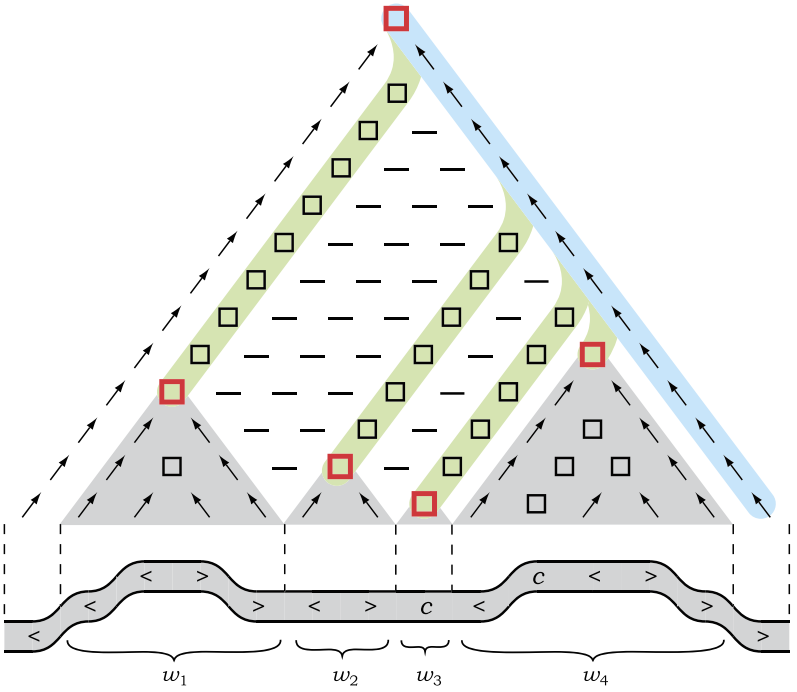

Fig. 3. A four-state trellis automaton from Example 2 operating on a well-nested string $w = <<<>>><>c<c<>>>$.

Lemma 1. *The automaton in Example 2 computes a right arrow ($\nearrow$) on each non-empty proper prefix of a string in L_0, and a left arrow ($\nwarrow$) on each non-empty proper suffix. On each string in L_0—and, more generally, on each string of the form uv, where $u \in L_0$ and v is any prefix of a string in L_0^+—the automaton computes a box ($\square$). The blank state ($-$) is computed on all remaining strings.*

Proof. Before presenting a formal proof that the automaton works on every input string as specified, its computation shall be explained on strings belonging to L_0. This is done inductively on the structure of strings in L_0.

For a one-symbol string $c \in L_0$, the initial function gives the intended result, a box ($\square$).

For a string of the form $<w_1 \dots w_k>$, with $k \geqslant 0$ and $w_1, \dots, w_k \in L_0$, according to the plan, the automaton computes a box ($\square$) on each w_i, as well as right arrows ($\nearrow$) on all its proper prefixes and left arrows ($\nwarrow$) on all proper suffixes. The task is to ensure that these boxes are propagated to the right, until they meet the left arrows ($\nwarrow$) spawned from the last right bracket ($>$), and all remaining space is filled with blanks ($-$), all as shown in Fig. 3.

The computations carried out between every two subsequent substrings w_i and w_{i+1} depend on the types of both substrings, that is, whether each of them is an individual neutral symbol or something enclosed in brackets. For two substrings in brackets, $w_i = <w>$ and $w_{i+1} = <w'>$, the middle zone between them is filled with blanks by the following transitions.

$$\Delta(\nwarrow, \nearrow) = - \tag{2a}$$
$$\Delta(\nwarrow, -) = - \tag{2b}$$
$$\Delta(-, \nearrow) = - \tag{2c}$$
$$\Delta(-, -) = - \tag{2d}$$
$$\Delta(-, \square) = - \tag{2e}$$

The box ($\square$) computed at the pinnacle of w_i is then propagated right–upward.

$$\Delta(\square, -) = \square \tag{2f}$$

If $w_i = c$ and $w_{i+1} = c$, then there are two parallel diagonals of boxes ($\square$) made by the following transition.

$$\Delta(\square, \square) = \square \tag{2g}$$

If $w_i = c$ and $w_{i+1} = <w>$, then a diagonal of boxes ($\square$) goes along a diagonal of right arrows ($\nearrow$) by another type of transitions.

$$\Delta(\square, \nearrow) = \square \tag{2h}$$

One more type of transitions is used when $w_i = <w>$ and $w_{i+1} = c$.

$$\Delta(\nwarrow, \square) = - \tag{2i}$$

This concludes the propagation of signals from all w_i.

On the right border, the signals coming from each w_i flow into the diagonal of left arrows ($\nwarrow$) spawned from the right bracket ($>$).

$$\Delta(\square, \nwarrow) = \nwarrow, \tag{2j}$$

$$\Delta(-, \nwarrow) = \nwarrow \tag{2k}$$

If w_k is of the form $<w>$, then two diagonals of left arrows ($\nwarrow$) go along each other by the following transition.

$$\Delta(\nwarrow, \nwarrow) = \nwarrow \tag{2l}$$

The left border is maintained by the following two transitions, the second of which is used only for $w_1 = <w>$.

$$\Delta(\nearrow, \square) = \nearrow, \tag{2m}$$

$$\Delta(\nearrow, \nearrow) = \nearrow \tag{2n}$$

Finally, once the left diagonal meets the right diagonal, they produce a box at the pinnacle.

$$\Delta(\nearrow, \nwarrow) = \square \tag{2o}$$

This completes the computations done at each level of brackets.

A formal proof of the statement of the lemma is carried out by induction on the length of an input string $w \in \{<, >, c\}^*$.

Base case I: $w = c$. The initial function produces the desired state $I(c) = \square$.

Base case II: $w = <$. This string is a non-empty proper prefix of the string $w' = <>$ from L_0, and accordingly, the initial function gives the right arrow: $I(<) = \nearrow$.

Base case III: $w = >$. This is a non-empty proper suffix of the same string $<>$, and the initial function produces the left arrow: $I(<) = \nwarrow$.

Induction step (prefix). Let w be a proper prefix of a string $<w_1 \dots w_k>$, where $k \geqslant 0$ and $w_1, \dots, w_k \in L_0$. Assume that w contains at least two symbols, and let $w = <x\tau$, where $\tau \in \{<, >, c\}$ is the last symbol of w and $x \in \{<, >, c\}^*$ is the middle part. The shorter prefix $<x$ is also a proper prefix of a string in L_0, and therefore, by the induction hypothesis, on the string $<x$, the automaton computes a right arrow state ($\nearrow$).

In order to determine the state computed on $x\tau$, first, consider the case when $x\tau$ contains the entire substring w_1 and zero or more symbols from $w_2 \dots w_k$. Then, $x\tau$ is an element of L_0 followed by a prefix of a string in L_0^+, and, by the induction hypothesis, the automaton enters a box-state ($\square$) on $x\tau$. The state on w is then computed by one of the automaton's transitions (2m), as follows.

$$\Delta(w) = \Delta(\Delta(<x), \Delta(x\tau)) = \Delta(\nearrow, \square) = \nearrow$$

The remaining possibility is that $x\tau$ is a proper prefix of w_1. The induction hypothesis then asserts that the state computed by the automaton on $x\tau$ is the right arrow ($\nearrow$). Then, another right arrow ($\nearrow$) is computed on w by the corresponding transition (2n).

Induction step (suffix). The proof is similar to the case of a prefix. This time, w is a proper suffix of $<w_1 \dots w_k>$. Let $w = \sigma x>$, with $\sigma \in \{<, >, c\}$ and $x \in \{<, >, c\}^*$. It is claimed that on w, the automaton computes a left arrow state ($\nwarrow$). The exact transition used to compute it depends on where σx begins in relation to the boundaries between the substrings $w_1, \dots, w_k$.

First, if σx begins exactly at the boundary between w_{i-1} and w_i, for some $i \in \{1, \dots, k\}$, that is, if $\sigma x = w_i \dots w_k$, this means that it is a concatenation of a string in L_0 and a prefix of a string in L_0^+. Then, $\Delta(\sigma x) = \square$ by the induction hypothesis. Next, since $x>$ is a proper suffix of a string in L_0, by the induction hypothesis, the state computed on $x>$ is a left arrow ($\nwarrow$). Then the transition (2j) computes a left arrow ($\nwarrow$) on w.

Secondly, if σx is a proper suffix of w_k, then $\Delta(\sigma x) = \nwarrow$, and $\Delta(x>) = \nwarrow$. A left arrow state ($\nwarrow$) is then computed on w by the transition (2l).

The third case is when σx breaks one of the substrings w_i, with $i \in \{1, \dots, k-1\}$, into two non-empty parts, that is, $\sigma x = y w_{i+1} \dots w_k$, where y is a non-empty proper suffix of w_i. This string σx is not of any of the first three forms listed in the statement of the lemma. Indeed, it cannot be a prefix of any string in L_0^+, because y contains unmatched right brackets ($>$); it is neither a suffix of a string in L_0, for the reason that no string in L_0 may have a suffix from L_0, whereas σx has a suffix w_k. Therefore, by the induction hypothesis, $\Delta(\sigma x) = -$. The state computed on the string $x>$ is again a left arrow ($\nwarrow$), and the transition (2k) produces another left arrow state ($\nwarrow$) on w.

Induction step (string from L_0). Let $w = <w_1 \dots w_k>$ be a string belonging to L_0, with $k \geqslant 0$ and $w_1, \dots, w_k \in L_0$. This makes its prefix $<w_1 \dots w_k$ a proper non-empty prefix of a string in L_0, and then, by the induction hypothesis, the automaton computes a right arrow ($\nearrow$) on this string. On the suffix $w_1 \dots w_k>$, similarly, the automaton produces a left arrow ($\nwarrow$). Then, on w, the transition function (2o) computes a box ($\square$).

Induction step (L_0 followed by a prefix of L_0^+). Let $w = uv$, where $u \in L_0$, and v is a non-empty prefix of a string from L_0^+. Let τ be the last symbol of v, so that $v = v'\tau$. Since v' is a also prefix of a string from L_0^+, by the induction hypothesis, the automaton computes a box ($\square$) on uv'. The goal is to show that a box state ($\square$) is also computed on w.

Let σ be the first symbol of u, and let $u = \sigma u'$. The question is to determine the state computed on $u'v$. This state depends on whether u is a single symbol (with two subcases depending on v) or a substring enclosed in brackets.

- Assume that $u = c$, so that $\sigma = c$, $u' = \varepsilon$ and $w = cv$, and further assume that v is a proper prefix of a string in L_0. Then, by the induction hypothesis, $\Delta(\mathcal{I}(v)) = \nearrow$, and then corresponding transition (2h) produces the required box state ($\square$) on w.
- If $u = c$, but v is *not* a proper prefix of a string in L_0, then it must be a concatenation of a full string from L_0 with a prefix of a string from L_0^+. In this case, $\Delta(\mathcal{I}(v)) = \square$, and the box state ($\square$) is then obtained for w by the transition (2g).
- If $u = <u_1 \dots u_k>$, with $k \geqslant 0$ and $u_1, \dots, u_k \in L_0$, then $\sigma = <$ and $u' = u_1 \dots u_k>$, and the substring in question is $u_1 \dots u_k > v$, where v is non-empty. This substring is neither a prefix of any string in L_0^+, nor a suffix of any string in L_0. Therefore, by the induction hypothesis, the automaton computes a blank state ($-$) on it. Then the transition (2f) produces the box ($\square$).

Induction step (strings of any other form). For a string w that is neither a prefix of any string in L_0^+, nor a suffix of any string in L_0, the goal is to show that the automaton computes a blank state ($-$) on w. Let $w = \sigma x \tau$, where σ and τ may be any symbols from the alphabet $\{<, >, c\}$. The proof of this case proceeds by showing which states *cannot* be computed on the substrings σx and $x\tau$.

The first claim is that the automaton cannot compute a left arrow ($\nwarrow$) on $x\tau$. Suppose that this happens. Then, by the induction hypothesis, $x\tau$ is a proper suffix of some string $<yx\tau$ in L_0; in particular, $\tau = >$. It is claimed that in this case the string w is one of the first three forms listed in the statement of this lemma, and therefore the state computed on w is not a blank state ($-$). The proof splits into three cases depending on the symbol σ.

- If $\sigma = c$, then w is a proper suffix of a string $<ycx> \in L_0$, contradicting the assumption.
- If $\sigma = >$, then, similarly, w is a proper suffix of $<y<>x> \in L_0$, and this is a contradiction.
- Let $\sigma = <$, and assume first that x is an element of L_0. Then, $w = <x>$ belongs to L_0 as well, contradiction.
  Otherwise, if x is not in L_0, while $x>$ is a suffix of a string in L_0, then x must contain more right brackets than left brackets. Consider the leftmost unmatched right bracket ($>$) in x, and let $x = x'>x''$, where $x' \in L_0^+$ is a substring preceding that bracket. Then, the string $<y><x'>x''>$ belongs to L_0, and $w = <x'>x''>$ is therefore a proper suffix of a string in L_0, which is a contradiction.

The second claim, that a right arrow ($\nearrow$) cannot be computed on σx, is proved symmetrically.

The last, third claim is that the automaton does not compute a box ($\square$) on σx. Suppose that it does. Then, by the induction hypothesis, $\sigma x = uv$, where $u \in L_0$ and v is a prefix of a string in L_0^+. Let $v\widetilde{v}$ be that string in L_0^+, and consider the following possibilities for τ.

- If $\tau = c$, then w is a prefix of a string $uvc\widetilde{v} \in L_0^+$.
- If $\tau = <$, then w is prefix of $uv<>\widetilde{v} \in L_0^+$.
- For $\tau = >$, first, assume that $v \in L_0^+$. Then, $w = <uv>$ is in L_0.
  If v is not in L_0^+, then it contains unmatched left brackets. Let $v = v'<v''$, where v'' is the substring following the rightmost unmatched left bracket. Then, w is a prefix of a string $<uv'<v''><\widetilde{v}$ from L_0^+.

In each case, the string w belongs to one of the first three classes of strings in the statement of the lemma, and hence the state computed on it is not the blank state ($-$). This is a contradiction.

With the above cases eliminated, there remain the following states that may potentially be computed on σx and on $x\tau$.

$$\Delta(\mathcal{I}(\sigma x)) \in \{\nwarrow, -\} \qquad \Delta(\mathcal{I}(x\tau)) \in \{\nearrow, \square, -\}$$

For each of the six resulting pairs, there is a corresponding transition (2a), (2b), (2c), (2d), (2e), (2i) that computes the blank state ($-$) on w.

Note that a transition $\Delta(\nearrow, -)$ is never needed in this proof, and therefore the automaton shall never use it on any input. Hence, it can be defined arbitrarily. □

5. Simulation of an input-driven automaton

The method for constructing trellis automata presented in its basic form in the above Example 2 shall now be used to establish the following result.

Theorem 1. *Every language recognized by an input-driven automaton is linear conjunctive.*

Given an arbitrary IDPDA $(\Sigma_{+1}, \Sigma_{-1}, \Sigma_0, Q, q_0, \Gamma, (\delta_<)_{<\in\Sigma_{+1}}, (\gamma_<)_{<\in\Sigma_{+1}}, (\delta_>)_{>\in\Sigma_{-1}}, (\delta_c)_{c\in\Sigma_0}, F)$ recognizing some language $L \subseteq (\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0)^*$, the proof of Theorem 1 is by constructing a trellis automaton that recognizes the language $¢L\$$, where the cent sign ($¢$) and the dollar sign ($\$$) are two new symbols. Then, by Lemma B, the unmodified language L is also linear conjunctive.

The core of the desired trellis automaton is the automaton from Example 2, extended to manipulate information on the IDPDA's computations. Each of the new automaton's states is a variant of one of the four original states $\{\nearrow, \nwarrow, \square, -\}$, with some attached data representing input symbols and internal states of the IDPDA.

Even though the state of an IDPDA at each point of its computation is uniquely determined, it depends on all input symbols up to the current position. On the other hand, when a trellis automaton computes its state on a substring, it has no knowledge of this substring's left context. Because of that, the deterministic operation of an IDPDA shall be simulated as one would typically simulate nondeterministic computation: for each well-nested substring of the input, the constructed trellis automaton shall trace the computation of an IDPDA on that substring *beginning with all possible states.*

According to this plan, a box-state ($\square$) from the automaton in Example 2 now has to remember a function $f\colon Q \to Q$, which maps a state, on which the IDPDA begins its computation on the current substring, to the resulting state after reading that substring. Each right arrow ($\nearrow$) remembers the left bracket $< \in \Sigma_{+1}$ from which it has been spawned. Left arrows ($\nwarrow$) also remember their original right bracket $> \in \Sigma_{-1}$, and besides it they calculate a function $f\colon Q \to Q$ representing the behaviour of the IDPDA on the current substring. Blank states ($-$) carry no data. Overall, the constructed automaton has the following set of states.

$$\mathcal{S} = \{\nearrow_< \mid < \in \Sigma_{+1}\} \cup \{\nwarrow^f_> \mid > \in \Sigma_{-1},\ f\colon Q \to Q\} \cup \{\square_f \mid f\colon Q \to Q\} \cup \{-\}$$

In order to formulate the intended meaning of these states in a way similar to Lemma 1, the definition of the language L_0 from Example 2 shall be extended to the new setting. Let L_0 now be the language of all non-empty well-nested strings over $\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ that are not representable as concatenations of two non-empty well-nested strings. This language is described by the following grammar.

$$S \to <A> \qquad (< \in \Sigma_{+1},\ > \in \Sigma_{-1})$$
$$S \to c \qquad (c \in \Sigma_0)$$
$$A \to SA \mid \varepsilon$$

Now, the intention is to have the automaton reach the following states on different substrings.

Lemma 2. *(Cf. Lemma 1.) There exists a trellis automaton with the set of states $\mathcal{S}$ that carries out the following computations.*

I. *On each proper non-empty prefix of a string in L_0, which is of the form $<u$, the automaton computes a right arrow ($\nearrow_<$) marked with the first symbol of this prefix.*
II. *For each proper non-empty suffix of a string in L_0, let it be represented as $xu_1 \dots u_k>$, where $k \geqslant 0$, $u_i \in L_0$ and x is a proper suffix of a string in L_0. Then, on this string, the automaton computes a left arrow ($\nwarrow^f_>$) marked with the last symbol of this suffix and with the function f representing the behaviour on $u_1 \dots u_k$.*
III. *On each string of the form uv, where $u \in L_0$ and v is any prefix of a string in L_0^+. Then the automaton should compute a box ($\square_f$) marked with the behaviour function f on u.*
IV. *The blank state ($-$) is computed in all other cases.*

Proof. The transitions of the new automaton are defined so that its computations extend those in Example 2, in the sense that if the subscripts of all states are ignored, then one would obtain valid transitions and valid computations of the original four-state automaton. Whereas the automaton in Example 2, executed on a string $<w_1 \dots w_k>$, only traces paths from well-nested substrings $w_1, \dots, w_k$ to the right diagonal and then to the pinnacle, the new automaton calculates the IDPDA's behaviour on each substring, transfers these functions to the right diagonal and calculates their composition there. This process is illustrated in Fig. 4, and it remains to define the transitions to carry out such computations.

On a left bracket ($<$), the initial function creates a right arrow that remembers that bracket.

$$\mathcal{I}(<) = \nearrow_<, \qquad \text{for } < \in \Sigma_{+1}$$

On a right bracket ($>$), the initial function stores the bracket and begins constructing the behaviour of the IDPDA with the identity function $id\colon Q \to Q$, which is its behaviour on the empty string.

$$\mathcal{I}(>) = \nwarrow^{id}_>, \qquad \text{for } > \in \Sigma_{-1}$$

On a neutral symbol (c), the initial function makes a box-state that stores the IDPDA's transition function by that symbol, $\delta_c\colon Q \to Q$.

$$\mathcal{I}(c) = \square_{\delta_c}, \qquad \text{for } c \in \Sigma_0$$

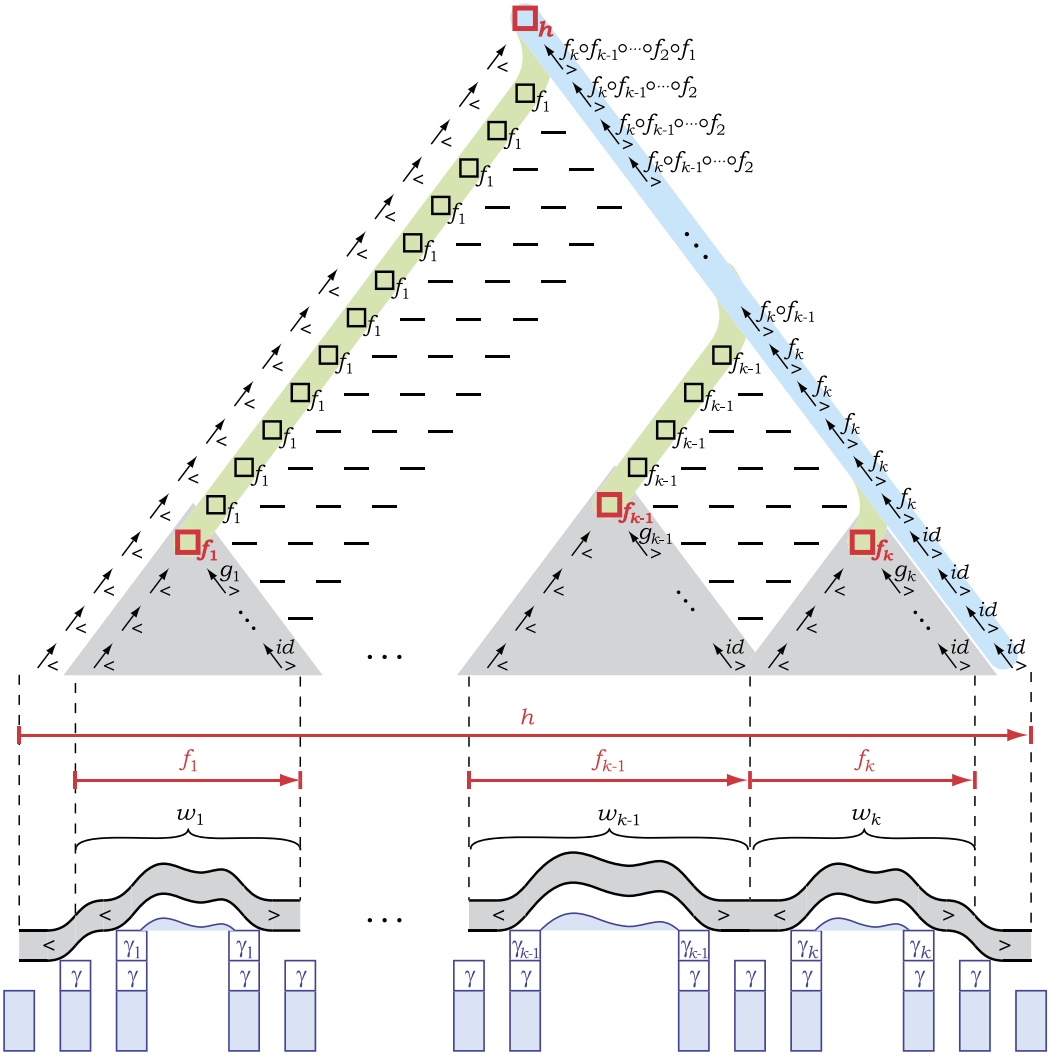

Fig. 4. (Bottom) Behaviour of an IDPDA on a well-nested string $<w_1 \dots w_{k-1} w_k>$; (top) how a simulating TA computes this behaviour inductively on the structure of brackets.

The transitions belong to fifteen groups, each corresponding to one of the fifteen transitions in Example 2, and this time manipulating the data attached to the states. For every two subsequent substrings w_i, w_{i+1} with balanced parentheses, if both w_i and w_{i+1} are enclosed in brackets, then the transitions filling the zone between them propagate the behaviour on w_i and ignore the rest of the data.

$$\delta(\nwarrow^{f}_{>}, \nearrow_{\ll}) = -, \qquad \text{for } f\colon Q \to Q,\ > \in \Sigma_{-1},\ \ll \in \Sigma_{+1} \tag{3a}$$

$$\delta(\nwarrow^{f}_{>}, -) = -, \qquad \text{for } f\colon Q \to Q,\ > \in \Sigma_{-1} \tag{3b}$$

$$\delta(-, \nearrow_{<}) = -, \qquad \text{for } < \in \Sigma_{+1} \tag{3c}$$

$$\delta(-, -) = - \tag{3d}$$

$$\delta(-, \square_g) = -, \qquad \text{for } g\colon Q \to Q \tag{3e}$$

$$\delta(\square_f, -) = \square_f, \qquad \text{for } f\colon Q \to Q \tag{3f}$$

If both w_i and w_{i+1} are individual neutral symbols, then the behaviour on w_i is propagated right–upwards by another type of transitions.

$$\delta(\square_f, \square_g) = \square_f, \qquad \text{for } f, g\colon Q \to Q \tag{3g}$$

If one of w_i and w_{i+1} is a neutral symbol and the other is enclosed in brackets, then some further transitions are needed to propagate the behaviour on w_i.

$$\delta(\square_f, \nearrow_{\ll}) = \square_f, \qquad \text{for } f\colon Q \to Q,\ \ll \in \Sigma_{+1} \tag{3h}$$

$$\delta(\overset{f}{\nwarrow}_{>}, \square_g) = -, \qquad \text{for } f, g\colon Q \to Q,\ > \in \Sigma_{-1} \tag{3i}$$

Turning to the transitions for maintaining the right border, they are now in charge of receiving the behaviour functions on all substrings $w_1, \ldots, w_k$, and of calculating their composition, thus producing the behaviour on their concatenation.

$$\Delta(\square_f, \overset{g}{\nwarrow}_{>}) = \overset{g\circ f}{\nwarrow}_{>}, \qquad \text{for } f, g\colon Q \to Q,\ > \in \Sigma_{-1} \tag{3j}$$

In all other cases on the right border, the left arrow-states maintain their data.

$$\Delta(-, \overset{g}{\nwarrow}_{>}) = \overset{g}{\nwarrow}_{>}, \qquad \text{for } g\colon Q \to Q,\ > \in \Sigma_{-1} \tag{3k}$$

$$\Delta(\overset{f}{\nwarrow}_{\gg}, \overset{g}{\nwarrow}_{>}) = \overset{g}{\nwarrow}_{>}, \qquad \text{for } f, g\colon Q \to Q,\ \gg, > \in \Sigma_{-1} \tag{3l}$$

On the left border, all transitions simply copy the left bracket ($<$).

$$\Delta(\nearrow_{<}, \square_f) = \nearrow_{<}, \qquad \text{for } f\colon Q \to Q,\ < \in \Sigma_{+1} \tag{3m}$$

$$\Delta(\nearrow_{<}, \nearrow_{\ll}) = \nearrow_{<}, \qquad \text{for } <, \ll \in \Sigma_{+1} \tag{3n}$$

Finally, when the right arrows ($\nearrow_{<}$) coming from the left bracket meet the left arrows ($\overset{f}{\nwarrow}_{>}$) coming from the right bracket to create a box-state, the trellis automaton has full information on how the IDPDA behaves on the string $w = <w_1 \ldots w_k>$. Namely, it knows the left bracket ($<$), the behaviour function f on $w_1 \ldots w_k$ and the right bracket ($>$). From these, it can determine the behaviour function h on w. Indeed, if the IDPDA begins reading w in a state $q \in Q$, then it first reads the left bracket ($<$), pushes $\gamma_<(q)$ onto the stack and enters the state $\delta_<(q)$, then processes $w_1 \ldots w_k$, finishing in the state $f(\delta_<(q))$, and finally pops the symbol $\gamma_<(q)$ from the stack and uses it in its transition by the right bracket ($>$). Accordingly, define $h(q) = \delta_>\big(f(\delta_<(q)), \gamma_<(q)\big)$. The transition of the trellis automaton creates a box-state labelled with this function h.

$$\Delta(\nearrow_{<}, \overset{f}{\nwarrow}_{>}) = \square_h, \qquad \text{for } f\colon Q \to Q,\ < \in \Sigma_{+1},\ > \in \Sigma_{-1} \tag{3o}$$

These transitions are put together in the following table, that extends the simpler transition table given in Example 2.

	$\nearrow_{\ll}$	$\overset{g}{\nwarrow}_{>}$	$\square_g$	$-$
$\nearrow_{<}$	$\nearrow_{<}$	$\square_h$	$\nearrow_{<}$	
$\overset{f}{\nwarrow}_{\gg}$	$-$	$\overset{g}{\nwarrow}_{>}$	$-$	$-$
$\square_f$	$\square_f$	$\overset{g\circ f}{\nwarrow}_{>}$	$\square_f$	$\square_f$
$-$	$-$	$\overset{g}{\nwarrow}_{>}$	$-$	$-$

A formal proof that all states reached by the automaton are as stated in Lemma 2 is carried out by induction on the length of substrings, with exactly the same partition into cases as in the proof of Lemma 1. At every step, one has to check that the information on the IDPDA's computation is calculated and transferred correctly.

To illustrate the similarity of these arguments better, the proof reuses the exact words of the original argument.

Base case I: $w = c$, with $c \in \Sigma_0$. The initial function produces the desired box-state $\mathcal{I}(c) = \square_{\delta_c}$ labelled with the behaviour on c.

Base case II: $w = <$, with $< \in \Sigma_{+1}$. This string is a non-empty proper prefix of the string $w' = <>$ from L_0, for some right bracket $> \in \Sigma_{-1}$, and accordingly, the initial function gives the right arrow labelled with the bracket $\mathcal{I}(<) = \nearrow_{<}$.

Base case III: $w = >$, with $> \in \Sigma_{-1}$. This is a non-empty proper suffix of the same string $<>$, and the initial function produces the left arrow labelled with the behaviour on the empty string: $\mathcal{I}(>) = \overset{id}{\nwarrow}_{>}$.

Induction step (prefix). Let w be a proper prefix of a string $<w_1 \ldots w_k>$, where $< \in \Sigma_{+1}$, $> \in \Sigma_{-1}$, $k \geqslant 0$ and $w_1, \ldots, w_k \in L_0$. Assume that w contains at least two symbols, and let $w = <x\tau$, where $\tau \in \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ is the last symbol of w and $x \in (\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0)^*$ is the middle part. The shorter prefix $<x$ is also a proper prefix of a string in L_0, and therefore, by the induction hypothesis, on the string $<x$, the automaton computes a right arrow state ($\nearrow_{<}$) labelled with the correct left bracket ($<$).

In order to determine the state computed on $x\tau$, first, consider the case when $x\tau$ contains the entire substring w_1 and zero or more symbols from $w_2 \ldots w_k$. Then, $x\tau$ is an element of L_0 followed by a prefix of a string in L_0^+, and, by the induction hypothesis, the automaton enters a box-state ($\square_{f_1}$) on $x\tau$, marked with a function f_1 representing the IDPDA's behaviour on w_1. The state on w is then computed by one of the automaton's transitions (3m), as follows.

$$\Delta(w) = \Delta(\Delta(<x), \Delta(x\tau)) = \Delta(\nearrow_{<}, \square_{f_1}) = \nearrow_{<}$$

The remaining possibility is that $x\tau$ is a proper prefix of w_1. The induction hypothesis then asserts that the state computed by the automaton on $x\tau$ is a right arrow ($\nearrow_{\ll}$) labelled with the first symbol of x. Then, a right arrow ($\nearrow_{<}$) labelled with the outer left bracket ($<$) is computed on w by the corresponding transition (3n).

Induction step (suffix). The proof is similar to the case of a prefix. This time, w is a proper suffix of $<w_1 \dots w_k>$. Let $w = \sigma x>$, with $\sigma \in \Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$ and $x \in (\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0)^*$. It is claimed that on w, the automaton computes a left arrow state ($\nwarrow^{f}_{>}$) labelled with the right bracket and with some behaviour function. The form of the behaviour function and the exact transition used to compute this state depend on where σx begins in relation to the boundaries between the substrings $w_1, \dots, w_k$.

First, if σx begins exactly at the boundary between w_{i-1} and w_i, for some $i \in \{1, \dots, k\}$, that is, if $\sigma x = w_i \dots w_k$, this means that it is a concatenation of a string in L_0 and a prefix of a string in L_0^+. Then, $\Delta(\sigma x) = \square_f$ by the induction hypothesis, where f is the behaviour on w_i. Next, since $x>$ is a proper suffix of a string in L_0, by the induction hypothesis, the state computed on $x>$ is a left arrow ($\nwarrow^{g}_{>}$) labelled with the behaviour g on $w_{i+1} \dots w_k$. Then the transition (3j) computes a left arrow ($\nwarrow^{g \circ f}_{>}$) on w, which is labelled with the behaviour on $w_i \dots w_k$.

Secondly, if σx is a proper suffix of w_k, then $\Delta(\sigma x) = \nwarrow^{f}_{\gg}$, where f is the behaviour function on some suffix of σx, and $\Delta(x>) = \nwarrow^{id}_{>}$. A left arrow state ($\nwarrow^{id}_{>}$) labelled with the identity function is then computed on w by the transition (3l).

The third case is when σx breaks one of the substrings w_i, with $i \in \{1, \dots, k-1\}$, into two non-empty parts, that is, $\sigma x = y w_{i+1} \dots w_k$, where y is a non-empty proper suffix of w_i. As in the proof of Lemma 1, this string σx is not of any of the first three forms listed in the statement of the lemma. Therefore, by the induction hypothesis, $\Delta(\sigma x) = -$. The state computed on the string $x>$ is a left arrow ($\nwarrow^{g}_{>}$) labelled with the behaviour g on $w_{i+1} \dots w_k$. Then the transition (3k) produces another left arrow state ($\nwarrow^{g}_{>}$) on w, marked with the same behaviour function g.

Induction step (string from L_0). Let $w = <w_1 \dots w_k>$ be a string belonging to L_0, with $< \in \Sigma_{+1}$, $> \in \Sigma_{-1}$, $k \geqslant 0$ and $w_1, \dots, w_k \in L_0$. This makes its prefix $<w_1 \dots w_k$ a proper non-empty prefix of a string in L_0, and then, by the induction hypothesis, the automaton computes a right arrow ($\nearrow_{<}$) labelled with the left bracket on this string. On the suffix $w_1 \dots w_k>$, similarly, the automaton produces a left arrow ($\nwarrow^{f}_{>}$) labelled with the right bracket and with the behaviour on $w_1 \dots w_k$. Then, on w, the transition function (3o) computes a box ($\square_h$) labelled with a function h defined on each state $q \in Q$ as $h(q) = \delta_{>}\big(f(\delta_{<}(q)), \gamma_{<}(q)\big)$. This is the desired behaviour function on w.

Induction step (L_0 followed by a prefix of L_0^+). Let $w = uv$, where $u \in L_0$, and v is a non-empty prefix of a string from L_0^+. Let τ be the last symbol of v, so that $v = v'\tau$. Since v' is a also prefix of a string from L_0^+, by the induction hypothesis, the automaton computes a box ($\square_f$) on uv', labelled with the behaviour f on u. The goal is to show that a box state ($\square_f$) is also computed on w, and thus this behaviour function is propagated further.

Let σ be the first symbol of u, and let $u = \sigma u'$. The question is to determine the state computed on $u'v$. This state depends on whether u is a single symbol (with two subcases depending on v) or a substring enclosed in brackets.

- Assume that $u = c$, with $c \in \Sigma_0$, so that $\sigma = c$, $u' = \varepsilon$ and $w = cv$, and further assume that v is a proper prefix of a string in L_0. Then, by the induction hypothesis, $\Delta(\mathcal{I}(v)) = \nearrow_{<}$, where the left bracket ($<$) is the first symbol of v, and then the corresponding transition (3h) produces the required box state ($\square_f$) on w.
- If $u = c$, but v is *not* a proper prefix of a string in L_0, then it must be a concatenation of a full string from L_0 with a prefix of a string from L_0^+. In this case, $\Delta(\mathcal{I}(v)) = \square_g$, where g is the behaviour on the prefix of v belonging to L_0. The box state ($\square_f$) is then obtained for w by the transition (3g).
- If $u = <u_1 \dots u_k>$, with $< \in \Sigma_{+1}$, $> \in \Sigma_{-1}$, $k \geqslant 0$ and $u_1, \dots, u_k \in L_0$, then $\sigma = <$ and $u' = u_1 \dots u_k>$, and the substring in question is $u_1 \dots u_k>v$, where v is non-empty. This substring is neither a prefix of any string in L_0^+, nor a suffix of any string in L_0. Therefore, by the induction hypothesis, the automaton computes a blank state ($-$) on it. Then the transition (3f) produces the box ($\square_f$).

Induction step (strings of any other form). For a string w that is neither a prefix of any string in L_0^+, nor a suffix of any string in L_0, the goal is to show that the automaton computes a blank state ($-$) on w. Let $w = \sigma x \tau$, where σ and τ may be any symbols from the alphabet $\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0$.

The proof of this case proceeds by showing which states *cannot* be computed on the substrings σx and $x\tau$. Namely, on $x\tau$, the automaton cannot compute a left arrow ($\nwarrow^{f}_{>}$), for any right bracket $> \in \Sigma_{-1}$ and $f\colon Q \to Q$, whereas on σx, it can neither compute a right arrow ($\nearrow_{<}$), for any left bracket $< \in \Sigma_{+1}$, nor a box ($\square f$), for any behaviour function f. The arguments are exactly the same as in the proof of Lemma 1, showing that if any of these possibilities takes place, then the state computed on w cannot be the blank state ($-$).

With the above cases eliminated, there remain the following states that may potentially be computed on σx and on $x\tau$.

$$\Delta(\mathcal{I}(\sigma x)) \in \{\nwarrow^{g}_{>} \mid > \in \Sigma_{-1},\ g\colon Q \to Q\} \cup \{-\}$$

$$\Delta(\mathcal{I}(x\tau)) \in \{\nearrow_< \mid < \in \Sigma_{+1}\} \cup \{\square_g \mid g\colon Q \to Q\} \cup \{-\}$$

For each of the six resulting pairs, there is a corresponding transition (3a), (3b), (3c), (3d), (3e), (3i) that computes the blank state $(-)$ on w. □

The second step of the construction is to extend the trellis automaton in Lemma 2 to accept all strings of the form $¢w\$$, where w is a string accepted by the given IDPDA. On an input string of this form, the automaton defined in Lemma 2 computes certain data on the substring w, as described in the lemma. These data show up on both sides of the triangle of states, as the states computed on various prefixes and suffixes of w. The extended automaton shall process these data using new states spawned from the left end-marker ($¢$) and from the right end-marker ($). On the left side of the triangle, the extended trellis automaton uses *right-angle arrow states* ($\overset{\curvearrowright}{q}$), defined for all states $q \in Q$. On the right side of the triangle, there are *left-angle arrow states* ($\overset{\curvearrowleft}{f}$), defined for each behaviour function $f\colon Q \to Q$. When the two angled arrows meet at the pinnacle, if the acceptance conditions are satisfied, then the automaton enters an accepting state $(*)$. Finally, there is a *dead state* (O), in which all remaining strings are rejected. Altogether, the set of states $\mathcal{S}$ of the automaton defined in Lemma 2 is augmented with the following new states.

$$\mathcal{S}' = \mathcal{S} \cup \{\overset{\curvearrowright}{q} \mid q \in Q\} \cup \{\overset{\curvearrowleft}{f} \mid f\colon Q \to Q\} \cup \{*, O\}$$

Before describing the intended meaning of these states, it is important to recall that, according to Definition 3, the string w need not be well-nested. In general, an arbitrary string $w \in (\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0)^*$ may first have some unmatched right brackets $(>)$, and then some unmatched left brackets $(<)$. Accordingly, w may be split into two parts, and the plan is to process the first part with unmatched right brackets $(>)$ in right-angled arrow states ($\overset{\curvearrowright}{q}$) on the left side of the triangle, whereas the second part is to be handled by left-angled arrows ($\overset{\curvearrowleft}{f}$) on the right side, as illustrated in Fig. 5.

This requires some *ad hoc* terminology. Let a string be called *descending*, if it is either empty, or a concatenation of one or more well-nested substrings and unmatched right brackets $(>)$ that ends with an unmatched right bracket. A concatenation of zero or more well-nested substrings and unmatched left brackets $(<)$ is called *ascending*. Then, an arbitrary string $w \in (\Sigma_{+1} \cup \Sigma_{-1} \cup \Sigma_0)^*$ can be uniquely represented as a concatenation $w = uv$ of a descending string u and an ascending string v.

The constructed trellis automaton simulates the computation of the IDPDA on u by calculating its state after each unmatched right bracket $(>)$ on the left side of the triangle. On the right side, as it is not known in which state the IDPDA begins reading v, the trellis automaton has to trace its computation beginning with all possible states. Fortunately, the notion of the behaviour of an IDPDA on a well-nested substring can be extended to ascending strings: indeed, the transitions by unmatched left brackets $(<)$ push symbols that shall never be popped, and therefore those push operations can be ignored, and unmatched left brackets $(<)$ can be handled as if they are neutral symbols.

Using this terminology, the states reached by the extended automaton can be easily described.

Lemma 3. *There exists a trellis automaton with the set of states $\mathcal{S}'$ that operates as follows.*

I. *On each string $¢ux$, where u is a descending string and x is an ascending string, the automaton computes a right-angle arrow ($\overset{\curvearrowright}{q}$) labelled with the state reached by the IDPDA upon reading u from its initial configuration.*
II. *On each string $yv\$$, where y is descending and v is ascending, the automaton computes a left-angle arrow ($\overset{\curvearrowleft}{f}$) labelled with the behaviour of the IDPDA on v.*
III. *On each string $¢w\$$, the automaton reaches an accepting state $(*)$ if and only if the IDPDA accepts w.*

Sketch of a proof. All transitions in the states from $\mathcal{S}$ are defined as in Lemma 2.

Consider the new transitions emerging from the left end-marker ($¢$). The initial function creates a right-angled arrow ($\overset{\curvearrowright}{q_0}$) marked with the initial state of the IDPDA.

$$\mathcal{I}(¢) = \overset{\curvearrowright}{q_0}$$

All subsequent transitions with right-angled arrows ($\overset{\curvearrowright}{q}$) as the first argument ignore everything except the signals coming from unmatched right brackets.

$$\Delta(\overset{\curvearrowright}{q}, \nearrow_<) = \overset{\curvearrowright}{q}, \qquad \text{for } q \in Q,\ < \in \Sigma_{+1} \tag{4a}$$

$$\Delta(\overset{\curvearrowright}{q}, \square_f) = \overset{\curvearrowright}{q}, \qquad \text{for } q \in Q,\ f\colon Q \to Q \tag{4b}$$

$$\Delta(\overset{\curvearrowright}{q}, -) = \overset{\curvearrowright}{q}, \qquad \text{for } q \in Q \tag{4c}$$

In particular, the behaviour function on the first well-nested substring, which comes in a box-state ($\square_f$), is also ignored. This and all other behaviour functions shall actually be merged into the signals from unmatched right brackets, as described in Lemma 2(part II). When such a signal reaches the left side of the triangle as a left arrow state ($\overset{f}{\nwarrow_>}$), it carries the behaviour f on the latest well-nested substring. Then, the trellis automaton simulates the computation of the IDPDA on that well-nested substring followed by the unmatched left bracket, in a single transition.

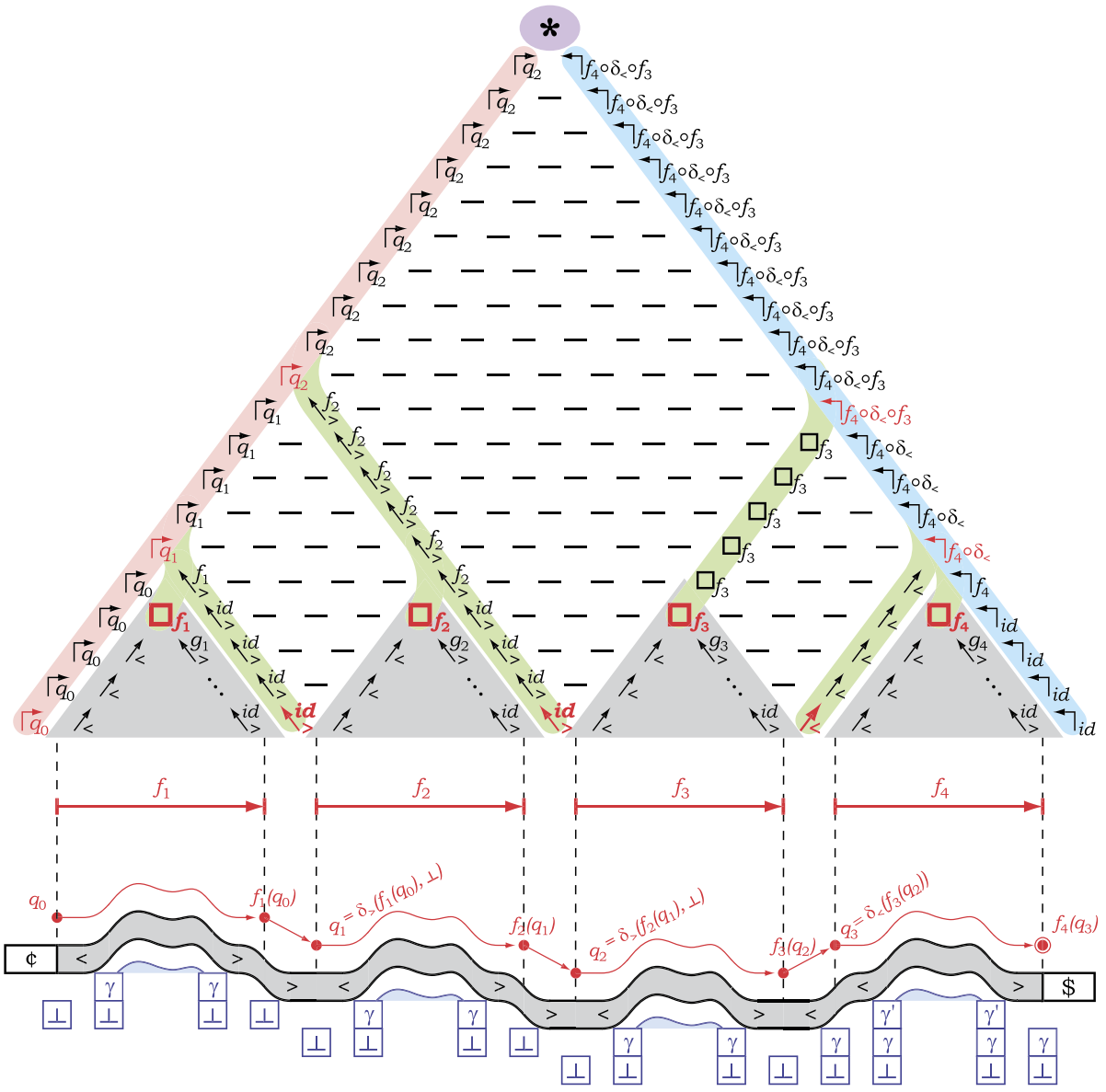

Fig. 5. (Bottom) Computation of an IDPDA on an ill-nested string $w_1{>}w_2{>}w_3{<}w_4$, where w_1, w_2, w_3, w_4 are its well-nested substrings; (top) how a TA simulates this computation on an input $¢w\$$.

$$\Delta(\uparrow_q, \nwarrow^{f}_{>}) = \uparrow_{\delta_>(f(q),\perp)}, \qquad \text{for } q \in Q,\ f\colon Q \to Q,\ > \in \Sigma_{-1} \tag{4d}$$

These are all transitions acting on the left side of the triangle. They are illustrated in Fig. 5, along with the simulated computation of the IDPDA. A formal proof of the first statement of this lemma can be carried out according to above explanation, using induction on the length of the prefix, and referring to Lemma 2(part II) to determine incoming signals.

Computations carried out on the right side of the triangle begin with the left end-marker ($), on which the automaton creates a left-angled arrow ($\leftarrow_{id}$) marked with the identity function $id\colon Q \to Q$, which describes the behaviour of the IDPDA on the empty suffix.

$$\mathcal{I}(\$) = \leftarrow_{id}$$

The triangle's right side receives signals of two types: boxes ($\square_f$) containing behaviour functions on well-nested substrings, and right arrows ($\nearrow_<$) corresponding to unmatched left brackets ($<$). In each case, their behaviour functions are composed with the behaviour carried by the left-angled arrow ($\leftarrow_g$).

$$\Delta(\square_f, \leftarrow_g) = \leftarrow_{g\circ f}, \qquad \text{for } f, g\colon Q \to Q \tag{4e}$$

$$\Delta(\nearrow_{<}, \lnot_g) = \lnot_{g \circ \delta_<}, \qquad \text{for } < \in \Sigma_{+1},\ g\colon Q \to Q \tag{4f}$$

All other states in the first argument are ignored.

$$\Delta(\nwarrow^{f}_{>}, \lnot_g) = \lnot_g, \qquad \text{for } > \in \Sigma_{-1},\ f, g\colon Q \to Q \tag{4g}$$

$$\Delta(-, \lnot_g) = \lnot_g, \qquad \text{for } g\colon Q \to Q \tag{4h}$$

By induction on the length of the suffix and with the help of Lemma 2(part I), one can prove that all states reached by these transitions satisfy part II of the present lemma.

At the pinnacle of the triangle, the two angled arrows meet. Consider the uniquely defined partition of the string into $w = uv$, where u is a descending prefix and v is an ascending suffix. Then, the topmost right-angled arrow ($\vdash_q$) carries the state in which the IDPDA finishes reading u; note that at this point, its stack is empty. At the same time, the left-angled arrow ($\lnot_g$) has the behaviour function g on v attached to it. It remains to evaluate g on q and see whether the resulting state is accepting.

$$\Delta(\vdash_q, \lnot_g) = *, \qquad \text{if } g(q) \in F \tag{4i}$$

The correctness of this step follows from parts I and II of this lemma.

All other transitions are defined to lead to the dead state (O). □

Once a trellis automaton recognizing the language ¢L\$ is constructed, the desired trellis automaton for L can be obtained from it by applying Lemma B twice. This completes the proof of Theorem 1.

6. An LL(1) language that is not linear conjunctive

To show that the families of LL languages, LR languages and unambiguous languages are all incomparable with the linear conjunctive languages, it is sufficient to present an LL grammar describing a language that is not linear conjunctive. More precisely, the witness grammar belongs to the class of *LL(1) grammars in the Greibach normal form*, which has the following particularly simple definition.

Definition 4. (See Korenjak and Hopcroft [19].) Let $G = (\Sigma, N, R, S)$ be an ordinary grammar in the Greibach normal form, that is, with each rule in R of the form $A \to aX_1 \dots X_\ell$, for some $A \in N$, $a \in \Sigma$, $\ell \geqslant 0$ and $X_1, \dots, X_\ell \in \Sigma \cup N$. If, for each pair of A and a, at most one such rule may exist, the grammar is called LL(1).

Once the witness grammar is presented, the task is to prove that no linear conjunctive grammar generates the same language. A method for proving such statements was described by Terrier [34], and it is based upon the following idea. Suppose that a language L is recognized by a trellis automaton. Fix any number $k \geqslant 1$ and consider the automaton's computations on any strings that are at least k symbols long. The top k rows of the computation triangle consist of $\frac{k(k+1)}{2}$ states, and each of them determines whether some substring of the input is accepted; if the input string is w, then these states correspond to all substrings of w that are over $|w| - k$ symbols long. The membership status of these $\frac{k(k+1)}{2}$ substrings in L forms an *acceptance pattern* comprised of $\frac{k(k+1)}{2}$ bits. This acceptance pattern is a property of w, and over various strings w, there can be up to $\frac{k(k+1)}{2}$ distinct acceptance patterns.

At the same time, note that those $\frac{k(k+1)}{2}$ states in the k top rows are completely determined by the k states forming the k-th upper row of the triangle, as shown in Fig. 6. If p is the number of states in the trellis automaton, then it can discriminate between at most p^k acceptance patterns. The key point of Terrier's [34] argument is that some languages are too rich in acceptance patterns, so that this power of discrimination is not enough to test membership of strings in those languages.

To formalize this intuition, consider a special complexity function of a language that defines the number of distinct acceptance patterns required in the top k rows.

Definition 5. Let $L \subseteq \Sigma^*$ be a language, let $k \geqslant 1$, and let $w = a_1 \dots a_n$ be a string with $n \geqslant k$. Consider all strings obtained from w by removing up to $k-1$ symbols from its beginning and from its end: that is, all strings of the form $a_{i+1} \dots a_{n-j}$, with $i, j \geqslant 0$ and $i + j < k$. Every such string is identified by a pair (i, j); the set of all such pairs for substrings in L is the acceptance pattern of w with respect to L and k.

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

Next, define the set of all possible acceptance patterns $S_{L,k,w}$ over all strings w.

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

Define an integer function $f_L\colon \mathbb{N} \to \mathbb{N}$ representing the cardinality of this set by $f_L(k) = |\widehat{S}_{L,k}|$.

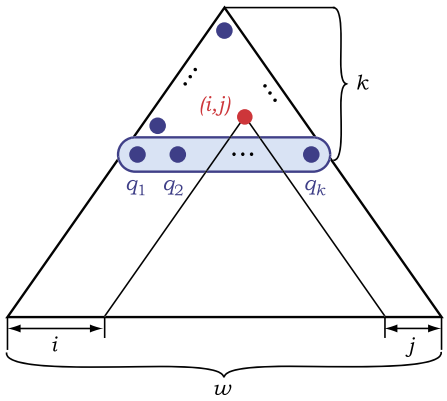

Fig. 6. Top k rows of a computation of a trellis automaton, and a pair (i, j), with $i + j < k$, as in Definition 5.

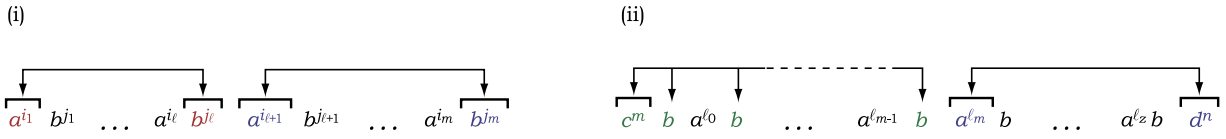

Fig. 7. The structure of strings in the languages (i) in Example 3, and (ii) in Lemma 4.

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 and $2^{\frac{k(k+1)}{2}}$. However, if the language L is recognized by a trellis automaton, then its complexity measure $f_L(k)$ is bounded by an exponential function.

Lemma C. *(See Terrier [34].) If $L \in \Sigma^*$ is a linear conjunctive language, then there is such a number p, that*

$$f_L(k) \leqslant p^k, \quad \text{for all } k \geqslant 1.$$

Lemma C was accompanied with an example of a language that maximizes this complexity measure, and hence is recognized by no trellis automaton.

Example 3. (See Terrier [34].) Consider the following language L_0, which can be described by a linear grammar.

$$L_0 = \{a^n w b^n \mid n \geqslant 1;\ w \in b\{a,b\}^* a \text{ or } w = \varepsilon\}.$$

Its concatenation with itself has the following form.

$$L = L_0 \cdot L_0 = \{a^{i_1} b^{j_1} \ldots a^{i_\ell} b^{j_\ell} a^{i_{\ell+1}} b^{j_{\ell+1}} \ldots a^{i_m} b^{j_m} \mid m \geqslant 2;\ i_t, j_t \geqslant 1,\ \exists \ell :\ i_1 = j_\ell \text{ and } i_{\ell+1} = j_m\}$$

This concatenation satisfies $f_L(k) = 2^{\frac{k(k+1)}{2}}$, and therefore, by Lemma C, is not linear conjunctive.

A string's membership condition in L is illustrated in Fig. 7(i). If the string begins with a^{i_1} and ends with b^{j_m}, then it is required that the pair $b^{i_1} a^{j_m}$ occurs somewhere inside the string; and if any symbols are erased from the beginning or from the end of a string, then an entirely different pair has to occur. Thus, the membership of the string in the language depends on its inner contents, and one can generate any acceptance pattern as in Fig. 6 by varying the inner contents.

The only imperfection of this fine example is that the language $L_0 \cdot L_0$ is *inherently ambiguous* [26, Prop. 8], that is, every ordinary grammar describing this language must be ambiguous. For that reason, it cannot be used to separate the linear conjunctive languages from the unambiguous languages and their subclasses. The desired separation is based on another example presented below. The language in the new example is LL(1), and even though it has fewer acceptance patterns than the language in Example 3, their number is still super-exponential.

Lemma 4. *The following language is described by an LL(1) grammar in Greibach normal form.*

$$L = \{c^m b a^{\ell_1} b \ldots a^{\ell_{m-1}} b a^{\ell_m} b \ldots a^{\ell_z} b d^n \mid m, n, z \geqslant 1,\ \ell_i \geqslant 0,\ \ell_m = n\}$$

On the other hand, $f_L(k) \geqslant (k+1)! = 2^{\Theta(k \log k)}$, and therefore L is not linear conjunctive.

Proof. The membership of a string in L can be tested as illustrated in Fig. 7(ii): the number of symbols c in the prefix points to the m-th substring a^{ℓ_m}, and then the number of symbols a in this substring should be the same as the number of symbols d in the suffix. As in Example 3, these two comparisons can be done independently of each other, so that the language L is representable as a concatenation of the following two languages.

$$L_1 = \{c^m b a^{\ell_1} b \dots a^{\ell_{m-1}} b \mid m \geqslant 1,\ \ell_i \geqslant 0\}$$

$$L_2 = \{a^n b w d^n \mid n \geqslant 1,\ w \in \{a,b\}^*\}$$

This representation leads to the following LL(1) grammar in Greibach normal form.

$$S \to cCaD$$

$$C \to cCA \mid b$$

$$A \to aA \mid b$$

$$D \to aDd \mid bB$$

$$B \to aB \mid bB \mid d$$

The main thing to prove about this language is that its complexity function $f_L(k)$ satisfies $f_L(k) \geqslant (k+1)!$, for each $k \geqslant 1$.

Consider any k-tuple of integers $(\ell_1, \dots, \ell_k)$, with each ℓ_m belonging to the range $k - m \leqslant \ell_m \leqslant k$. For every such k-tuple, the corresponding string $w_{\ell_1,\dots,\ell_k}$ is defined as follows.

$$w_{\ell_1,\dots,\ell_k} = c^k b a^{\ell_1} b \dots a^{\ell_k} b d^k$$

By definition, the acceptance pattern $S_{L,k,w_{\ell_1,\dots,\ell_k}}$ for this string contains all pairs (i, j), with $i, j \geqslant 0$ and $i + j < k$, that satisfy $c^{k-i} b a^{\ell_1} b \dots a^{\ell_k} b d^{k-j} \in L$. The latter condition is equivalent to $\ell_{k-i} = k - j$. Note that ℓ_{k-i} is between i and k by construction, whereas $k - j$ is between $i + 1$ and k. Thus, for each i, the acceptance pattern contains at most one of the pairs $(i, 0)$, $(i, 1)$, …, $(i, k - 1 - i)$, namely, the one with the second component $k - \ell_{k-i}$, as long that ℓ_{k-i} is at least $i + 1$. If ℓ_{k-i} is i, then none of those pairs are in $S_{L,k,w_{\ell_1,\dots,\ell_k}}$.

Accordingly, the sets $S_{L,k,w_{\ell_1,\dots,\ell_k}}$ corresponding to different k-tuples $(\ell_1, \dots, \ell_k)$ are pairwise distinct, and since there are $(k + 1)!$ such k-tuples, the complexity function for L satisfies $f_L(k) = |\{S_{L,k,w} \mid w \in \Sigma^*,\ |w| \geqslant k\}| \geqslant (k + 1)!$, as claimed. Then, by Lemma C, the language L is not linear conjunctive. □

Theorem 2. *The family of linear conjunctive languages is incomparable with the unambiguous languages, the deterministic languages and the LL(k) languages.*

Indeed, linear conjunctive grammars can describe the language $\{a^n b^n c^n \mid n \geqslant 0\}$, see Example 1, which cannot be defined by any ordinary grammar. On the other hand, the language in Lemma 4 is LL(1), but not linear conjunctive.

For future research, it is also interesting to observe that the language in Lemma 4 is recognized by a finite-turn deterministic one-counter automaton. Given an input string $c^m w d^n$, with $w \in \{a, b\}^*$, the automaton first reads the prefix c^m and stores its length in the counter. Then it decrements the counter as it reads the first $m + 1$ instances of b, ignoring any as encountered between them. Right after the $(m + 1)$-th b, the automaton reads the next block of as, storing its length in the counter, and then skips all remaining as and bs. Finally, as the symbols d are read, the counter is decremented, thus comparing their number to the number of as. This is a three-turn deterministic one-counter automaton.

7. The updated hierarchy

This paper has resolved the potential inclusions between several families of formal languages, which were marked by dotted lines in the earlier Fig. 1. The updated hierarchy is presented in Fig. 8. All inclusions in the figure are known to be proper, except those labelled with question marks: the language families defined by conjunctive and Boolean grammars, both unambiguous and of the general form (*UnambConj*, *UnambBool*, *Conj*, *Bool*), may theoretically coincide. Languages witnessing each separation are given in Table 1.

In addition, as presented in Table 2, most pairs of families not connected by a directed path are now known to be incomparable—with several exceptions listed in the second part of the table. It is conjectured that in each of the remaining cases, the families are also incomparable. If the conjecture is true, then it remains unproved due to the general lack of methods for proving that a given language is not described by any conjunctive or Boolean grammar [27]. Indeed, nothing is known about any limitations of these grammars, besides their complexity upper bound, which is $\mathrm{DTIME}(n^\omega) \cap \mathrm{DSPACE}(n)$, where $O(n^\omega)$ is the complexity of matrix multiplication, with $\omega < 2.4$ [28]. Finding out the limitations of conjunctive and Boolean grammars, and resolving the last question marks in Fig. 8, as well as the last omissions listed in Table 2, would significantly advance the knowledge on formal grammars.

Among other research problems suggested by the results of this paper, one can consider comparing several extensions of input-driven automata against various models related to linear conjunctive grammars, such as, for instance, those restricted

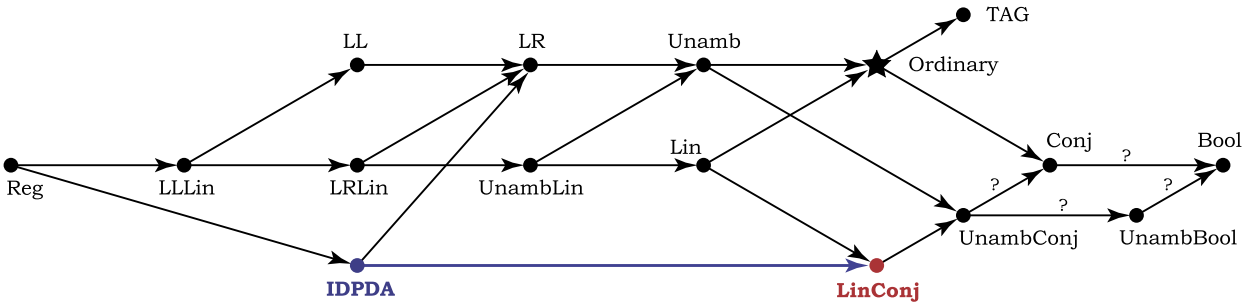

Fig. 8. The updated hierarchy of formal grammars (the names of the classes are explained in Fig. 1).

Table 1
Strictness of inclusions in Fig. 8.

Inclusion	Separating language
$Reg \subset LLLin$, $Reg \subset IDPDA$	$\{a^n b^n \mid n \geqslant 0\}$
$LL \subset LR$, $LLLin \subset LRLin$	$\{a^n c b^n \mid n \geqslant 0\} \cup \{a^n d b^{2n} \mid n \geqslant 0\}$
$LR \subset Unamb$, $LRLin \subset UnambLin$	$\{a^n b^n \mid n \geqslant 0\} \cup \{a^n b^{2n} \mid n \geqslant 0\}$
$IDPDA \subset LR$, $IDPDA \subset LinConj$	$\{a^n b^n \mid n \geqslant 0\} \cup \{b^n a^n \mid n \geqslant 0\}$
$Unamb \subset Ordinary$, $UnambLin \subset Lin$	$\{a^i b^m c^m \mid i, m \geqslant 0\} \cup \{a^n b^n c^j \mid n, j \geqslant 0\}$
$LLLin \subset LL$, $LRLin \subset LR$, $UnambLin \subset Unamb$, $Lin \subset Ordinary$	$\{a^m b^m c^n d^n \mid m, n \geqslant 0\}$
$Lin \subset LinConj$, $Unamb \subset UnambConj$, $Ordinary \subset Conj$, $Ordinary \subset TAG$	$\{a^n b^n c^n \mid n \geqslant 0\}$
$LinConj \subset UnambConj$	Lemma 4; also $\{a^{4^n} \mid n \geqslant 0\}$ [16]

Table 2
Incomparable language families in Fig. 8: (first part) incomparable; (second part) not known to be incomparable.

Pairs of families		Justifications of incomparability
$LinConj$,	LL, LR, $Unamb$	Theorem 2
$LinConj$,	$Ordinary$, TAG	By Example 3 [34] and $\{a^n b^n c^n d^n e^n \mid n \geqslant 0\} \in LinConj \setminus TAG$
$IDPDA$,	$LLLin$, $LRLin$, $UnambLin$, Lin	By $\{a^m b^m c^n d^n \mid m, n \geqslant 0\}$ and $\{a^n b^n \mid n \geqslant 0\} \cup \{b^n a^n \mid n \geqslant 0\}$ [2]
$IDPDA$,	LL	By $\{a^n b^n \mid n \geqslant 0\} \cup \{a^n c^n \mid n \geqslant 0\}$ [31] and $\{a^n b^n \mid n \geqslant 0\} \cup \{b^n a^n \mid n \geqslant 0\}$ [2]
LL, LR,	$UnambLin$, Lin	By $\{a^m b^m c^n d^n \mid m, n \geqslant 0\}$ and $\{a^n b^n \mid n \geqslant 0\} \cup \{a^n b^{2n} \mid n \geqslant 0\}$
LL,	$LRLin$	By $\{a^m b^m c^n d^n \mid m, n \geqslant 0\}$ and $\{a^n b^n \mid n \geqslant 0\} \cup \{a^n b^{2n} \mid n \geqslant 0\}$
$Unamb$,	Lin	By $\{a^m b^m c^n d^n \mid m, n \geqslant 0\}$ and $\{a^m b^m c^i \mid m, i \geqslant 0\} \cup \{a^j b^n c^n \mid j, n \geqslant 0\}$
$Ordinary$,	$UnambConj$, $UnambBool$	$\{a^n b^n c^n \mid n \geqslant 0\} \in UnambConj \setminus Ordinary$, but $Ordinary$ may be a subset of either class
TAG,	$UnambConj$, $Conj$, $UnambBool$, $Bool$	$\{a^n b^n c^n d^n e^n \mid n \geqslant 0\} \in UnambConj \setminus TAG$, but TAG may be a subset of any of these classes
$Conj$,	$UnambBool$	Nothing is known, these families may even coincide

by the LR(0) condition [1] or extended with context operators [3,4]. For example, it should not be difficult to see that Caucal's [6] *synchronized pushdown automata* can be simulated by *linear conjunctive grammars with left context operators* [4] through an accordingly modified construction from this paper. Likely, Caucal's automata could be simulated already by linear conjunctive grammars, but that would require elaborating the construction significantly. On the other hand, if linear conjunctive grammars are restricted by the LR(0) condition or by the LL(k) condition [27, Sect. 5.4], then it might be the case that they can no longer describe the Dyck language—but whether this is true or not, remains to be proved. Investigating these questions may lead to important refinements to the hierarchy of formal grammars.

References

[1] T. Aizikowitz, M. Kaminski, LR(0) conjunctive grammars and deterministic synchronized alternating pushdown automata, in: Computer Science in Russia, CSR 2011, St. Petersburg, Russia, 14–18 June 2011, in: LNCS, vol. 6651, 2011, pp. 345–358.
[2] R. Alur, P. Madhusudan, Visibly pushdown languages, in: ACM Symposium on Theory of Computing, STOC 2004, Chicago, USA, June 13–16, 2004, 2014, pp. 202–211.
[3] M. Barash, A. Okhotin, An extension of context-free grammars with one-sided context specifications, Inform. and Comput. 237 (2014) 268–293.
[4] M. Barash, A. Okhotin, Linear grammars with one-sided contexts and their automaton representation, RAIRO Theor. Inform. Appl. 49 (2) (2015) 153–178.
[5] B. von Braunmühl, R. Verbeek, Input-driven languages are recognized in log$n$ space, North-Holl. Math. Stud. 102 (1985) 1–19.
[6] D. Caucal, Synchronization of pushdown automata, in: Developments in Language Theory, DLT 2006, Santa Barbara, USA, 26–29 June 2006, in: LNCS, vol. 4036, 2006, pp. 120–132.
[7] K. Čulík II, J. Gruska, A. Salomaa, Systolic trellis automata I, Int. J. Comput. Math. 15 (1984) 195–212.
[8] K. Čulík II, J. Gruska, A. Salomaa, Systolic trellis automata II, Int. J. Comput. Math. 16 (1984) 3–22.
[9] C. Dyer, One-way bounded cellular automata, Informa. Control 44 (1980) 261–281.
[10] P.W. Dymond, Input-driven languages are in log$n$ depth, Inform. Process. Lett. 26 (1988) 247–250.
[11] S. Ginsburg, S.A. Greibach, Deterministic context-free languages, Informa. Control 9 (6) (1966) 620–648.
[12] S. Ginsburg, M. Harrison, Bracketed context-free languages, J. Comput. System Sci. 1 (1) (1967) 1–23.
[13] O.H. Ibarra, S.M. Kim, Characterizations and computational complexity of systolic trellis automata, Theoret. Comput. Sci. 29 (1984) 123–153.
[14] A. Jeż, Conjunctive grammars can generate non-regular unary languages, Internat. J. Found. Comput. Sci. 19 (3) (2008) 597–615.
[15] A. Jeż, A. Okhotin, Conjunctive grammars over a unary alphabet: undecidability and unbounded growth, Theory Comput. Syst. 46 (1) (2010) 27–58.
[16] A. Jeż, A. Okhotin, Unambiguous conjunctive grammars over a one-letter alphabet, in: Developments in Language Theory, DLT 2013, Paris, France, 18–21 June 2013, in: LNCS, vol. 7907, 2013, pp. 277–288.
[17] A.K. Joshi, L.S. Levy, M. Takahashi, Tree adjunct grammars, J. Comput. System Sci. 10 (1) (1975) 136–163.
[18] D.E. Knuth, On the translation of languages from left to right, Informa. Control 8 (6) (1965) 607–639.
[19] A.J. Korenjak, J.E. Hopcroft, Simple deterministic languages, in: 7th Annual Symposium on Switching and Automata Theory, SWAT 1966, Berkeley, California, USA, 23–25 October 1966, IEEE Computer Society, 1966, pp. 36–46.
[20] V. Kountouriotis, Ch. Nomikos, P. Rondogiannis, Well-founded semantics for Boolean grammars, Inform. and Comput. 207 (9) (2009) 945–967.
[21] K. Mehlhorn, Pebbling mountain ranges and its application to DCFL-recognition, in: Automata, Languages and Programming, ICALP 1980, Noordweijkerhout, the Netherlands, 14–18 July 1980, in: LNCS, vol. 85, 1980, pp. 422–435.
[22] A. Okhotin, Conjunctive grammars, J. Autom. Lang. Comb. 6 (4) (2001) 519–535.
[23] A. Okhotin, On the equivalence of linear conjunctive grammars to trellis automata, RAIRO Theor. Inform. Appl. 38 (1) (2004) 69–88.
[24] A. Okhotin, On the number of nonterminals in linear conjunctive grammars, Theoret. Comput. Sci. 320 (2–3) (2004) 419–448.
[25] A. Okhotin, Boolean grammars, Inform. and Comput. 194 (1) (2004) 19–48.
[26] A. Okhotin, Unambiguous Boolean grammars, Inform. and Comput. 206 (2008) 1234–1247.
[27] A. Okhotin, Conjunctive and Boolean grammars: the true general case of the context-free grammars, Comput. Sci. Rev. 9 (2013) 27–59.
[28] A. Okhotin, Parsing by matrix multiplication generalized to Boolean grammars, Theoret. Comput. Sci. 516 (2014) 101–120.
[29] A. Okhotin, K. Salomaa, Complexity of input-driven pushdown automata, SIGACT News 45 (2) (2014) 47–67.
[30] F.C.N. Pereira, D.H.D. Warren, Parsing as deduction, in: 21st Annual Meeting of the Association for Computational Linguistics, ACL 1983, Cambridge, Massachusetts, USA, 15–17 June 1983, 1983, pp. 137–144.
[31] D.J. Rosenkrantz, R.E. Stearns, Properties of deterministic top–down grammars, Informa. Control 17 (1970) 226–256.
[32] W.C. Rounds, LFP: a logic for linguistic descriptions and an analysis of its complexity, Comput. Linguist. 14 (4) (1988) 1–9.
[33] W. Rytter, An application of Mehlhorn's algorithm for bracket languages to log(n) space recognition of input-driven languages, Inform. Process. Lett. 23 (2) (1986) 81–84.
[34] V. Terrier, On real-time one-way cellular array, Theoret. Comput. Sci. 141 (1995) 331–335.