Pattern Anal Applic (2014) 17:465–480
DOI 10.1007/s10044-013-0322-1

ORIGINAL ARTICLE

Fundamental methodological issues of syntactic pattern recognition

Mariusz Flasiński · Janusz Jurek

Received: 22 February 2012 / Accepted: 30 January 2013 / Published online: 9 March 2013

Abstract Fundamental open problems, which are frontiers of syntactic pattern recognition are discussed in the paper. Methodological considerations on crucial issues in areas of string and graph grammar-based syntactic methods are made. As a result, recommendations concerning an enhancement of context-free grammars as well as constructing parsable and inducible classes of graph grammars are formulated.

Keyword Syntactic pattern recognition · Formal language · Graph grammar

1 Introduction

Representing a pattern as a structure of the form of string, tree or graph and a set of structures as a formal language is the main idea of syntactic pattern recognition [6, 24, 27, 42, 55], which is one of the main approaches in the area of machine recognition. A generation of such a language is made with a formal grammar. An analysis and a recognition of an unknown structure is performed with a formal automaton. If patterns are complex, they are defined in a hierarchical way. Thus, at the bottom of the hierarchy we use elementary patterns in order to build simple substructures (These elementary patterns are called *primitives* and they are represented with symbols of a language alphabet.). Then, using such simple substructures we construct more complex substructures and so on.

M. Flasiński (✉) · J. Jurek
IT Systems Department, Jagiellonian University,
ul. prof. St. Lojasiewicza 4, Cracow 30-348, Poland
e-mail: mariusz.flasinski@uj.edu.pl

Syntactic pattern recognition prevails over “standard” pattern recognition approaches (probabilistic, discriminant function-based, NN, etc.) when patterns considered can be characterized better with structural features than vectors of features. What is more, using this approach not only can we make a classification (in a sense of ascribing a pattern to a pre-defined category), but also a (structural) interpretation of an unknown pattern. Therefore, for structurally-oriented recognition problems such as: character recognition, speech recognition, scene analysis, chemical and biological structures analysis, texture analysis, fingerprint recognition, geophysics, a syntactic approach has been applied successfully since its beginning in the early 1960s for the next two decades. A rapid development of syntactic methods has slowed down since 1990s and the experts in this area (see e.g. [26]) have found this approach stagnating.

Methodological considerations on the issues which have an impact on further development of syntactic methods are made in the paper. Firstly, however, key open problems constituting the frontiers of this research area should be identified. It can be easily noticed in the literature concerning syntactic pattern recognition [6, 24, 27, 42, 55] that in the field of string-based models a lot of efficient methods have been developed for structural patterns that can be generated with regular or context-free grammars. On the other hand, if a set of patterns cannot be represented with context-free languages, i.e. it is of a context-sensitive nature, then defining an efficient recognition method is difficult. It results from a non-polynomial time complexity of automata analyzing context-sensitive languages. Therefore, defining string grammars generating languages with a polynomial membership problem that are stronger than context-free grammars seems to be still the key open problem in this area.

If a pattern is structurally complex, a linear-like string description is very often too weak for its representation.

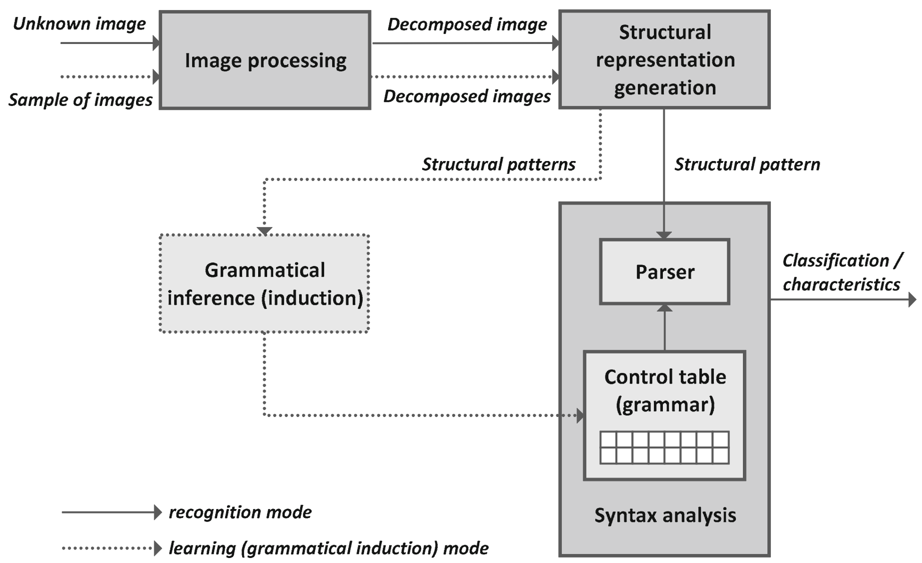


Fig. 1 A general scheme of a syntactic pattern recognition system

Then, a graph representation is usually used. It means that one should use a graph grammar for a generation of a set of patterns and a graph automaton (parser) for its analysis. Unfortunately, a problem of parsing of non-trivial graph languages is PSPACE-complete or NP-complete [4, 51, 56]. Therefore, defining graph grammars generating languages with a polynomial membership problem is the second crucial open problem in syntactic pattern recognition.

Before we consider two open key problems identified above in Sects. 3 and 4, respectively, we try to formulate in Sect. 2 some general methodological recommendations concerning a research in syntactic pattern recognition. Our considerations are based on the 20 years research experience in both string-based and graph-based syntactic pattern recognition [12–22, 33, 35]. Hopefully, our recommendations concerning methodological aspects of a research in syntactic pattern recognition launch a discussion on prospects and limitations of a future development of this field.

2 General remarks on syntactic pattern recognition model

As we have mentioned it in a previous section, a grammar (a pattern generator) and an automaton (a pattern recognizer/analyzer) are basic formalisms of syntactic pattern recognition. For most applications of the theory of formal languages, including: programming languages, a construction of compilers, etc., these formalisms are sufficient, since a grammar is defined by a designer on the basis of a well-defined syntax of the language.

In case of syntactic pattern recognition, however, a syntax of the language is not known in an explicit way, and only a sample of patterns is given. Since usually a number of sample patterns is big, defining a grammar "by hand" is impossible. Therefore, one has to construct an algorithm of a grammatical inference (induction) that generates a grammar automatically on the basis of the sample. Defining such an algorithm is much more difficult than defining an algorithm of generating a control table for an automaton on the basis of the grammar. On the other hand, a lack of a grammatical inference algorithm makes the use of a syntactic pattern recognition model impossible in most of real-world applications [26, 29]. This algorithm, allowing one to devise a pattern recognition system with a self-learning mechanism (cf. Fig. 1), need not be as efficient as a parsing algorithm, since inductive learning of the system is usually made in the off-line mode. Thus, its (any) polynomial complexity is enough[1]. Summing up our considerations, let us formulate the first methodological recommendation concerning a syntactic pattern recognition model.

I. A syntactic pattern recognition model should be complete. It means that it should consist of the following three components: a grammar, an efficient syntax analyzer and a grammatical inference algorithm of a polynomial complexity.

Before we analyze conditions of an efficient use of syntactic approach in a visual pattern recognition, we discuss assumptions that are made for such an application of grammar-based techniques. An analysis of an image in a syntactic pattern recognition system begins with an image processing phase (cf. Fig. 1). Typical operations during this phase include: noise reduction, smoothing, boundary sharpening/accentuation, edge detection, segmentation, etc. These enhancement/restoration operations are performed in order to improve a quality of an image and to make an image analysis more effective. A selection of preprocessing

[1] On the other hand, a parsing algorithm should be as efficient as it is possible especially, if a recognition performed by the system is to be made in a real-time mode.

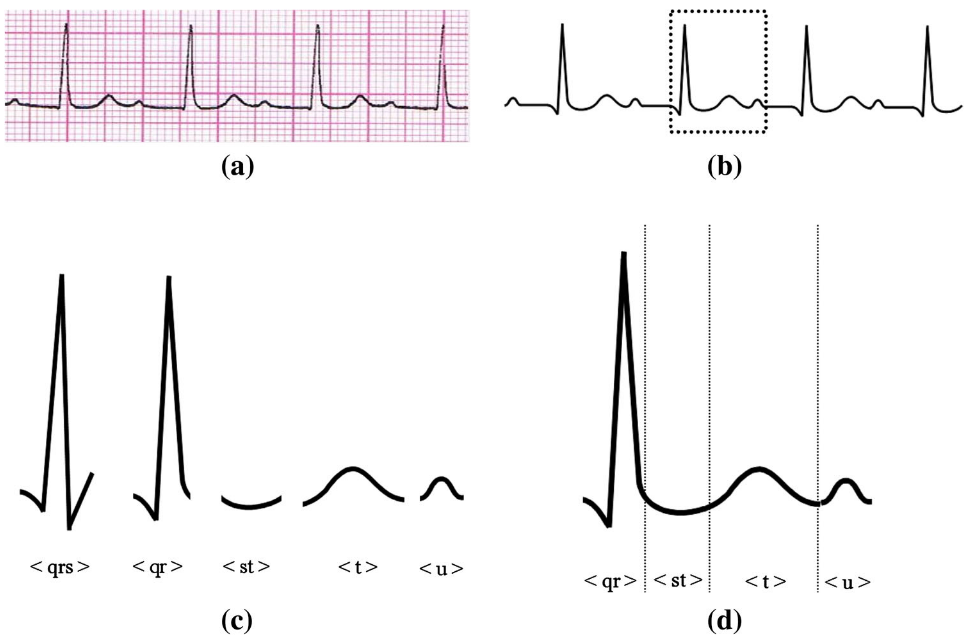


Fig. 2 Phases of image processing in a syntactic pattern recognition system (ECG): **a** an input image, **b** an image after an image processing phase, **c** examples of ECG primitives (elementary patterns), **d** a structural representation after a segmentation

operations mainly depends on a nature of an image and conditions of image acquisition. For example, an image of ECG shown in Fig. 2a processed in a syntactic pattern recognition system has been, firstly, filtered (grid lines have been removed) and smoothed (cf. Fig. 2b) in order to make an identification of typical ECG structural components more effective. Secondly, in order to identify predefined ECG primitives (elementary patterns), some of them shown in Fig. 2c, the image has been segmented. The result of such a segmentation for a fragment of the image is shown in Fig. 2d. Let us notice that now the image is represented as a (generalized) structural pattern. A description of the sub-image shown in Fig. 2d of the form <qr> <st> <t> <u>, where <qr>, <st>, <t>, <u> are symbols representing primitives, is treated as a word of a formal language consisting of possible patterns of ECG images.

An abstract/generalized representation of a pattern as a structure defined with a pre-defined *primitives* is a good point of a syntactic approach, since it is a kind of an analogy to a recognition based on a pre-defined perceptual *concepts* made by a human being. On the other hand, such a generalization of phenomena (images) performed by a computer system can be too rough, because of a fuzzy/vague nature of the real-world phenomena. Therefore, a symbolic representation-based syntactic pattern recognition scheme has been often "enhanced" in order to handle a problem of a fuzziness of the real-world phenomena, as well as a problem of a noise/distortion appearing at a stage of an image acquisition [24].

In the first approach we define transformations corresponding to distortions of strings representing patterns. There are three kinds of such distortions. A substitution error consists in an occurrence of a terminal symbol a instead of b in a string, which usually is a result of a misrecognition of a primitive. Deletion or insertion errors appear when there is a lack of some terminal symbol in a phrase or a certain symbol occurs, whereas it should not, respectively. These two errors result usually from segmentation errors. Having all the possible errors determined, we should *expand* a grammar generating "ideal" patterns by adding productions corresponding to error transformations. Now, we can use a parser, which computes a distance between an analyzed string x and a proper string y (i.e. a string belonging to an underlying language). Such a parser is called a *minimum-distance error-correcting parser, MDECP* [2]. This distance can be computed simply as the smallest number of error transformations required to obtain a string x from a string y. If we ascribe various costs (weights) to various error transformations, a weighted distance can be calculated.

If errors resulted from preprocessing phases are more "subtle" than differences between symbolic (category-based) primitives, *attributed grammars* are applied [36]. In such an approach, attributes which characterize features of primitives in detail (e.g. numeric features) are used. Productions of an attributed grammar contain a syntactic part (corresponding to "standard" productions of non-attributed grammars) and a "semantic" part, called a *semantic rule*. Such a rule allows one to evaluate attributes of certain symbols appearing in the production in terms of attributes of other symbols. A distance between an analyzed pattern and the language consisting of model ("ideal") patterns can be computed during parsing not only on the basis of structural distortions, but also with the help of vectors of attributes.

The third approach to a syntax analysis of noisy patterns can be used, if one is able to observe that some patterns occur more frequently than others. Such a phenomenon can be noticed, for example, during a process of a grammatical inference performed on the basis of a sample of patterns. In such a case occurrences of patterns can be used for evaluating their probabilities. As a result a *stochastic grammar* can be defined [23]. In such a grammar, probabilities are assigned to productions, so during a derivation a probability of a generated pattern can be computed. A corresponding parser, called a *maximum-likelihood error-correcting parser, MLECP*, evaluates additionally a probability with which an unknown pattern belongs to an underlying language.

After a brief presentation of the main approaches to a problem of a fuzzy/vague nature of real-world phenomena, we can formulate the second methodological remark concerning a pattern recognition model.

II. If a syntactic pattern recognition model is to be used for a classification/interpretation of real-world objects or phenomena[2], it should be enhanced with a mechanism allowing one to handle a problem of their fuzzy/vague nature. Error-correcting parsing, attributed grammars and stochastic grammars are typical enhancement mechanisms applied in such a case.

Decision-theoretic classification methods make use of a generic pattern representation of the form of a feature vector. In consequence they are all-purpose in a sense they can be applied for various application areas. On the contrary, developing a syntactic model, we define a representation, which is adequate (so, specific) for a given application area, i.e. a nature of patterns occurring in this area [6, 24, 27, 42, 55]. A form of a structural representation determines, in turn, a form (type) of a formal grammar which is a basis for a construction of a model. A generative power of a formal grammar is its fundamental characterization. In order to define it formally, we introduce firstly basic notions. We will make it in a general way, i.e. we do not determine a grammar structure (like a quadruple-structure for standard Chomsky's grammars), since the structure varies for grammars considered in this paper.

If Σ is a set of any symbols, then Σ^* (*Kleene star*) denotes a set of all strings that can be constructed by catenating symbols of Σ, including the empty string (empty word), denoted with λ. A language L is a subset of Σ^*.

Let G be a grammar. Let components of G are denoted in the following way. V is a set of symbols (alphabet). $\Sigma \subset V$ is a set of terminal symbols, i.e. symbols that occur in words of a language generated with G. P is a set of productions (rules) used to generate a language. A production is denoted by: $\gamma \longrightarrow \delta,\ \gamma, \delta \in V^*$, which means that a substring γ can be replaced by a substring δ. $N = V \setminus \Sigma$ is a set of nonterminal symbols. Nonterminal symbols are auxiliary symbols and they are used in a process of deriving language words with the help of productions. (They play a role similar to variable symbols in mathematics.) They do not occur in words of a language generated with G. (The language contains only terminal symbols.) $S \in N$ is the starting symbol.

An application of a production to a string $\alpha \in V^*$ that results in obtaining a string $\beta \in V^*$ is called a derivation step, denoted $\alpha \Longrightarrow \beta$. Thus, for defining a production (rule) we use a symbol $\longrightarrow$, whereas for denoting its application a symbol $\Longrightarrow$ is used. A sequence of derivation steps (including the empty sequence) is denoted with $\overset{*}{\Rightarrow}$.

A language generated with G is a set $L(G) = \{\alpha | S \overset{*}{\Rightarrow} \alpha,\ \alpha \in \Sigma^*\}$.

Let X denotes a type of formal grammars. A class X of languages is a set $\mathcal{L}(X) = \{L | \exists G\, \textit{of the type}\, X : L = L(G)\}$, i.e. it is a set containing all the languages L that can be generated with any grammar G of the type X. We say that grammars of a type X are of a bigger generative power than grammars of a type Y, if $\mathcal{L}(Y) \subsetneq \mathcal{L}(X)$.

In general, the bigger generative power of a grammar is, the bigger computational complexity of the corresponding automaton is. Moreover, in case of a growth of a generative power of a grammar, constructing an efficient inference algorithm is even more difficult than defining an efficient automaton. Summing up our considerations, we can propose the following methodological principle.

III. Any syntactic pattern recognition method should be constructed for a specific problem of a strictly-defined application area, and with the use of the Ockham Razor principle with respect to generative power of an underlying grammar. That is, a grammar should be of the smallest generative power yet sufficient to generate all the possible patterns.

3 Enhanced string context-free grammars

In an introduction we have identified an issue of an enhancement of a generative power of context-free grammars as the one of most important key open problems in syntactic pattern recognition. In this section we discuss it in a more detailed way.

3.1 Survey of models

In this section we present and discuss certain types of enhanced context-free grammars. Such grammars are

[2] Sometimes a syntactic pattern recognition scheme is used for analyzing objects or systems being *artefacts*, like for example a particle physics detector system (see e.g. [19]). Then, an enhancement of a syntactic model can be unnecessary.

required to generate all the context-free languages and also certain context-sensitive languages[3]. There are a lot of taxonomies and characterizations of enhanced CFGs. In the theory of formal languages Dassow and Păun [8, 9] have defined a taxonomy for enhanced CFGS, called here *regulated rewriting (controlled) grammars* that is of a great importance for studying formal properties of such grammars. In the field of Natural Language Processing (NLP) various types of enhanced CFGs, which are convenient for solving crucial problems in this area, have been defined within a class of the so-called *mildly context-sensitive grammars, MCSGs* [57]. We will try to analyze important types of grammars from the point of view of syntactic pattern recognition. Especially, we will have in mind the first methodological recommendation formulated in a previous section, that is a possibility of constructing an efficient parser and a polynomial inference algorithm.

In order to enhance a context-free grammar, we should devise it with an ability of controlling a derivation process. In a standard (Chomskyan) paradigm it can be made either by including certain derivation control operators in grammar productions or by defining a separated (w.r.t. productions) derivation control mechanism. We begin with the first approach. An indexed grammar [1] introduced by Aho in 1968 was the first type of grammars developed within this approach. Let us define it formally.

Definition 1 An indexed grammar is a quintuple $G = (V, \Sigma, I, P, S)$, where: V, $\Sigma \subset V$, $S \in N$ are defined as in a previous section, I is a set of indices, P is a finite set of productions of one of the three forms:

$$(1)\ A \longrightarrow \alpha \quad \text{or} \quad (2)\ A[..] \longrightarrow B[i..] \quad \text{or}$$
$$(3)\ A[i..] \longrightarrow [..]\alpha,$$

where A and $B \in N$, $i \in I$, $[..]$ represents a stack of indices, a string in I^*, $\alpha \in V^*$.

Indices may follow any nonterminal and they are introduced in order to model a context for a derivation. Let us propose the following notation:

- $[..]$ represents a stack of indices, a string in I^*,
- $[i..]$ represents a stack of indices where $i \in I$ is the top element of the stack.

Let $A \in N, B \in N, X_j \in V, \beta \in V^*, \gamma \in V^*, i \in I, \delta_j \in I^*$. A derivation in indexed grammars is defined in the following way.

[3] One can enhance other classes of grammars than CFGs with mechanisms discussed in the paper. Nevertheless, in syntactic pattern recognition we are interested primarily in enhancing CFGs.

1. If $A \longrightarrow X_1, \ldots, X_k$ is a production of type (1), then $\beta A\delta\gamma \Rightarrow \beta X_1\delta_1$, …, $X_k\delta_k\gamma$, where $\delta_j = \delta$ if $X_j \in N$ and $\delta_j = \lambda$ if $X_j \in \Sigma$.
2. If $A[..] \longrightarrow B[i..]$ is a production of type (2), then $\beta A\delta\gamma \Rightarrow \beta Bi\delta\gamma$.
3. If $A[i..] \longrightarrow [..]X_1, \ldots, X_k$ is a production of type (3), then $\beta Ai\delta\gamma \Rightarrow \beta X_1\delta_1$, …, $X_k\delta_k\gamma$, where $\delta_j = \delta$ if $X_j \in N$ and $\delta_j = \lambda$ if $X_j \in \Sigma$.

Firstly, in order to show how such a derivation is performed, we define a simple indexed grammar G generating a context-sensitive language $L(G) = \{a^{2^n}, n > 0\}$. Let $G = (\{S, A, a\}, \{a\}, \{i\}, P, S)$, where P is:

1.	$S[..] \rightarrow S[i..]$	(a production of a type 2)
2.	$S \rightarrow AA$	(a production of a type 1)
3.	$A[i..] \rightarrow [..]AA$	(a production of a type 3)
4.	$A \rightarrow a$	(a production of a type 1)

Now, e.g. a string a^8 is derived in the following way.

$$S[] \overset{1}{\Rightarrow} S[i] \overset{1}{\Rightarrow} S[ii] \overset{2}{\Rightarrow} A[ii]A[ii] \overset{3}{\Rightarrow} A[i]A[i]A[ii] \overset{3}{\Rightarrow} A[i]A[i]A[i]A[i]$$
$$\overset{3}{\Rightarrow} A[]A[]A[i]A[i]A[i] \overset{3}{\Rightarrow} \cdots \overset{3}{\Rightarrow} A[]A[]A[]A[]A[]A[]A[]A[]$$
$$\overset{4}{\Rightarrow} aA[]A[]A[]A[]A[]A[]A[] \overset{4}{\Rightarrow} \cdots \overset{4}{\Rightarrow} aaaaaaaa$$

A symbol $\overset{k}{\Rightarrow}$ denotes an application of the kth production.

In spite of a concise form of indexed grammars, they are of a big descriptive power, which is enough to generate such complex structural patterns like e.g. fractals. For example, let us define an indexed grammar G used for generating the Sierpinski Triangle (see Fig. 3a) as an image called the Sierpinski tiling arrowhead (the grammar generates the basic structure of the image).

Let $G = (\{S, A, B, d, l, r\}, \{d, l, r\}, \{i\}, P, S)$, where P is:

1.	$S[..] \rightarrow S[i..]$	(a production of a type 2)
2.	$S \rightarrow A$	(a production of a type 1)
3.	$A[i..] \rightarrow [..]BrArB$	(a production of a type 3)
4.	$B[i..] \rightarrow [..]AlBlA$	(a production of a type 3)
5.	$A \rightarrow d$	(a production of a type 1)
6.	$B \rightarrow d$	(a production of a type 1)

The primitives d, l, r are defined in the following way (cf. Fig. 3b):

- d is a straight line segment,
- l is left 60° "turn",
- r is right 60° "turn".

Let us derive one of basic forms of the Sierpinski tiling arrowhead (see Fig. 3f) as follows.

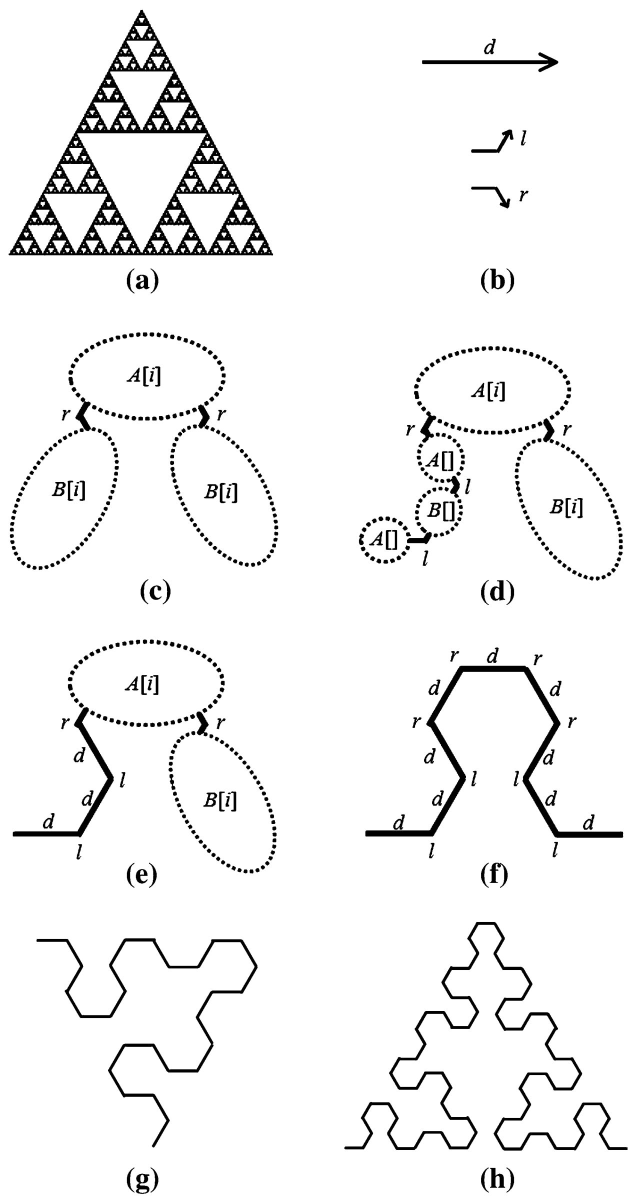


Fig. 3 **a** The Sierpinski *Triangle*, **b** primitives used for a generation of the Sierpinski *tiling arrowhead*, **c–e** an image interpretation of chosen derivation steps, **f** the basic generator of the Sierpinski *tiling arrowhead*, **g** a successive structure when the first production is applied three times, **h** a successive structure when the first production is applied four times

$$\begin{aligned}
&S[] \overset{1}{\Rightarrow} S[i] \overset{1}{\Rightarrow} S[ii] \overset{2}{\Rightarrow} A[ii] \overset{3}{\Rightarrow} B[i]rA[i]rB[i] \overset{4}{\Rightarrow} A[]lB[]lA[]rA[i]rB[i]\\
&\quad \overset{5}{\Rightarrow} dlB[]lA[]rA[i]rB[i] \overset{6}{\Rightarrow} dldlA[]rA[i]rB[i] \overset{5}{\Rightarrow} dldldrA[i]rB[i]\\
&\quad \overset{3}{\Rightarrow} dldldrB[]rA[]rB[]rB[i] \overset{6}{\Rightarrow} dldldrdrA[]rB[]rB[i]\\
&\quad \overset{5}{\Rightarrow} dldldrdrdrB[]rB[i] \overset{6}{\Rightarrow} dldldrdrdrdrB[i]\\
&\quad \overset{4}{\Rightarrow} dldldrdrdrdrA[]lB[]lA[] \cdots \overset{5}{\Rightarrow} \cdots \overset{6}{\Rightarrow} \cdots\\
&\quad dldldrdrdrdrdldld
\end{aligned}$$

Let us describe the derivation in an intuitive way. We begin with putting an index i on the stack with the help of the first production. The more indices i we put on the stack the more complex structure we receive. We start a proper generation of a structure by applying the second production. Productions: 3 and 4 generate primitives: r and l (at the same time they remove indices i from the stacks). The effect of the first application of a production 3 is shown in Fig. 3c. The effect of the first application of a production 4 is shown in Fig. 3d. Productions: 5 and 6 generate a primitive d (cf. Fig. 3e). The final effect of the derivation is shown in Fig. 3f.

To obtain a more complex structure than the one shown in Fig. 3f, the first production should be applied three times (cf. Fig. 3g). If we apply the first production four times a successive more complex structure is received (cf. Fig. 3h), etc.

As one can see, additional stacks of indices assigned to nonterminals are used in indexed grammars. If a production is applied to a nonterminal with a stack, then all nonterminals of the right-hand side of the production receive copies of this stack. Such a mechanism allows us to reflect contextual dependencies during a derivation. Neither a polynomial parser nor a polynomial grammatical inference algorithm have been defined for indexed grammars.

Let us notice that some additional syntactic constructs (brackets: [ ], indices) occur in both grammar productions and non-final phrases of a derivation, apart from terminal and nonterminal symbols. These constructs do not occur in words of a language generated and they play a role of *operators* controlling a derivation. An occurrence of such operators is typical for mildly context-sensitive grammars (MCSGs) [57] used in NLP and mentioned above. Mildly context-sensitive languages (MCSLs) fulfill the following properties. MCSLs contain context-free languages and certain languages with context dependencies ($L_1 = \{a^n b^n c^n | n \geq 0\}, L_2 = \{a^n b^m c^n d^m | n, m \geq 0\}, L_3 = \{ww | w \in \{a, b\}^*\}$). Their membership problem is solvable in a deterministic polynomial time. MCSLs have the linear growth property (if strings of a language are ordered in a sequence according to their length, then two successive lengths do not differ in arbitrary large amounts). The best known MCSGs include: *linear indexed grammars*, *head grammars*, and *combinatory categorial grammars*[4].

Now, we briefly characterize MCSGs mentioned above. Let us start with linear indexed grammars (LIGs) introduced by Gazdar [25]. LIG differs from an indexed grammar in the form of productions. In LIG at most one nonterminal in each production receives the stack of

[4] *Tree adjoining grammars* are the fourth well-known MCSGs, but, in fact, they are tree (not string) grammars.

indices. (In indexed grammars *all* nonterminals receive copies of the stack.) Let us introduce the following definition.

Definition 2 A linear indexed grammar, LIG, is a quintuple $G = (V, \Sigma, I, P, S)$, where $V, \Sigma \subset V$, $S \in N$ are defined as in a previous section, I is a set of indices, P is a finite set of productions of one of the three forms:

(1) $A[..] \longrightarrow \alpha B[..]\gamma$ or (2) $A[i..] \longrightarrow \alpha B[..]\gamma$ or (3) $A[..] \longrightarrow \alpha B[i..]\gamma$,

where: A and $B \in N$, $i \in I$, $[..]$ represents a stack of indices, a string in I^*, α and $\gamma \in V^*$.

For example, let us define a linear indexed grammar G such that $L(G) = \{a^n b^n c^n d^n,\ n > 0\}$. Let $G = (\{S, B, a, b, c, d\}, \{a, b, c, d\}, \{i\}, P, S)$, where P is:

1.	$S[..] \rightarrow aS[i..]d$	(a production of a type 3)
2.	$S[..] \rightarrow B[..]$	(a production of a type 1)
3.	$B[i..] \rightarrow bB[..]c$	(a production of a type 2)
4.	$B[] \rightarrow \lambda$	(a production of a type 1)

A derivation of a string *aabbccdd* is made in the following way.

$$S[] \overset{1}{\Rightarrow} aS[i]d \overset{1}{\Rightarrow} aaS[ii]dd \overset{2}{\Rightarrow} aaB[ii]dd \overset{3}{\Rightarrow} aabB[i]cdd$$
$$\overset{3}{\Rightarrow} aabbB[]ccdd \overset{4}{\Rightarrow} aabbccdd$$

A polynomial parsability of LIGs is their main advantage. On the other hand, the grammars are of the less descriptive power than (common) indexed grammars as one can easily see in the example above. The other models based on indexed grammars include: distributed index grammars [52], global index grammars [7], and sequentially indexed grammars [10]. Last two types of grammar have been constructed in order to preserve as much as possible of a generative power of indexed grammars, being still parsable in a polynomial time.

The head grammars (HGs) were introduced by Pollard in 1984 [46]. They are defined in the following way.

Definition 3 A head grammar, HG, is a quadruple $G = (V, \Sigma, P, S)$, where V, $\Sigma \subset V$, $S \in N$ are defined as in a previous section, P is a set of productions of the form:

$A \rightarrow f(\alpha_1, \ldots, \alpha_n)$ or $A \rightarrow \alpha_1$ where:

$A \in N$, α_i is either a nonterminal or a headed string, f is either a concatenation or a head wrapping operation.

Head grammars differ from context-free grammar in containing a distinguished symbol "$\uparrow$" in each string. This symbol corresponds to *the head* of the string. The nonterminals of a head grammar derive headed strings or pairs of terminal strings (u, v) that we denote $(u \uparrow v)$. There are two types of operations that can be performed using the head. The first is a *concatenation* $C_{i,n}$. It joins n head-divided words in order and inserts a new head in the string $C_{i,n}(u_1 \uparrow v_1, \ldots, u_i \uparrow v_i, \ldots, u_n \uparrow v_n) = u_1v_1, \ldots, u_i \uparrow v_i, \ldots, u_nv_n$. The second operation is *wrapping* W which inserts one word into another based on the head position $W(u_1 \uparrow v_1, u_2 \uparrow v_2) = u_1u_2 \uparrow v_2v_1$.

For example, let us define a head grammar G such that $L(G) = \{a^n b^n c^n d^n,\ \ n > 0\}$. Let $G = (\{S, T, a, b, c, d\}, \{a, b, c, d\},\ \ P, S)$, where P is:

1. $S \rightarrow C_{1,1}(\lambda \uparrow \lambda)$ 2. $S \rightarrow C_{2,3}(a \uparrow \lambda, T, d \uparrow \lambda)$
3. $T \rightarrow W(S, b \uparrow c)$

A derivation of a string *aabbccdd* is made as follows.

$$S \overset{1}{\Rightarrow} C_{1,1}(\lambda \uparrow \lambda) = \lambda \uparrow \lambda$$
$$T \overset{3}{\Rightarrow} W(\lambda \uparrow \lambda, b \uparrow c) = b \uparrow c$$
$$S \overset{2}{\Rightarrow} C_{2,3}(a \uparrow \lambda, b \uparrow c, d \uparrow \lambda) = ab \uparrow cd$$
$$T \overset{3}{\Rightarrow} W(ab \uparrow cd, b \uparrow c) = abb \uparrow ccd$$
$$S \overset{2}{\Rightarrow} C_{2,3}(a \uparrow \lambda, abb \uparrow ccd, d \uparrow \lambda) = aabb \uparrow ccdd$$

Combinatory categorial grammars (CCGs) were introduced by Steedman in 1987 [53]. Let us introduce their definition.

Definition 4 A combinatory categorial grammar, CCG, is a quintuple $G = (V, \Sigma, f, R, S)$, where V is a finite, nonempty alphabet, $\Sigma \subset V$ is a finite, nonempty set of terminal symbols—lexical items (with N we denote a set of nonterminal symbols $N = V \setminus \Sigma$—such symbols are also called "atomic categories" which can be combined into more complex functional categories by using the backward operator\or the forward operator /), $S \in N$ is the starting symbol, f is the terminal function that maps terminal symbols to finite subsets of $C(N)$, the set of categories, where $N \subseteq C(N)$ and if $c_1, c_2 \in C(N)$ then $(c_1/c_2) \in C(N)$ and $(c_1 \backslash c_2) \in C(N)$, R is a set of combinatory rules of one of the four forms, that involve variables x, y, z over $C(N)$, and each $|_i \in \{\backslash, /\}$:

1. forward application: $(x/y)\, y \rightarrow x$
2. backward application: $y\ (x\backslash y) \rightarrow x$
3. generalized forward composition for some $n \geq 1$: $(x/y)\ (\ldots(y|_1z_1)|_2\ldots|_nz_n) \rightarrow (\ldots(x|_1z_1)|_2\ldots|_nz_n)$
4. generalized backward composition for some $n \geq 1$: $(\ldots(y|_1z_1)\ |_2\ldots\ |_nz_n)\ (x\backslash y) \rightarrow (\ldots\ (x|_1z_1)\ |_2\ldots|_nz_n)$

Derivations in a CCG involve the use of the combinatory rules in R (instead of productions in a "common" formal grammar). Let the "derives" relation be defined as: $\alpha c\beta \Rightarrow \alpha c_1c_2\beta$ if R contains a combinatory rule that has $c_1c_2 \rightarrow c$ as an instance, and α and β are string of categories. Then the string languages generated by a CCG is defined as:

$L(G) = \{a_1,\ldots,a_n | S \Rightarrow \cdots \Rightarrow c_1,\ldots,c_n, c_i \in f(a_i), a_i \in \Sigma \cup \{\lambda\}, 1 \le i \le n\}$.

For example, let us define a combinatory categorial grammar G such that $L(G) = \{a^n b^n c^n d^n, \ n > 0\}$. Let $G = (\{S, T, A, B, D, a, b, c, d\}, \ \{a, b, c, d\}, \ f, \ R, \ S)$, where R is:

rule r1: $(x^S/T)(T\backslash A/T\backslash B) \to (x^S\backslash A/T\backslash B)$
rule r2: $(A/D)(x^S\backslash A) \to (x^S/D)$
rule r3: $(x^S/y)y \to x^S$
rule r4: $y(x^S\backslash y) \to x^S$

and f is:

f1: $f(a) = \{(A/D)\}$
f2: $f(b) = \{ B \}$
f3: $f(d) = \{D\}$
f4: $f(c) = \{(T\backslash A/T\backslash B)\}$
f5: $f(\lambda) = \{ (S/T), T \}$

A derivation of a string *abcd* is performed in the following way.

$$S \Longrightarrow (S\backslash D)D \Longrightarrow (A/D)(S\backslash D)D \Longrightarrow (A/D)(S\backslash D/T)TD$$
$$\Longrightarrow (A/D)B(S/T)(S\backslash D/T\backslash B)TD \Longrightarrow (A/D)B(S\backslash D/T\backslash B)D$$
$$\Longrightarrow aB(S\backslash D/T\backslash B)D \Longrightarrow ab(S\backslash D/T\backslash B)D \Longrightarrow abcD \Longrightarrow abcd$$

The derivation is made by applying rules: 3, 2, 4, and 1, and then $f(\lambda)$, $f(a)$, $f(b)$, $f(c)$, $f(d)$.

Similarly as in the case of mildly context-sensitive grammars applied in the field of NLP and presented above, derivation control operators included in productions have been recently used in two types of *enhanced* context-free grammars introduced by Okhotin in the theory of formal languages. These grammars allow one to specify such theoretical operations over sets of languages as their intersection, negation, etc. Let us consider the following definition [40].

Definition 5 A *conjunctive grammar* is a quadruple $G = (V, \Sigma, P, S)$, where V, $\Sigma \subset V$, $S \in N$ are defined as in a previous section, P is a finite set of rules, each of the form:

$A \to \alpha_1 \& \ldots \& \alpha_m$,

where $A \in N$, $m \ge 1$ and $\alpha_1, \ldots, \alpha_m \in V^*$. Each string α_i is called a *conjunct*.

Intuitively speaking, a rule in a conjunctive grammar specifies that every string which satisfies each of the conditions α_i is generated by A.

For example, let us define a conjunctive grammar G such that $L(G) = \{a^n b^n c^n, \ n > 0\}$. Let $G = (\{S, A, C, F, G, a, b, c\}, \{a, b, c\}, P, S)$, where P is:

$S \to AF \,\&\, GC \quad A \to Aa \mid \lambda \quad C \to Cc \mid \lambda$
$F \to bFc \mid \lambda \quad G \to aGb \mid \lambda$

In the grammar G non-terminal A generates any number of a symbols, while F generates strings with equal numbers of b symbols and c symbols (b^n c^n). On the other hand, G generates strings with equal numbers of a symbols and b symbols ($a^n b^n$) while C generates strings with any number of c symbols. By taking the conjunction of the languages associated with AF and GC (since $S \to AF \,\&\, GC$), grammar G generates the language $L(G) = \{a^n b^n c^n, \ n > 0\}$.

Boolean grammars defined by Okhotin in 2004 [41] are more general than conjunctive grammars. Additionally, a negation operator can be used in productions that results in a possibility of expressing every Boolean operation over sets of languages. Both conjunctive and Boolean grammars generate all context-free languages and a subset of context-sensitive languages. Polynomial parsers have been defined for both classes. An investigation in grammatical inference has not been led, because of theoretical objectives of the research (enhancing generative power of CFGs allowing to express logical operations over sets of underlying context-free languages).

After presenting types of grammars with derivation control operators included in productions, let us introduce grammars with a separated control mechanism, i.e. the mechanism that is not "hidden" in left- or right-hand sides of a production. Such a methodology is used in programmed grammars introduced by Rosenkrantz in 1969 [48].

Definition 6 A programmed grammar is a quintuple $G = (V, \Sigma, J, P, S)$, where: $V, \Sigma \subset V, S \in N$ are defined as in a previous section, J is a set of production labels, P is a finite set of productions of the form:

$(r) \quad \alpha \longrightarrow \beta \quad S(U)\,F(W)$, in which

$\alpha \longrightarrow \beta, \alpha \in V^*NV^*, \beta \in V^*$, is called the *core*, (r) is the production label, $r \in J$, $U \subset J$ is the success field and $W \subset J$ is the failure field.

A derivation is defined as follows. A production labelled with (1) is applied firstly. If it is possible to apply a production (r), then after its application the next production is chosen from its success field U. Otherwise, we choose the next production from the failure field W.

Let us define a programmed grammar G such that $L(G) = \{a^n \ b^n \ c^n, \ n > 0\}$. Let $G = (\{S, A, B, C, a, b, c\}, \{a, b, c\}, \{1, \ldots, 7\}, P, S)$, where P is:

1.	$S \to ABC$	$S(\{2,5\})$	$F(\emptyset)$	5.	$A \to a$	$S(\{6\})$	$F(\emptyset)$
2.	$A \to aA$	$S(\{3\})$	$F(\emptyset)$	6.	$B \to b$	$S(\{7\})$	$F(\emptyset)$
3.	$B \to bB$	$S(\{4\})$	$F(\emptyset)$	7.	$C \to c$	$S(\{\emptyset\})$	$F(\emptyset)$
4.	$C \to cC$	$S(\{2,5\})$	$F(\emptyset)$				

For example, a derivation of a string *aabbcc* is made as follows.

$$S \overset{1}{\Rightarrow} ABC \overset{2}{\Rightarrow} aABC \overset{3}{\Rightarrow} aAbBC \overset{4}{\Rightarrow} aAbBcC$$
$$\overset{5}{\Rightarrow} aabBcC \overset{6}{\Rightarrow} aabbcC \overset{7}{\Rightarrow} aabbcc$$

Now, we use once more an example of the Sierpinski Triangle (see Fig. 3a) for showing a big descriptive power of a programmed grammar. Let us define a programmed grammar G, which generates the Sierpinski tiling arrowhead (it has been generated with an indexed grammar at the beginning of this section).

Let $G = (\{S, A_n, B_n, A_o, B_o, r, l, d\}, \{r, l, d\}, \{1, \ldots, 7\}, P, S)$, where primitives r, l and d are defined as in Fig. 3b, and the set of productions P is:

1.	$S \longrightarrow B_n r A_n r B_n$	$S(\{2, 6\})$	$F(\emptyset)$
2.	$A_n \longrightarrow A_o$	$S(\{2\})$	$F(\{3\})$
3.	$B_n \longrightarrow B_o$	$S(\{3\})$	$F(\{4\})$
4.	$A_o \longrightarrow B_n r A_n r B_n$	$S(\{4\})$	$F(\{5\})$
5.	$B_o \longrightarrow A_n l B_n l A_n$	$S(\{5\})$	$F(\{2, 6\})$
6.	$A_n \longrightarrow d$	$S(\{6\})$	$F(\{7\})$
7.	$B_n \longrightarrow d$	$S(\{7\})$	$F(\emptyset)$

Let us derive the basic generator of the Sierpinski tiling arrowhead (see Fig. 4f) as follows.

$$S \overset{1}{\Rightarrow} B_n r A_n r B_n \overset{2}{\Rightarrow} B_n r A_o r B_n \overset{3}{\Rightarrow} B_o r A_o r B_n \overset{3}{\Rightarrow} B_o r A_o r B_o$$
$$\overset{4}{\Rightarrow} B_o r B_n r A_n r B_n r B_o \overset{5}{\Rightarrow} A_n l B_n l A_n r B_n r A_n r B_n r B_o$$
$$\overset{5}{\Rightarrow} A_n l B_n l A_n r B_n r A_n r B_n r A_n l B_n l A_n \overset{6}{\Rightarrow} d l B_n l A_n r B_n r A_n r B_n r A_n l B_n l A_n$$
$$\overset{6}{\Rightarrow} \cdots \overset{6}{\Rightarrow} d l B_n l d r B_n r d r B_n r d l B_n l d \overset{7}{\Rightarrow} \cdots \overset{7}{\Rightarrow} d l d l d r d r d r d r d l d l d$$

Let us describe the derivation in an intuitive way. The successive iterations of a development of subsequent structures are "programmed" in the grammar G. We start with applying the first production (see Fig. 4a). Secondly, all the nonterminals indexed with n (i.e. A_n and B_n) are replaced with nonterminals indexed with o (i.e. A_o and B_o) with the help of productions: 2 and 3 (see Fig. 4b). Then, each nonterminal indexed with o is developed into a substructure $B_n r A_n r B_n$ or $A_n l B_n l A_n$ with the help of productions: 4 or 5, respectively (see Fig. 4c, d). At this moment, we can replace all the nonterminals with terminals d with the help of productions: 6 and 7 (cf. Fig. 4e, f) finishing the generation or we can begin the next iteration starting from a form shown in Fig. 4d.

A static control mechanism of programmed grammars (success and failure fields include fixed indices of productions) has been extended in DPLL(k) grammars (Dynamically Programmed LL(k) grammars) [19]. Instead of success and failure fields, every production is devised with a control tape. A head of a tape can write/read indices of productions and it can move. A derivation is made according to a content of a tape. We introduce the following definition.

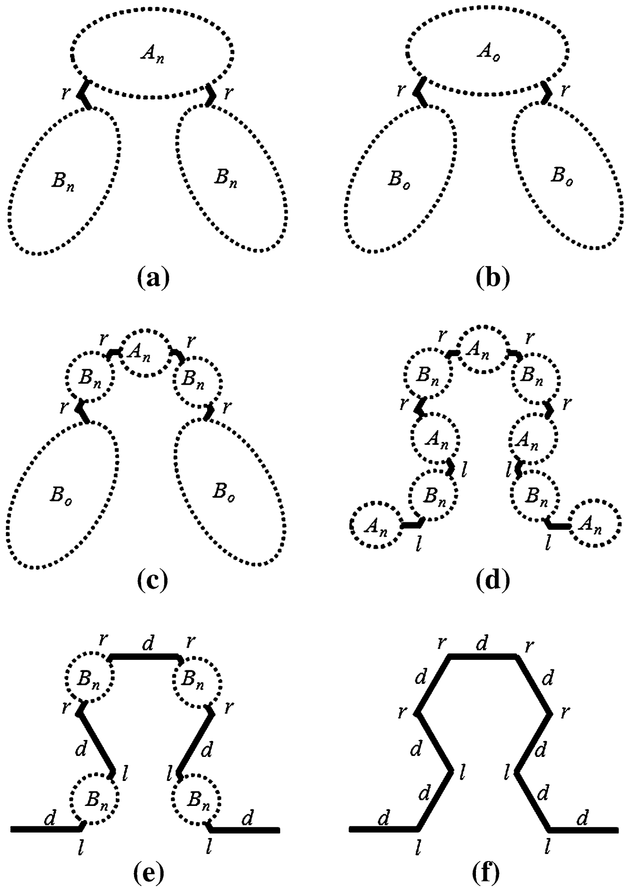


Fig. 4 A generation of the Sierpinski *tiling arrowhead* with the help of a programmed grammar: **a** a structure after an application of the first production, **b** a structure after applying productions: 2 and 3, **c–e** structures after applying productions: 4, 5 and 6, **f** the final form of the first iteration (the basic generator)

Definition 7 A dynamically programmed context-free grammar, a DP grammar, is a quintuple $G = (V, \Sigma, O, P, S)$, where V, $\Sigma \subset V, S \in N$ are defined as in a previous section, O is a set of operations on a tape: *add, read, move*, P is a finite set of productions of the form:

$p_i = (\mu_i, L_i, R_i, A_i, DCL_i)$, in which

$\mu_i : \bigcup DCL_k \longrightarrow \{TRUE, FALSE\}$ is the predicate of applicability of the production $p_i, L_i \in N$ and $R_i \in V^*$ are left- and right-hand sides of p_i, respectively, a pair (L_i, R_i) will be called a core of p_i (we assume that for each two various productions p_i, p_j from P, the core of p_i is different from the core of p_j, i.e. either $L_i \neq L_j$ or $R_i \neq R_j$), A_i is the sequence of actions of a type *add, move* $\in O$ performed over $\bigcup DCL_k, DCL_i$ is a derivation control tape for p_i.

A derivation for dynamically programmed grammars is defined in the following way. Apart from testing whether L_i occurs in a sentential form derived, the predicate of applicability of a production p_i is checked. If it is true, then L_i is replaced with R_i, and actions over derivation control tapes for certain productions are performed. A derivation control tape for a production corresponds to a success field of programmed grammars. The difference is that, whereas in common programmed grammars this field is fixed at the moment of defining a grammar, in dynamically programmed grammars this "field" is dynamically filled with labels of productions during a derivation with the help of the set of actions A_i.

In order to construct a polynomial syntax analyzer for dynamically programmed grammars, restriction forcing a deterministic derivation and limiting "recursive steps" have been imposed in the following way.

Definition 8 Let $G = (V, \Sigma, O, P, S)$ be a dynamically programmed context-free grammar, $First_k(x)$ denotes a set of all the k-length terminal prefixes of strings derivable from x in a grammar G[5], $\underset{core}{\overset{*}{\Rightarrow}}$ denotes a sequence of derivation steps consisting in applying only production cores. The grammar G is called a Dynamically Programmed LL(k) grammar, DPLL(k), if the following two conditions are fulfilled.

(1) Let $w \in \Sigma^*, A \in N = V \setminus \Sigma, x, y, \alpha, \beta, \gamma \in V^*$. Then, for every two left-hand side derivations in G:

$$S \overset{*}{\Rightarrow} wA\alpha \Rightarrow w\beta\alpha \underset{core}{\overset{*}{\Rightarrow}} wx$$

$$S \overset{*}{\Rightarrow} wA\alpha \Rightarrow w\gamma\alpha \underset{core}{\overset{*}{\Rightarrow}} wy$$

such that: $First_k(x) = First_k(y)$ the following condition holds: $\beta = \gamma$.

(2) For a grammar G there exists a certain number ξ such that for any left-hand side derivation $S \overset{*}{\Rightarrow} wA\alpha \overset{\pi}{\Rightarrow} w\beta\alpha$ (where $w \in \Sigma^*, A \in N, \alpha, \beta \in V^*, \pi$ is a string of indices of productions applied) fulfilling a condition: $|\pi| \geq \xi$, the first symbol of $\beta\alpha$ is the terminal one.

The first condition is analogical to a constraint put on a context-free grammar by a definition of well-known LL(k) grammar [49] in order to make a derivation deterministic by checking the first k symbols of the right-hand sides of productions. However, in a DPLL(k) grammar there can be more than one productions generating wx from $wA\alpha$, but only for one the predicate of applicability is fulfilled at this derivational step. With such a definition, a left-hand recursion can occur. Therefore, a number of "recursive" steps is limited with the second condition.

Although DPLL(k) grammars are weaker than DP grammars, they still can generate a large subclass of context-sensitive languages. For example let us define a DPLL(k) grammar G such that $L(G) = \{a^n\ b^n\ c^n,\ n > 0\}$.

[5] $First_k(x)$ was introduced for the LL(k)-subclass of CFGs by Rosenkrantz and Stearns [49].

Let $G = (\{S, A, B, C, a, b, c\}, \{a, b, c\}, \{add, move, read\}, P, S)$, where P is:

Label	μ	Core	Actions
1	*TRUE*	$S \to aAbBcC$	$\varnothing$
2	*TRUE*	$A \to aA$	*add(4,4); add(6,6);*
3	*TRUE*	$A \to \lambda$	*add(4,5); add(6,7);*
4	*read*(4) = 4	$B \to bB$	*move(4);*
5	*read*(4) = 5	$B \to \lambda$	*move(4);*
6	*read*(6) = 6	$C \to cC$	*move(6);*
7	*read*(6) = 7	$C \to \lambda$	*move(6);*

A derivation of a string *aaabbbccc* is made in the following way.

Production	Sentence derived	DCL$_4$	DCL$_6$
	S		
1	*aAbBcC*		
2	*aaAbBcC*	$\underline{4}$	$\underline{6}$
2	*aaaAbBcC*	$\underline{4}4$	$\underline{6}6$
3	*aaabBcC*	$\underline{44}5$	$\underline{6}67$
4	*aaabbBcC*	#45	$\underline{6}67$
4	*aaabbbBcC*	$\#\#\underline{5}$	$\underline{6}67$
5	*aaabbbcC*	###_	$\underline{6}67$
6	*aaabbbccC*	###_	$\#6\underline{7}$
6	*aaabbbcccC*	###_	$\#\#\underline{7}$
7	*aaabbbccc*	###_	###_

A descriptive power of a DPLL(k) grammar has been increased in its *generalized* version, called GDPLL(k), allowing one to generate such "complex" languages as e.g. $L(G) = \{a^{2^n}, n > 0\}$ [35]. A parsing algorithm for both DPLL(k) and GDPLL(k) grammars is of the $O(n)$ computational complexity, and a grammatical inference algorithm is also of a polynomial complexity, $O(m^3 \cdot n^3)$, where m is a sample size, n is the maximum length of a string in a sample [34].

All the formalisms discussed above can be characterized as string grammars belonging to the Chomskyan paradigm, i.e. they are "enhanced" versions of the Chomsky's context-free grammars generating certain context-sensitive languages. Below we present some models going beyond this paradigm, which are also, in our opinion, worth considering in a context of syntactic pattern recognition.

A hybrid syntactic-structural model based on an *augmented regular expression, ARE* was introduced in 1997 by Alquezar and Sanfeliu [3]. Intuitively speaking, $ARE = (R, V, T, L)$ is defined by a *regular expression* R, in which the stars are replaced by natural-valued variables, called *star variables V*, and these variables are related through a finite number of *linear equations L*. Additionally, with a *star tree T* a structure of the parentheses' embedment in the expression is determined. An analysis of a string *s* in order to verify its belonging to a language *L*(*ARE*) is performed in two steps. Firstly, with parsing of *s*, which is performed by a finite state-automaton corresponding to *R*, a verification of a *general structure* of *s* is made. Secondly, if a verification is positive, testing a fulfillment of constraints *L* that result from the parsing (i.e. an evaluation of a finite linear relations between star variables) is made. With augmented regular expressions, a considerable subclass of context-sensitive languages can be represented. A syntactic-structural analysis performed according to this scheme is of a polynomial complexity. A learning method has been defined in this model, as well.

In the field of Natural Language Processing Joshi, Levy and Takahashi defined in 1975 [31, 32] a *tree adjoining grammar, TAG* belonging to a class of mildly context-sensitive grammars (MCSGs). A *TAG* generates labelled trees by an application of operations of two kinds over *initial trees* and *auxiliary trees*. These operations include: *substitution* that attaches a tree to a *substitution node* (a nonterminal leaf node marked with a special symbol) of a tree derived from an initial tree, and *adjunction* that inserts an auxiliary tree into an internal node of a tree. A *string language* generated with a TAG is defined as a set of frontiers of trees generated. Thus, a tree adjoining grammar generates a kind of derivation trees of strings belonging to a mildly context-sensitive langauge. In fact, it is a tree-rewriting system, not a string-rewriting system. Parsing for TAGs is of a polynomial complexity.

Contextual grammars, CGs, used also in the NLP area, were introduced by Marcus in 1969 [37]. CGs go beyond the Chomskyan paradigm of string rewriting. An *operation of inserting words* into derived phrases according to contextual dependencies is used here instead of Chomskyan productions involving nonterminal symbols for generating phrases. An insertion operation is performed with *contexts* being pairs of words connected with sets of words called *selectors*. During a derivation, elements of contexts are "wrapped around" associated elements of selectors, called *selector elements*. Contextual grammars of various types are incomparable with grammars of the Chomsky hierarchy, which are a "standard" formal model in syntactic pattern recognition. Nevertheless, some context-sensitive languages can be generated by CGs. They are also worth considering in a context of syntactic pattern recognition, since for some classes of CGs a polynomial parsing algorithms have been defined (e.g. [28]), as well as a polynomial algorithm of grammatical inference [38].

3.2 Methodological remarks on string-based syntactic pattern recognition

Let us begin with summarizing properties of models presented in Sect. 3.1 with respect to their completeness in a sense of the first methodological recommendation introduced in Sect. 2, i.e. a possibility of defining algorithms of: syntax analysis and grammatical inference.

General characteristics	Type of grammars	Parsing	Inference
Chomskyan string grammars with derivation control operators included in productions	Indexed [1]	Non-polyn.	Undefined
	Linear indexed [25]	Polynomial	Undefined
	Head [46]	Polynomial	Undefined
	Combinatory categorial [53]	Polynomial	Undefined
	Conjunctive [40]	Polynomial	Undefined
	Boolean [41]	Polynomial	Undefined
Chomskyan string grammars with a separated derivation control mechanism	Programmed [48]	Non-polyn.	Undefined
	DPLL(*k*) [19]	Polynomial	Polynomial
Other approaches	ARE [3]	Polynomial	Non-polyn.
	Tree adjoining [31, 32]	Polynomial	Undefined
	Marcus CG [37]	Polynomial	Polynomial

As one can see, a definition of a grammatical inference algorithm is the main problem here. In most enhanced models, such as indexed grammars, head grammars, combinatory grammars and conjunctive/Boolean grammars, a derivation control mechanism is "hidden" (cleverly) in grammar productions. It is made with the use of some syntax constructs like stacks, heads, operators that do not occur in the words of a language. Let us notice that the main idea of standard inference methods (i.e. used for regular and context-free languages) consists in looking for similarities among sample strings. An alphabet of a grammar inferred contains only terminal symbols that occur in the language sample and nonterminals that are entered in a (relatively) simple way as "classes of abstractions" for certain substrings. The only "operator" used is a simple *catenation operator*. In a way, this operator is "visible" in a derived word. However, if grammar productions contain operators that disappear during a derivation and do not occur in a derived word, a grammatical inference problem becomes very difficult. The reason is the fact that a syntax of sample words does not deliver any information related to a history of obtaining these words with such operators. It is hardly likely that algorithms reconstructing such a history can be of a polynomial complexity. On the other hand, for Chomskyan string grammars with a separated derivation control mechanism (DPLL(k)) a polynomial grammatical inference algorithm has been defined [34].

At the same time, for some mathematical linguistics models, which goes beyond a "standard" Chomsky's string grammars approach, inference algorithms have been defined. Thus, such "unconventional" approaches are worth considering as candidates for theoretical basis of syntactic pattern recognition methods.

Let us summarize our considerations on enhanced string-based syntactic pattern recognition models and parsing/inference issues with the following methodological recommendation.

IV. A possibility of defining an algorithm of grammatical inference is a key issue for constructing an effective syntactic pattern recognition system. Defining a control mechanism enhancing a grammar as a separate element makes a development of an efficient grammatical inference algorithm easier than "hiding" this mechanism in left- or right-hand sides of productions with the help of additional syntactic operators. At the same time, a possibility of applying models going beyond a standard Chomskyan string grammar paradigm in the field of syntactic pattern recognition is worth studying.

Now, we sum up the second methodological recommendation introduced in Sect. 2, i.e. a possibility of an enhancement of a syntactic pattern recognition model with: error-correcting parsing, adding attributes to a grammar or adding probability information to a grammar. Theoretical considerations verified by a practical application experience [5, 24] show that such an enhancement does not cause any problems in case of constructing a parsing algorithm for a syntactic pattern recognition. The use of attributed or stochastic grammars does not make a construction of an inference algorithm more difficult than in case of "pure" grammars either [5, 24]. However, in case of using an error-correcting parsing scheme, an expanded grammar is to be defined, as we have discussed it in Sect. 2 Usually, it is a human being who decides, which structures are "proper" and which structures are distortions of "proper" structures. Therefore, using such a scheme would hinder solving a grammatical inference problem. In fact, this problem belongs to Artificial Intelligence rather than to a pattern recognition area.

4 Parsable graph languages in syntactic pattern recognition

In spite of the fact that graph grammars have been widely used for image representation and synthesis since the late 1960s and the early 1970s (e.g. [44, 45]), when they were firstly proposed for this purpose, their application in an area of syntactic pattern recognition has been relatively rare. A possibility of their use in this area depends strongly on a balance between a generative power sufficient for representing a class of complex "multidimensional" patterns and a parsing efficiency. In this section we consider this problem and try to formulate certain recommendations concerning its possible solutions.

4.1 Survey of research into graph-based syntactic pattern recognition

Although the first graph automata were proposed in the late 1960s, only a few graph grammar-based syntactic pattern recognition models have been presented for last 40 years[6]. Web automata were proposed by Rosenfeld and Milgram in 1972 [47]. An efficient parser for expansive graph grammars was constructed by Fu and Shi in 1983 [50]. In 1990 two parsing algorithms for plex grammars were defined independently by Bunke and Haller [5], and Peng, Yamamoto and Aoki [43]. In the early 1990s two parsing methods for relational grammars were proposed independently by: Wittenburg [58], and Ferruci, Tortora, Tucci and Vitiello [11]. An efficient, $O(n^2)$, parser for the ETPL(k) subclass of the well-known edNLC [30] graph grammars was constructed in 1993 [17, 18]. A parsing method for reserved graph grammars was proposed by Zhang, Zhang and Cao in 2001 [59].

4.2 Methodological remarks on graph-based syntactic pattern recognition

Constructing parsing algorithm for graph languages is much more difficult task than for string languages. There are two reasons of such a difficulty. First of all, a graph structure is unordered by its nature, whereas a linear order is defined by a string structure[7]. During parsing, however, succeeding pieces of an analyzed structure (sub-words in case of strings, subgraphs in case of graphs), called here *handles*, are teared off repetitively in order to be matched with predefined structures (predefined on the basis of right-hand sides of grammar productions) that are stored in a parser memory. An answer to a question: *what a succeeding piece is?* is easy when there is any kind of ordering determined for an analyzed structure. In case of the lack of any order in a graph structure, this question resolves itself into the problem of subgraph isomorphism, which is the NP-complete one.

There is, however, the second reason causing constructing graph parser very difficult. In case of string

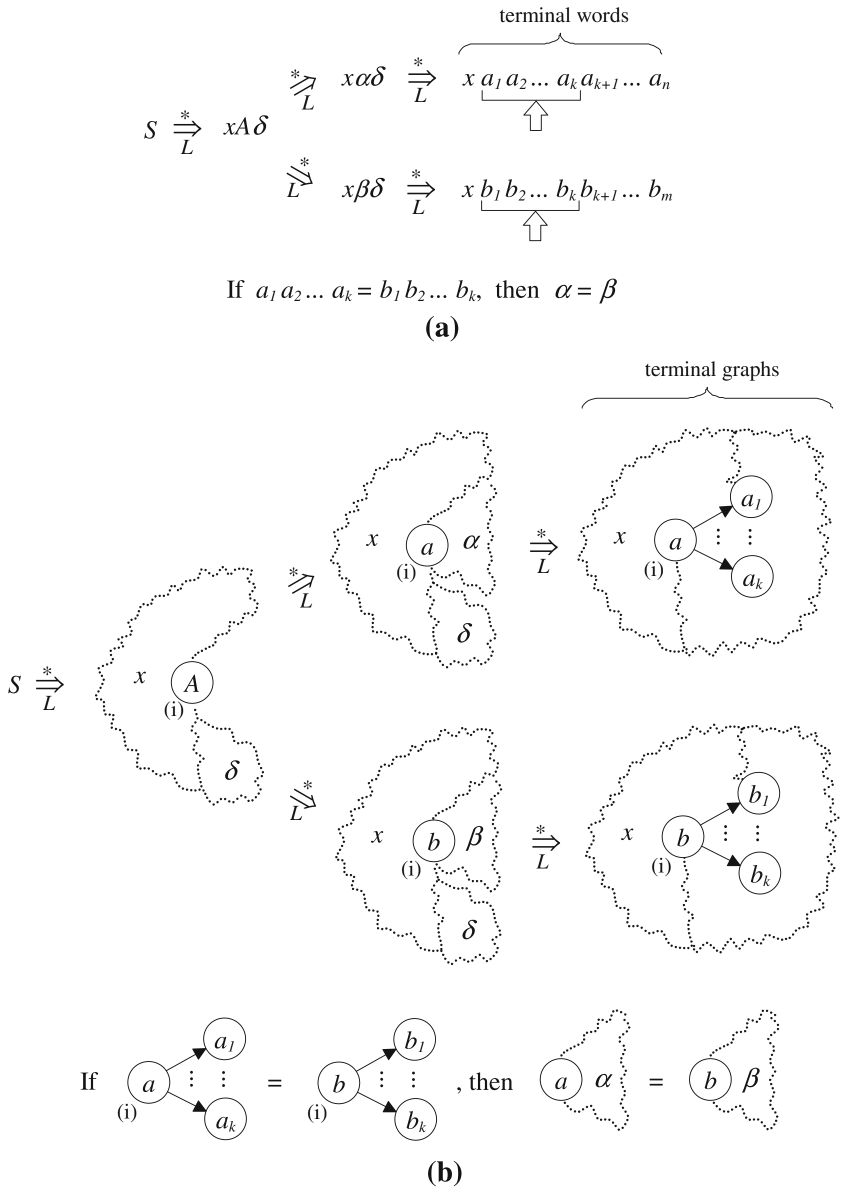
[6] In the survey, we present those parsing models that have been applied in the area of syntactic pattern recognition.

[7] In case of tree structure we have at least the partial ordering.

grammars we know how (or rather *where*) to glue/embed a right hand-side of a production in a structure transformed during a derivational step. It results from a uniform rigid structure of strings. However, in case of graph grammars we have to specify how to embed the right-hand side graph in the rest-graph in an explicit way. Such a specification is made with the help of the third component of a production, i.e. the *embedding transformation*. The embedding transformation allows one to modify a derived graph structure. On the other hand, it acts at the border between the left- (right-) hand sides of the production and their context, i.e. its behaviour is "context sensitive"-like.

These two "features" of graph grammars cause their big generating power. On the other hand, they result in a hard (i.e. PSPACE-complete or NP-complete) membership problem for classes of graph grammars interesting from the application point of view, which has been shown at the beginning of 1980s independently by: Brandenburg [4], Slisenko [51] and Turan [56]. Therefore, in case of a construction of a graph parser, either a generative power of a class of grammars is usually decreased with imposing certain restrictions or one defines a specific class of graph languages with a polynomial membership problem (like e.g. in case of expansive grammars [50]). Summing up, we can define the following methodological recommendation.

V. Constructing a syntactic pattern recognition model based on a class of graph grammar defined, one should focus primarily on a polynomial complexity of a membership problem for languages generated by this class.



terminal words
$S \overset{*}{\underset{L}{\Rightarrow}} xA\delta$
$x\alpha\delta \overset{*}{\underset{L}{\Rightarrow}} x\,a_1 a_2 \dots a_k a_{k+1} \dots a_n$
$x\beta\delta \overset{*}{\underset{L}{\Rightarrow}} x\,b_1 b_2 \dots b_k b_{k+1} \dots b_m$
If $a_1 a_2 \dots a_k = b_1 b_2 \dots b_k$, then $\alpha = \beta$
(a)
terminal graphs
S, x, A, (i), δ, a, α, b, β, a_1, a_k, b_1, b_k
If … = …, then … = …
(b)

Fig. 5 An analogy between restrictions imposed on: **a** LL(k) string grammars and **b** ETPL(k) graph grammars

Now, let us consider the second issue relating to constructing effective syntactic pattern recognition model, i.e. an inference algorithm. In case of graph grammars applied for syntactic pattern recognition, an efficient (polynomial) algorithm has been defined only for the parsable ETPL(k) [17, 18] grammars [21]. As we have mentioned it previously, a grammatical inference problem is much more complex and difficult than a parsing problem. Analyzing both the parsing algorithm and the inference algorithm for ETPL(k) grammars, one could easily see that the model has been constructed with the help of analogies to string grammars. Particularly, deterministic properties of a derivation of ETPL(k) graph languages have been obtained on the analogy of the well-known string deterministic LL(k) grammars [49]. Let us, now, consider this analogy.

A basic idea of LL(k) grammars is shown in Fig. 5a. A condition imposed on CFG causing its deterministic derivation (that results in a polynomial parsing complexity) can be formulated in the following way. For any derivational step, we should be able to choose a production in an unambiguous way on the basis of an analysis of a corresponding piece of this word (i.e. a *handle* mentioned above) that is of the length k. We can call such a property of an LL(k) grammar: *a property of an unambiguous choice of a production with respect to the k-length prefix in a (leftmost) derivation*.

Now, let us look at Fig. 5b illustrating a basic idea of ETPL(k) graph grammars. In this case imposing a condition on edNLC graph grammars causing their deterministic derivation has been the main objective, as well. Thus, we have demanded an unambiguity of a production choice during a (leftmost) derivation. For a string LL(k) grammar such an unambiguity has concerned the k-length prefix of a word (i.e. the k-length handle). In case of graphs it should concern a subgraph of a terminal graph. Such a subgraph contains a node a having an index (i) determining a place of a production application and its k successors: $a_1, \ldots, a_k$. Such a subgraph is called the *k-successors handle*. If for every derivational step in a grammar we can choose a production in an unambiguous way on the basis of an analysis of the k-successors handle, then we say that the grammar has *a property of an unambiguous choice of a production with respect to the k-successors handle in a (leftmost) derivation*.

Similarly, defining a general scheme of an inference of an ETPL(k) grammar, the author has made use of analogies of the well-known scheme of a *formal derivatives method* used for inferencing string grammars [24]. Summing up, looking for analogies in the area of string languages seems to be a good methodological technique while we make a research in the area of graph languages. Thus, let us formulate the following methodological recommendation.

VI. During a process of constructing a graph parser and an algorithm of a graph grammar inference one should look for analogical constructs and mechanisms in an area of the theory of string languages.

5 Conclusions

The theory of formal languages (mathematical linguistics) constitutes a formal basis for various research areas in computer science. A specificity of an area determines methodological principles that should be followed during a research. In other words, for various areas various principles are valid. The general principles concerning an application of mathematical linguistics formalisms in a syntactic pattern recognition area have been formulated in the paper.

These principles can be summarized as follows. A syntactic pattern recognition model should include not only an efficient syntax analyzer, but also a grammatical inference algorithm of a polynomial complexity. If the model is to be used for a recognition of real-world objects or phenomena, then it should be enhanced with such techniques as: error-correcting parsing, attributed grammars or stochastic grammars. In order to ensure a computational efficiency of the model, a type of a grammar should be of the smallest generative power yet sufficient to generate all the possible patterns. For context-free grammars enhanced with the help of additional syntactic operators, a construction of a grammatical inference algorithm is easier than for grammars with a control mechanism “hidden” in productions. A polynomial complexity of a membership problem is a key issue for a graph grammar-based pattern recognition. In order to define algorithms of: syntax analysis and grammatical inference for graph grammar-based methods one should look for analogical algorithms for string grammars.

Let us remember that they are not necessarily valid in other research areas that make use of formalisms of mathematical linguistics such, as e.g. Natural Language Processing, compiler design.

A syntactic pattern recognition paradigm is primarily an approach in a machine recognition area. However, it relates also strongly to other fields of computer science, like: Artificial Intelligence (a problem of pattern understanding, see e.g. [54]), a construction of secret sharing techniques (see e.g. [39]), etc. Therefore, a set of methodological principles presented in the paper will be extended in the future with new ones connected with the AI paradigms.

References

1. Aho AV (1968) Indexed grammars—an extension of context-free grammars. J Assoc Comput Mach 15:647–671
2. Aho AV, Peterson TG (1972) A minimum distance error-correcting parser for context-free languages. SIAM J Comput 1:305–312
3. Alquezar R, Sanfeliu A (1997) Recognition and learning of a class of context-sensitive languages described by augmented regular expressions. Pattern Recogn 30:163–182
4. Brandenburg FJ (1983) On the complexity of the membership problem of graph grammars. Proc. Int. Workshop on Graphtheoretic Concepts in Computer Sci. WG'83, June 16-18 1983, Osnabrück, Germany, Trauner Verlag, pp 40–49
5. Bunke HO, Haller B (1990) A parser for context free plex grammars. Lecture Notes Comput Sci 411:136–150
6. Bunke HO, Sanfeliu A, eds (1990) Syntactic and structural pattern recognition—theory and applications. World Scientific, Singapore
7. Castaño JM (2003) GIGs: Restricted context-sensitive descriptive power in bounded polynomial-time. In: Proceedings of conference on intelligent text processing and computational linguistics (CICLing '03)
8. Dassow J, Păun G (1990) Regulated rewriting in formal language theory. Springer, New York
9. Dassow J (2004) Grammars with regulated rewriting. In: Martin-Vide C, Mitrana V, Păun G (eds) Formal languages and applications. Springer, Berlin
10. Eijck van J (2008) Sequentially indexed grammars. J Logic Comput 18(2):205–228
11. Ferruci F, Tortora G, Tucci M, Vitiello G (1994) A predictive parser for visual languages specified by relational grammars. Proceedings of the IEEE symposium on visual languages-VL'94: 245–252
12. Flasiński M (1988) Parsing of edNLC-graph grammars for scene analysis. Pattern Recogn 21:623–629
13. Flasiński M (1989) Characteristics of edNLC—graph grammars for syntactic pattern recognition. Comput Vision Graphics Image Process. CGIP-47: 1–21
14. Flasiński M (1990) Distorted pattern analysis with the help of Nodel Label Controlled graph languages. Pattern Recogn 23: 765–774
15. Flasiński (1991) Some notes on a problem of constructing the best matched graph. Pattern Recogn 24:1223–1224
16. Flasiński M, Lewicki G (1991) The convergent method of constructing polynomial discriminant functions for pattern recognition. Pattern Recogn 24:1009–1015
17. Flasiński M (1993) On the parsing of deterministic graph languages for syntactic pattern recognition. Pattern Recogn 26:1–16
18. Flasiński M (1998) Power properties of NLC graph grammars with a polynomial membership problem. Theoret Comput Sci 201:189–231
19. Flasiński M, Jurek J (1999) Dynamically programmed automata for quasi context sensitive languages as a tool for inference support in pattern recognition-based real-time control expert systems. Pattern Recogn 32:671–690
20. Flasiński M, Jurek J (2006) On the analysis of fuzzy string patterns with the help of extended and stochastic GDPLL(k) grammar. Fundam Inf 71:1–14
21. Flasiński M (2007) Inference of parsable graph grammars for syntactic pattern recognition. Fundam Inf 80:379–413
22. Flasiński M, Myśliński S (2010) On the use of graph parsing for recognition of isolated hand postures of Polish Sign Language. Pattern Recogn 43:2249–2264
23. Fu KS, Swain PH (1971) Stochastic programmed grammars for syntactic pattern recognition. Pattern Recogn 4:83–100
24. Fu KS (1982) Syntactic pattern recognition and applications. Prentice Hall, Englewood Cliffs
25. Gazdar G (1988) Applicability of indexed grammars to natural languages. In: Reyle U, Rohrer C (eds) Natural language parsing and linguistic theories. D. Reidel Publ. Comp., Englewood Cliffs, NJ
26. Goldfarb L (2004) Pattern representation and the future of pattern recognition. Proceedings of the ICPR-2004 workshop, August 22, 2004, Cambridge
27. Gonzales RC, Thomason MG (1978) Syntactic pattern recognition: an introduction. Addison-Wesley, Reading
28. Harbusch K (2003) An efficient online parser for contextual grammars with at most contextufree selectors. Lecture Notes Comput Sci 2588:168–179
29. Jain AK, Duin RPW, Mao J (2000) Statistical pattern recognition: a review. IEEE Trans Pattern Anal Mach Intell 22:4–37
30. Janssens D, Rozenberg G (1980) On the structure of node-label-controlled graph languages. Inf Sci 20:191–216
31. Joshi AK, Levy LS, Takahashi M (1975) Tree adjunct grammars. J Comput Syst Sci 10:136–163
32. Joshi AK (1985) How much context-sensivity is necessary for characterizing structural descriptions—tree adjoining grammars. In: Dowty D et al. (eds) Natural language processing—theoretical, computational and psychological perspective. Cambridge University Press, New York
33. Jurek J (2000) On the linear computational complexity of the parser for quasi context sensitive languages. Pattern Recognit Lett 21:179–187
34. Jurek J (2004) Towards grammatical inferencing of GDPLL(k) grammars for applications in syntactic pattern recognition-based expert systems. Lecture Notes Comput Sci 3070:604–609
35. Jurek J (2005) Recent developments of the syntactic pattern recognition model based on quasi-context sensitive languages. Pattern Recognit Lett 26:1011–1018
36. Knuth D (1971) Semantics of context-free languages. Math Syst Theory 2:127–145
37. Marcus S (1969) Contextual grammars. Rev Roum Math Pures Appl 14:1525–1534
38. Oates T, Armstrong T, Becerra-Bonache L, Atamas M (2006) Inferring grammars for mildly context sensitive languages in polynomial time. Lecture Notes Comput Sci 4201:137–147
39. Ogiela MR, Ogiela U (2012) DNA-like linguistic secret sharing for strategic information systems. Int J Inf Manag 32:175–181
40. Okhotin A (2001) Conjunctive grammars. J Autom Lang Comb 6:519–535
41. Okhotin A (2004) Boolean grammars. Inf Comput 194:19–48
42. Pavlidis T (1977) Structural pattern recognition. Springer, New York
43. Peng KJ, Yamamoto T, Aoki Y (1990) A new parsing scheme for plex grammars. Pattern Recogn 23:393–402
44. Pfaltz JL, Rosenfeld A (1969) Web grammars. Proceedings of the first international conference on artificical intelligence, Washington DC, pp 609–619
45. Pfaltz JL (1972) Web grammars and picture description. Comput Graph Image Process 1:193–220
46. Pollard C (1984) Generalized phrase structure grammars, head grammars, and natural language. PhD thesis, Stanford University, CA
47. Rosenfeld A, Milgram DL (1972) Web automata and web grammars. Mach Intell 7:307–324

48. Rosenkrantz DJ (1969) Programmed grammars and classes of formal languages. J Assoc Comput Mach 16:107–131
49. Rosenkrantz DJ, Stearns RE (1970) Properties of deterministic top-down grammars. Inf Control 17:226–256
50. Shi QY, Fu KS (1983) Parsing and translation of attributed expansive graph languages for scene analysis. IEEE Trans Pattern Anal Mach Intell 5:472–485
51. Slisenko AO (1982) Context-free grammars as a tool for describing polynomial-time subclasses of hard problems. Inform Proc Lett 14:52–56
52. Staudacher P (1993) New frontiers beyond context-freeness: DI-grammars and DI-automata. In: Proceedings of the sixth conference on European chapter of the association for computational linguistics (EACL '93).
53. Steedman M (1987) Combinatory grammars and parasitic gaps. Nat Lang Ling Theory 5:403–439
54. Tadeusiewicz R, Ogiela MR (2004) Medical image understanding technology. Springer, Berlin-Heidelberg-New York
55. Tanaka E (1995) Theoretical aspects of syntactic pattern recognition. Pattern Recogn 28:1053–1061
56. Turan G (1982) On the complexity of graph grammars. Rep. Automata Theory Research Group. Szeged, 1982
57. Vijay-Shanker K, Weir DJ (1994) The equivalence of four extensions of context-free grammars. Math Syst Theory 27:511–546
58. Wittenburg K (1992) Earley-style parsing for relational grammars. Proceedings of the IEEE symposium on visual languages - VL'92:192–199
59. Zhang DQ, Zhang K, Cao J (2001) A context-sensitive graph grammar formalism for the specification of visual languages. Comput J 44:186–200