Chapter 8
Syntactic Pattern Analysis

In *syntactic pattern analysis*, also called *syntactic pattern recognition* [97, 104], reasoning is performed on the basis of structural representations which describe things and phenomena belonging to the world. A set of such structural representations, called (structural) patterns, constitutes the database of an AI system. This set is not represented in the database explicitly, but with the help of a formal system, which generates all its patterns. A *generative grammar*, introduced in Chap. 1 and Sect. 2.5 during a discussion of the main ideas of Chomsky's theory, is the most popular formal system used for this purpose. The grammar generates structural patterns by the application of *string rewriting rules*,[1] which are called *productions*. Thus, a (string) generative grammar constitutes a specific type of *Abstract Rewriting System, ARS*, introduced in Sect. 6.5, which is called a *String Rewriting System, SRS*. Therefore, reasoning by syntactic pattern analysis can be treated as *reasoning by symbolic computation*, which has been discussed in Sect. 6.5.

Generating structural patterns with the help of a generative grammar is introduced in the first section of this chapter. In Sects. 8.2 and 8.3 the analysis and the interpretation of structural patterns are discussed, respectively. The problem of automatic construction of a grammar on the basis of sample patterns is considered in the fourth section. In the last section graph grammars are introduced. Formal definitions of notions introduced in this chapter are contained in Appendix E.

[1] Such structural patterns can be of the form of strings or graphs. Therefore, two types of generative grammars are considered: string grammars and graph grammars. In syntactic pattern recognition tree grammars, which generate tree structures, are also defined. Since a tree is a particular case of a graph, we do not introduce tree grammars in the monograph. The reader is referred, e.g., to [104].

M. Flasiński, *Introduction to Artificial Intelligence*,
DOI 10.1007/978-3-319-40022-8_8

8.1 Generation of Structural Patterns

We begin our definition of a *generative grammar* in the Chomsky model [47] by introducing a *set of terminal symbols*. Terminal symbols are expressions which are used for constructing sentences belonging to a language generated by a grammar.[2] Let us assume that we construct a grammar for a subset of the English language consisting of sentences which contain four subjects (male names): Hector, Victor, Hyacinthus, Pacificus, two predicates: accepts, rejects, and four objects: globalism, conservatism, anarchism, pacifism. For example, the sentences Hector accepts globalism and Hyacinthus rejects conservatism belong to the language.[3] Let us denote this language by L_1. Then, a set of terminal symbols T_1 is defined as follows:

$T_1 = \{$Hector, Victor, Hyacinthus, Pacificus, accepts, rejects, globalism, conservatism, anarchism, pacifism$\}$.

As we have mentioned above, sentences are generated with rewriting rules called *productions*.[4] Let us consider an application of a production with the following example. Let there be given a phrase:

$$\text{Hector accepts } B\ , \tag{8.1}$$

where B is an auxiliary symbol, called a *nonterminal symbol*,[5] which denotes an object in a sentence. Let there be given a production of the form:

$$B \rightarrow \text{globalism}\ . \tag{8.2}$$

An expression placed on the left-hand side of an arrow is called the *left-hand side of a production* and an expression placed on the right-hand side of an arrow is called the *right-hand side of a production*. An application of the production to the phrase, denoted by $\Longrightarrow$, consists of replacing an expression of the phrase which is equivalent to the left-hand side of the production (in our case it is the symbol B) by the right-hand side of the production. Thus, an application of the production is denoted in the

[2] Let us remember that the notions of *word* and *sentence* are treated symbolically in formal language theory. For example, if we define a grammar which generates single words of the English language, then letters are *terminal symbols*. Then, English words which consist of these letters are called *words (sentences)* of a formal language. However, if we define a grammar which generates sentences of the English language, then English words can be treated as *terminal symbols*. Then sentences of the English language are called *words (sentences)* of a formal language. As we see later on, *words (sentences)* of a formal language generated by a grammar can represent any string structures, e.g., stock charts, charts in medicine (ECG, EEG), etc. Since in this section we use examples from a natural language, strings consisting of symbols are called sentences.

[3] We omit the terminal symbol of a full stop in all examples in order to simplify our considerations. Of course, if we use generative grammars for Natural Language Processing (NLP), we should use a full stop symbol.

[4] We call them *productions*, because they are used for generating—"producing"—sentences of a language.

[5] Nonterminal symbols are usually denoted by capital letters.

following way:

$$\text{Hector accepts } B \Longrightarrow \text{Hector accepts globalism}. \tag{8.3}$$

Now, we define a set of all productions which generate our language. Let us denote this set by P_1:

(1) $S \rightarrow$ Hector A
(2) $S \rightarrow$ Victor A
(3) $S \rightarrow$ Hyacinthus A
(4) $S \rightarrow$ Pacificus A
(5) $A \rightarrow$ accepts B
(6) $A \rightarrow$ rejects B
(7) $B \rightarrow$ globalism
(8) $B \rightarrow$ conservatism
(9) $B \rightarrow$ anarchism
(10) $B \rightarrow$ pacifism .

For example, the sentence Pacificus rejects globalism is generated as follows:

$$S \overset{4}{\Longrightarrow} \text{Pacificus } A \overset{6}{\Longrightarrow} \text{Pacificus rejects } B \overset{7}{\Longrightarrow} \text{Pacificus rejects globalism}. \tag{8.4}$$

Indices of the productions applied are placed above double arrows. A sequence of production applications used for generating a sentence is called a *derivation* of this sentence. If we are not interested in presenting the sequence of *derivational steps*, then we can simply write:

$$S \overset{*}{\Longrightarrow} \text{Pacificus rejects globalism}, \tag{8.5}$$

which means that we can generate (derive) a sentence Pacificus rejects globalism with the help of grammar productions starting with a symbol S.

As we can see our *set of nonterminal symbols*, which we denote by N_1, consists of three auxiliary symbols S, A, B, which are responsible for generating a subject, a predicate, and an object, i.e.,

$$N_1 = \{S, A, B\}.$$

In a generative grammar a nonterminal symbol which is used for starting any derivation is called *the start symbol (axiom)* and it is denoted by S. Thus, our grammar G_1 can be defined as a quadruple $G_1 = (T_1, N_1, P_1, S)$. A language generated by a grammar G_1 is denoted $L(G_1)$. The language $L(G_1)$ is the set of all the sentences which can be derived with the help of productions of the grammar G_1. It can be proved that for our language L_1, which has been introduced in an informal way at the beginning of this section, the following holds: $L(G_1) = L_1$.

A generative grammar has an interesting property: it is a finite "object" (it consists of finite sets of terminal and nonterminal symbols and a finite set of productions), however it can generate an infinite language, i.e., a language which consists of an

infinite number of sentences. Before we consider this property, let us introduce the following denotations. Let a be a symbol. By a^n we denote an expression which consists of an n-element sequence of symbols a, i.e.,

$$a^n = \underbrace{aaa \ldots aaa}_{n \text{ times}} , \tag{8.6}$$

where $n \geq 0$. If $n = 0$, then it is a 0-element sequence of symbols a, called the *empty word* and denoted by λ. Thus: $a^0 = \lambda, a^1 = a, a^2 = aa, a^3 = aaa$, etc.

For example, let us define a language L_2 in the following way:

$$L_2 = \{a^n \ , \ n \geq 1\} . \tag{8.7}$$

The language L_2 is infinite and consists of n-element sequences of symbols a, where additionally a has to occur at least once. (The empty word does not belong to L_2.) It can be generated by the following simple grammar G_2:

$$G_2 = (T_2, N_2, P_2, S) ,$$

where $T_2 = \{a\}$, $N_2 = \{S\}$, and the set of productions P_2 contains the following productions:

$$(1) \ S \rightarrow aS$$
$$(2) \ S \rightarrow a .$$

A derivation of a sentence of a given length $r \geq 2$ (i.e., $n = r$) in G_2 is performed according to the following scheme:

$$S \stackrel{1}{\Longrightarrow} aS \stackrel{1}{\Longrightarrow} aaS \stackrel{1}{\Longrightarrow} \ldots \stackrel{1}{\Longrightarrow} a^{r-1}S \stackrel{2}{\Longrightarrow} a^r , \tag{8.8}$$

i.e., firstly the first production is applied $(r - 1)$ times, then the second production is applied once at the end. If we want to generate a sentence of a length $n = 1$, i.e., the sentence a, the we apply the second production once.

Let us notice that defining a language L_2 as an infinite set is possible due to the first production. This production is of an interesting form: $S \rightarrow aS$, which means that it *refers to itself* (a symbol S occurs on the left- and right-hand sides of the production). Such a form is called *recursive*, from Latin *recurrere*—"running back". This "running back" of the symbol S during a derivation each time after applying the first production, makes the language L_2 infinite.

Now, we discuss a very important issue concerning generative grammars. There are a lot of classes (types) of generative grammars. These classes can be arranged in a hierarchy according to the criterion of their *generative power*. We introduce this criterion with the help of the following example. Let us define the next formal language as follows:

$$L_3 = \{a^n b^m \ , \ n \geq 1 \ , \ m \geq 2\} . \tag{8.9}$$

The language L_3 consists of the subsequence of symbols a and the subsequence of symbols b. Additionally, the symbol a has to occur at least once, and the symbol b has to occur at least twice. For example, the sentences abb, $aabb$, $aabbb$, $aabbbb$, ... , $aaabb$, etc. belong to this language. It can be generated by the following grammar:

$$G_3 = (T_3, N_3, P_3, S) ,$$

where $T_3 = \{a , b\}$, $N_3 = \{S , A , B\}$, and the set of productions P_3 contains the following productions:

$$\begin{array}{l} (1)\ S \rightarrow aA \\ (2)\ A \rightarrow aA \\ (3)\ A \rightarrow bB \\ (4)\ B \rightarrow bB \\ (5)\ B \rightarrow b . \end{array}$$

The first production is used for generating the first symbol a. The second production generates successive symbols a in a recursive way. We generate the first symbol b with the help of the third production. The fourth production generates successive symbols b in a recursive way (analogously to the second production). The fifth production is used for generating the last symbol b of the sentence. For example, a derivation of the sentence a^3b^4 is performed as follows:

$$\begin{aligned} S \overset{1}{\Longrightarrow} aA \overset{2}{\Longrightarrow} aaA \overset{2}{\Longrightarrow} aaaA \overset{3}{\Longrightarrow} aaabB \overset{4}{\Longrightarrow} \\ \overset{4}{\Longrightarrow} aaabbB \overset{4}{\Longrightarrow} aaabbbB \overset{5}{\Longrightarrow} aaabbbb . \end{aligned} \tag{8.10}$$

Now, let us define a language L_4 which is a modification of the language L_3, in the following way:

$$L_4 = \{a^m b^m , m \geq 1\} . \tag{8.11}$$

The language L_4 differs from the language L_3 in demanding an equal number of symbols a and b. The language L_4 cannot be generated with the help of grammars having productions of the form of grammar G_3, since such productions do not ensure the condition of an equal number of symbols. It results from the fact that in such a grammar we firstly generate a certain number of symbols a and then we start to generate symbols b, but the grammar "does not remember"how many symbols a have been generated. We say that a grammar having productions in the a form of the productions of G_3 has *too weak generative power* to generate the language L_4. Now, we introduce a grammar G_4 which is able to generate the language L_4:

$$G_4 = (T_4, N_4, P_4, S) ,$$

where $T_4 = \{a , b\}$, $N_4 = \{S\}$, and the set of productions P_4 contains the following productions:

$$(1)\ S \rightarrow aSb$$
$$(2)\ S \rightarrow ab\ .$$

For example, a derivation of the sentence a^4b^4 is performed in the following way:

$$S \overset{1}{\Longrightarrow} aSb \overset{1}{\Longrightarrow} aaSbb \overset{1}{\Longrightarrow} aaaSbbb \overset{2}{\Longrightarrow} aaaabbbb\ . \tag{8.12}$$

As we can see a solution to the problem of an equal number of symbols a and b is obtained by generating the same number of both symbols in each derivational step.

Thus, the grammar G_4 has *sufficient generative power* to generate the language L_4. The generative power of classes of formal grammars results from the form of their productions. Let us notice that the productions of grammars G_1, G_2, G_3 are of the following two forms:

$$<\textit{nonterminal symbol}> \rightarrow <\textit{terminal symbol}><\textit{nonterminal symbol}>$$
$$\text{or} \quad <\textit{nonterminal symbol}> \rightarrow <\textit{terminal symbol}>\ .$$

Using productions of such forms, we can only "stick" a symbol onto the end of the phrase which has been derived till now. Grammars having only productions of such a form are called *regular grammars*.[6] In the Chomsky hierarchy such grammars have the weakest generative power.

For grammars such as the grammar G_4, we do not demand any specific form of the right-hand side of productions. We require only a single nonterminal symbol at the left-hand side of a production. Such grammars are called *context-free grammars*.[7] They have greater generative power than regular grammars. However, we have to pay a certain price for increasing the generative power of grammars. We discuss this issue in the next section.

8.2 Analysis of Structural Patterns

Grammars are used for generating languages. However, in Artificial Intelligence we are interested more in the languages' analysis. For this analysis formal automata are applied. Various types of automata differ from one another in their construction (structure), depending on the corresponding classes of grammars. Let us begin by defining an automaton of the simplest type, i.e., a *finite-state automaton*. This class

[6] In fact, such grammars are *right regular grammars*. In *left regular grammars* a nonterminal symbol occurs (if it occurs) before a terminal symbol.

[7] There are also grammars which have a stronger generative power in the Chomsky hierarchy, namely *context-sensitive grammars* and *unrestricted (type-0) grammars*. Their definitions are contained in Appendix E.

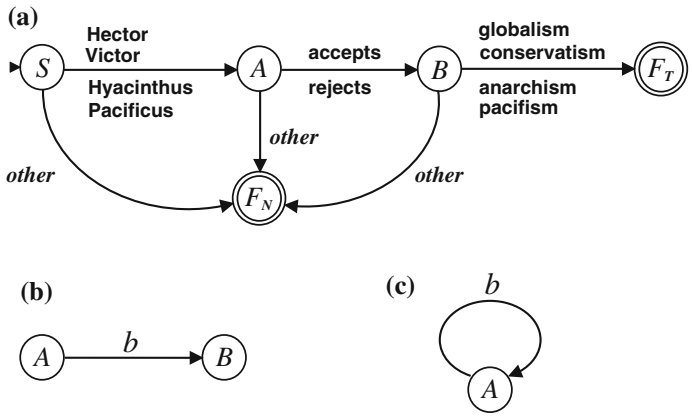

Fig. 8.1 **a** The finite-state automaton A_1, **b-c** basic constructs for defining a finite-state automaton

was introduced by Claude Elwood Shannon[8] in [272][9] in 1948, and formalized by Stephen C. Kleene in [160] in 1956 and Michael Oser Rabin[10] and Dana Stewart Scott[11] in [234] (nondeterministic automata) in 1959. A finite-state automaton is used for analysis of languages generated by regular grammars, called *regular languages*.[12]

Let us start by defining the automaton A_1 shown in Fig. 8.1a, which is constructed for the language L_1 (the grammar G_1) introduced in the previous section. Each node of the graph is labeled with a symbol (S, A, B, F_T, F_N) and represents a possible *state* of the automaton. The state in which the automaton starts working is called the *initial state* S and is marked with a small black triangle in Fig. 8.1a. States in which the automaton finishes working (F_T and F_N in Fig. 8.1a) are called *final states* and they are marked with a double border. Directed edges of the graph define *transitions* between states. A transition from one state to another takes place if the automaton

[8]Claude Elwood Shannon—a professor of the Massachusetts Institute of Technology, a mathematician and electronic engineer, the "father" of information theory and computer science.

[9]The idea of a finite-state automaton is based on the model of Markov chain which is introduced in Appendix B for genetic algorithms.

[10]Michael Oser Rabin—a professor of Harvard University and the Hebrew University of Jerusalem, a Ph.D. student of Alonzo Church. His outstanding achievements concern automata theory, computational complexity theory, cryptography (Miller-Rabin test), and pattern recognition (Rabin-Karp algorithm). In 1976 he was awarded the Turing Award (together with Dana Scott).

[11]Dana Stewart Scott—a professor of computer science, philosophy, and mathematics at Carnegie Mellon University and Oxford University, a Ph.D. student of Alonzo Church. His excellent work concerns automata theory, semantics of programming languages, modal logic, and model theory (a proof of the independence of the continuum hypothesis). In 1976 he was awarded the Turing Award.

[12]The languages L_1, L_2, and L_3 introduced in the previous section are regular languages.

reads from its input[13] an element determining this transition. For example, let us assume that the automaton is in the state S. If the automaton reads from the input one of the elements Hector, Victor, Hyacinthus, or Pacificus, then it goes to the state A. Otherwise, it goes according to the transition *other*[14] to the final state F_N, which means that the input expression is rejected as not belonging to the language L_1. If the automaton is in the state A, then it goes to the state B in case there is one of the predicates accepts or rejects at the input. This is consistent with the definition of the language L_1, in which the predicate should occur after one of the four subjects. If the automaton is in the state B, in turn, it expects one of four objects: globalism, conservatism, anarchism, or pacifism. After reading such an object the automaton goes to the final state F_T, which means that the input expression is accepted as belonging to the language L_1.

Formally, a finite-state automaton A constructed for a language L generated by a regular grammar $G = (T, N, P, S)$ is defined as a quintuple: $G = (Q, T, \delta, q_0, F)$. T is the set of terminal symbols which are used by the grammar G for generating the language L. Q is the set of states. (In our example shown in Fig. 8.1a it is the set $\{S, A, B, F_T, F_N\}$.) q_0 is the initial state, F is the set of final states. (In our example $q_0 = S$, F consists of states F_T and F_N.) δ is the *state-transition function*,[15] which determines transitions in the automaton (in Fig. 8.1a transitions are represented by directed edges of the graph). A pair (*the state of the automaton*, *the terminal symbol at the input*) is an argument of the function. The function *computes* the state the automaton should go into. For example, $\delta(S, \text{Hyacinthus}) = A, \delta(A, \text{accepts}) = B$ (cf. Fig. 8.1a).

A method for a generation (synthesis) of a finite-state automaton on the basis of a corresponding regular grammar has been developed. States of the automaton relate to nonterminal symbols of the grammar (the initial state relates to the start symbol, additionally we can define final states). Each production of the form $A \rightarrow bB$ is represented by a transition $\delta(A, b) = B$ (cf. Fig. 8.1b). Each recursive production of the form $A \rightarrow bA$ is represented by a recursive transition $\delta(A, b) = A$ (cf. Fig. 8.1c). Each production finishing a derivation (there is a single terminal symbol on the right-hand side of the production) corresponds to a transition to the final acceptance state. The reader can easily see that the automaton A_1 shown in Fig. 8.1a has been constructed on the basis of the grammar G_1 according to these rules. In the previous section we have said that generative grammars are arranged in a hierarchy according to their generative power. The same applies to automata. What is more, each class of grammar relates to some type of automaton. Automata which correspond to weaker grammars (in the sense of generative power) are not able to analyze languages generated by stronger classes of grammars. For example, a finite-state automaton is

[13] The *input* of the automaton is the place where the expression to be analyzed is placed. If there is some expression at the *input*, then the automaton reads the expression one element (a terminal symbol) at a time and it performs the proper transitions.

[14] The *other* transition means that the automaton has read an element which is different from those denoting transitions coming out from the current state.

[15] The state-transition function is not necessarily a function in the mathematical sense of this notion.

too weak to analyze the *context-free language* $L_4 = \{a^m b^m \ , \ m \geq 1\}$ introduced in the previous section.

A *(deterministic) pushdown automaton*[16] is strong enough to analyze languages such as L_4. This automaton uses an additional working memory, called a *stack*.[17] The transition function δ of such an automaton is defined in a different way from the finite-state automaton. A pair (*the top of the stack*, *the sequence of symbols at the input*[18]) is its argument. As a result, the function can "generate" various actions of the automaton. In the case of our automaton the following actions are allowed:

- accept (the automaton has analyzed the complete expression at the input and it has decided that the expression belongs to the language),
- reject (the automaton has decided, during its working, that the expression does not belong to the language),
- remove_symbol (the automaton removes a terminal symbol from the input and a symbol occurring at the top of the stack),
- apply_production_on_stack(i) (the automaton takes the left-hand side of a production i from the top of the stack and it adds the right-hand side of the production i to the top of the stack).

Before we consider the working of an automaton A_4 constructed for the language $L_4 = \{a^m b^m \ , \ m \geq 1\}$, let us define its transition function in the following way:

$$\delta(S, aa) = \text{apply\_production\_on\_stack(1)}, \tag{8.13}$$
$$\delta(S, ab) = \text{apply\_production\_on\_stack(2)}, \tag{8.14}$$
$$\delta(a, a) = \text{remove\_symbol}, \tag{8.15}$$
$$\delta(b, b) = \text{remove\_symbol}, \tag{8.16}$$
$$\delta(\lambda, \lambda) = \text{accept}, \quad \lambda - \text{the empty word}, \tag{8.17}$$
$$\delta(v, w) = \text{reject}, \ \text{otherwise}. \tag{8.18}$$

The automaton A_4 tries to reconstruct a derivation of the expression which is at its input. It does this by analyzing a sequence consisting of *two* symbols[19] of the expression, because this is sufficient to decide which production of the grammar G_4 (the first or the second) is to be used at the moment of generating this sequence.

[16] In order to simplify our considerations we introduce here a specific case of a pushdown automaton, i.e. an *LL(k) automaton*, which analyzes languages generated by *LL(k) context-free grammars*. These grammars are defined formally in Appendix E.

[17] In computer science a *stack* is a specific structure of a data memory with certain operations, which works in the following way. Data elements can be added only to the top of the stack and they can be taken off only from the top. A stack of books put one on another is a good example of a stack. If we want to add a new book to the stack, we have to put it on the top of a stack. If we want to get some book, then we have to take off all books which are above the book we are interested in.

[18] This *sequence of symbols* has a fixed length. The length of the sequence is a parameter of the automaton. In the case of *LL(k)* automata, k is the length of the sequence, which is analyzed in a single working step of the automaton.

[19] The automaton A_4 is an *LL(2)* automaton.

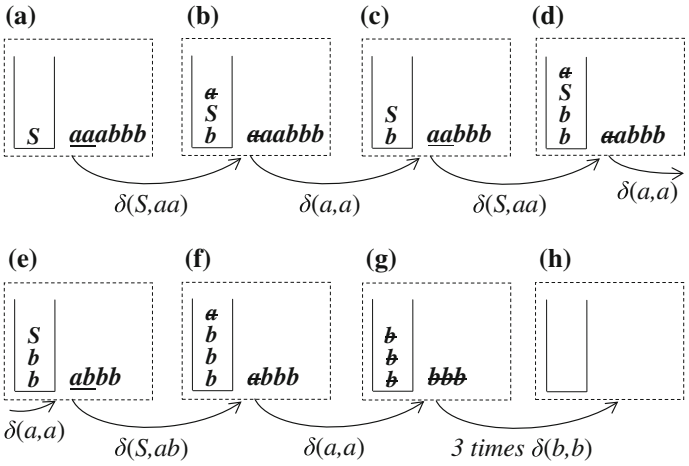

Fig. 8.2 Analysis of the sentence *aaabbb* by the automaton A_4

To be convinced that this is a proper analysis method, let us return to the derivation (8.12) in a previous section. If we want to decide how many times the first production has been applied for the generation of the sentence *aaaabbbb*, then it is enough to check two symbols forward. If we scan a sentence from left to right, then as long as we have a two-element sequence *aa* we know that production (1) has been applied, which corresponds to transition (8.13). If we meet a sequence *ab* (in the middle of the sentence), then it means that production (2) has been applied, which corresponds to transition (8.14). Now, let us analyze the sentence *aaabbb*. The automaton starts working having the start symbol *S* on the stack and the sentence *aaabbb* at the input as shown in Fig. 8.2a. The underlined part of the sentence means the sequence which is analyzed in a given step. Arrows mean transitions and are labeled with the transition function used in a given step. So, there is *S* at the top of the stack and *aa* are the first two symbols of the input. Thus, the first step is performed by the automaton according to transition (8.13). This means that the left-hand side of production (1), i.e., *S*, is taken off the stack and the right-hand side of this production, i.e., *aSb*, is put on the top of the stack[20] as shown in Fig. 8.2b. Now, *a* is at the top of the stack and *a* is at the input. Thus, the next step is performed according to transition (8.15), i.e., the symbol *a* is removed from the top of the stack and from the input. This is denoted by crossing out both symbols. We obtain the situation shown in Fig. 8.2c. This situation is analogous to the one before the first step. (*S* is at the top of the stack, *aa* is at the input.) Thus, we perform a transition according to (8.13) once more, i.e., *S* is taken off the stack and the right-hand side of this production, *aSb*, is put on the top of the stack. This results in the situation shown in Fig. 8.2d. Again *a* is at

[20]The right-hand side of the production, *aSb*, is put on the stack "from back to front", i.e., firstly (at the bottom of the stack) symbol *b* is put, then symbol *S*, then finally (at the top of the stack) symbol *a*.

the top of the stack and a is at the input. So, we perform the transition according to (8.15) and we get the configuration shown in Fig. 8.2e. Since S is on the top of the stack and ab are the first two symbols at the input, we should apply (8.14), which corresponds to production (2) of the grammar G_4. The automaton replaces S on the stack by the right-hand side of the second production, i.e., ab (cf. Fig. 8.2f). As we can see, the next steps consist of removing symbols from the stack according to formula (8.15) and three times formula (8.16). At the end both the stack and the input are empty, as shown in Fig. 8.2h. This corresponds to a transition according to (8.17), which means acceptance of the sentence as belonging to the language L_4. Any other final configuration of the stack and the input would result in rejecting the sentence according to formula (8.18).

One can easily notice that the working of a pushdown automaton is more complex than that of a finite-state automaton. In fact, the bigger the generative power of a generative grammar is, the bigger the computational complexity of the corresponding automaton. The analysis of regular languages is more efficient than the analysis of context-free languages. Therefore, subclasses of context-free grammars with efficient corresponding automata have been defined. The most popular efficient subclasses include: *LL(k) grammars* introduced by Philip M. Lewis[21] and Richard E. Stearns[22] in [180] in 1968, *LR(k) grammars* defined by Donald E. Knuth[23] in [163] in 1965, and *operator precedence grammars* defined by Robert W. Floyd[24] in [98] in 1963. For these types of grammars corresponding efficient automata have been defined.

The problem of syntax analysis (analyzing by automaton) becomes much more difficult if context-free grammars have too weak generative power for a certain application. As we have mentioned above, in the Chomsky hierarchy there are two remaining classes of grammars, namely *context-sensitive grammars* and *unrestricted (type-0) grammars*. *A linear bounded automaton* and *the Turing machine* are two types of

[21] Philip M. Lewis—a professor of electronic engineering and computer science at the Massachusetts Institute of Technology and the State University of New York, a scientist at General Electric Research and Development Center. His work concerns automata theory, concurrency theory, distributed systems, and compiler design.

[22] Richard Edwin Stearns—a professor of mathematics and computer science at the State University of New York, a scientist at General Electric. He was awarded the Turing Award in 1993. He has contributed to the foundations of computational complexity theory (with Juris Hartmanis). His achievements concern the theory of algorithms, automata theory, and game theory.

[23] Donald Ervin Knuth—a professor of computer science at Stanford University. The "father" of the analysis of algorithms. He is known as the author of the best-seller "The Art of Computer Programming" and the designer of the Tex computer typesetting system. Professor D. Knuth is also known for his good sense of humor (e.g., his famous statement: "Beware of bugs in the above code; I have only proved it correct, not tried it."). He was awarded the Turing Award in 1974.

[24] Robert W. Floyd—a computer scientist, physicist, and BA in liberal arts. He was 33 when he became a full professor at Stanford University (without a Ph.D. degree). His work concerns automata theory, semantics of programming languages, formal program verification, and graph theory (Floyd-Warshall algorithm).

automata which correspond to these classes of grammars, respectively. Both types of automata are inefficient computationally, so they cannot be used effectively in practical applications. Therefore, *enhanced* context-free grammars have been defined in order to solve this problem. Such grammars include *programmed grammars* defined by Daniel J. Rosenkrantz[25] in [248] in 1969, *indexed grammars* introduced by Alfred Vaino Aho[26] in [1] in 1968, and *dynamically programmed grammars* published in [95] in 1999.

8.3 Interpretation of Structural Patterns

In the previous section we have shown how to use an automaton for checking whether an expression (sentence) belongs to a language generated by a grammar. In other words, an automaton has been used to test whether an expression is built properly from the point of view of a language's syntax, which is important, e.g., in Natural Language Processing. Generally, in Artificial Intelligence we are interested not only in the syntactical correctness of expressions, but also we are interested in their semantic aspect, i.e., we want to perform a proper interpretation of expressions.[27] Let us consider once more our example of Hector, Victor, et al. introduced in Sect. 8.1. Let us assume that Hector and Victor accept globalism and conservatism, and they reject anarchism and pacifism. On the other hand, Hyacinthus and Pacificus accept anarchism and pacifism, and they reject globalism and conservatism. Let us assume that only such propositions belong to a new language L_5. Now, we can define a grammar G_5 which not only generates sentences which are correct syntactically, but also these propositions are consistent with the assumptions presented above. (That is, these propositions are true.) The set of productions P_5 of the grammar G_5 is defined as follows:

[25] Daniel J. Rosenkrantz—a professor of the State University of New York, a scientist at General Electric, the Editor-in-Chief of the prestigious *Journal of the ACM*. His achievements concern compiler design and the theory of algorithms.

[26] Alfred Vaino Aho—a physicist, an electronic engineer, and an eminent computer scientist, a professor of Columbia University and a scientist at Bell Labs. His work concerns compiler design, and the theory of algorithms. He is known as the author of the excellent books (written with J.D. Ullman and J.E. Hopcroft) *Data Structures and Algorithms* and *The Theory of Parsing, Translation, and Compiling*.

[27] Similarly to the logic-based methods discussed in Sect. 6.1.

(1) $S \rightarrow$ Hector A_1
(2) $S \rightarrow$ Victor A_1
(3) $S \rightarrow$ Hyacinthus A_2
(4) $S \rightarrow$ Pacificus A_2
(5) $A_1 \rightarrow$ accepts B_1
(6) $A_1 \rightarrow$ rejects B_2
(7) $A_2 \rightarrow$ rejects B_1
(8) $A_2 \rightarrow$ accepts B_2
(9) $B_1 \rightarrow$ globalism
(10) $B_1 \rightarrow$ conservatism
(11) $B_2 \rightarrow$ anarchism
(12) $B_2 \rightarrow$ pacifism .

One can easily check that with the help of the set of productions P_5 we can generate all the valid propositions of our model of the world. On the other hand, it is impossible to generate a false proposition, e.g., Hector rejects globalism, although this sentence is correct syntactically.

Now, for the grammar G_5 we can define a finite-state automaton A_5. This automaton is shown in Fig. 8.3a. (For a simplicity we have not defined the final rejection state F_N and transitions to this state.) Let us notice that the automaton A_5 not only checks the syntactical correctness of a sentence, but it also interprets these sentences and accepts only those sentences which are valid in our model of the world.

The automata, which have been introduced till now are called *acceptors (recognizers)*. They accept (a state F_T) or do not accept (a state F_N) a sentence depending on a specific criterion such as syntax correctness or validity (truthfulness) in some model. *Transducers* are the second group of automata. During an analysis they generate expressions on their *outputs*.[28] For example, they can be used for translating expressions of a certain language into expressions of another language. The transition function of such an automaton determines a goal state and writes some expression into the output. For example, let us define a transducer A_6, which translates language L_5 into Polish. We define the transition function as follows: $\delta(A_1,$ accepts$) = (B_1,$ akceptuje$)$, $\delta(A_1,$ rejects$) = (B_2,$ odrzuca$)$, etc. The transducer A_6 is shown in Fig. 8.3b.

Although *Natural Language Processing, NLP*, is one the most important application areas of transducers, there are also other areas in which they are used, i.e., interpretation of the world by automata is not limited to the case of describing the world with the help of natural languages. Certain phenomena are described with the help of other representations (e.g., charts) which express their essence in a better way. Then, in syntactic pattern recognition we can ascribe a certain interpretation to terminal symbols, as shown, for example, in Fig. 8.3c. Graphical elements represented by terminal symbols (in our example: "straight arrows" and "arc arrows") are called *primitives*. Primitives play the role of elementary components used for defining charts.

[28] Therefore, transducers are called also *automata with output*.

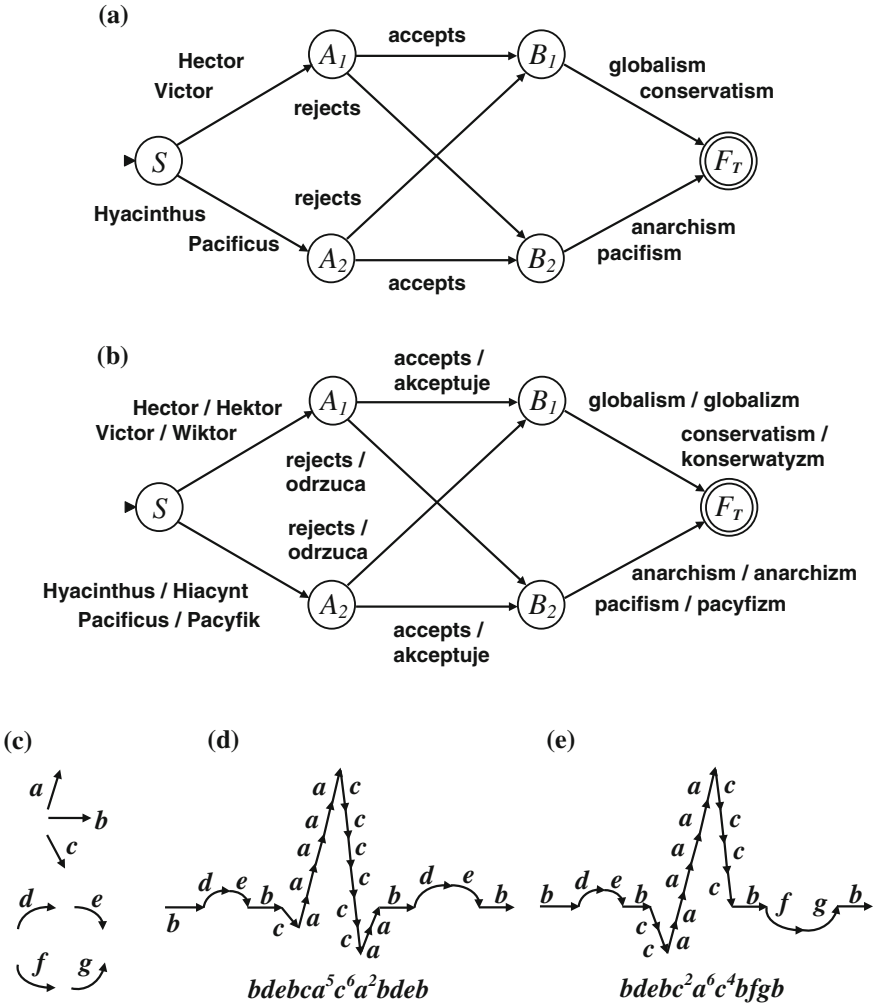

Fig. 8.3 **a** An automaton A_5 which accepts propositions that are valid in the model defined by the language L_5, **b** a transducer A_6 which translates the language L_5 into Polish, **c** an example of structural primitives, **d**-**e** structural representations of ECG patterns

For example, in medical diagnosis we use ECG charts in order to identify heart diseases. An example of a structural representation of a normal ECG is shown in Fig. 8.3d, whereas the case of a myocardial infarction is shown in Fig. 8.3e. These representations can be treated as *sentences* defined in the language of ECG patterns. Thus, on the basis of a set of such representations (sentences) we can define a grammar which generates this language. Given a grammar we can construct an automaton (transducer), which writes an interpretation of an ECG to its output.

Even if we look at an ECG casually, we notice that the primitives occurring in charts are diversified with respect to, e.g., their length or the angle of a depression. Therefore, in order to achieve a more precise structural representation, attributes can be ascribed to primitives. For example, two attributes, the length (l) and the deflection angle ($\pm\alpha$), are ascribed to the primitive a shown in Fig. 8.4a. In such a case *attributed grammars* are used for pattern generation. Automata applied to the interpretation of attributed patterns (expressions) additionally compute the *distance* between an analyzed pattern and a model pattern. This distance allows us to assess the degree of confidence of the interpretation made by the automaton.

Instead of ascribing attributes to a primitive, we can define discrete patterns of deviations of a model primitive as shown in Fig. 8.4b. Then, we can ascribe probabilities to deviations, e.g., on the basis of the frequency of their occurrence. One such model is *stochastic grammars* introduced in the 1970s and the 1980s and then developed by King-Sun Fu[29] and Taylor L. Booth[30] [34,103,104]. In such grammars the probability of the application of each production is defined. A *stochastic automaton* gives the probability that a chart represents a recognized phenomenon expressed by a corresponding structural pattern after analyzing a part of the chart. Stochastic grammars and automata are also used in Natural Language Processing, which is discussed in Chap. 16. It is interesting that *Markov chains*,[31] which have been introduced for genetic algorithms in Chap. 5 are also a mathematical model for a stochastic automaton.

In approaches to the distortion of structural patterns discussed till now we have assumed that the structure of such representations is correct. In other words, a primitive could be distorted but it has to occur in a structure in the proper place. This means that if structural representations are hand-written sentences of a natural language then vaguely hand-written letters are the only kind of errors. However, in practice we can omit some letter (e.g., if we write "gramar"), we can incorrectly add some letter (e.g., if we write "grammuar"), or we can replace a correct letter by an incorrect one (e.g., if we write "glammar"). Fortunately, in syntactic pattern recognition certain metrics are defined which can be used to compute the distance between a model pattern and its *structural* distortion. The *Levenshtein metrics*[32] [179] are some of the most popular metrics used for this purpose. They are introduced in Appendix G.

---

[29] King-Sun Fu—a professor of electrical engineering and computer science at Purdue University, Stanford University and University of California, Berkeley. The "father" of syntactic pattern recognition, the first president of the International Association for Pattern Recognition (IAPR), and the author of excellent monographs, including *Syntactic Pattern Recognition and Applications*, Prentice-Hall 1982. After his untimely death in 1985 IAPR established the biennial King-Sun Fu Prize for a contribution to pattern recognition.

[30] Taylor L. Booth—a professor of mathematics and computer science at the University of Connecticut. His research concerns Markov chains, formal language theory, and undecidability. A founder and the first President of the Computing Sciences Accreditation Board (CSAB).

[31] Markov chains are defined formally in Appendix B.2.

[32] Vladimir Iosifovich Levenshtein—a professor of computer science and mathematics at the Keldysh Institute of Applied Mathematics in Moscow and the Steklov Mathematical Institute. In 2006 he was awarded the IEEE Richard W. Hamming Medal.

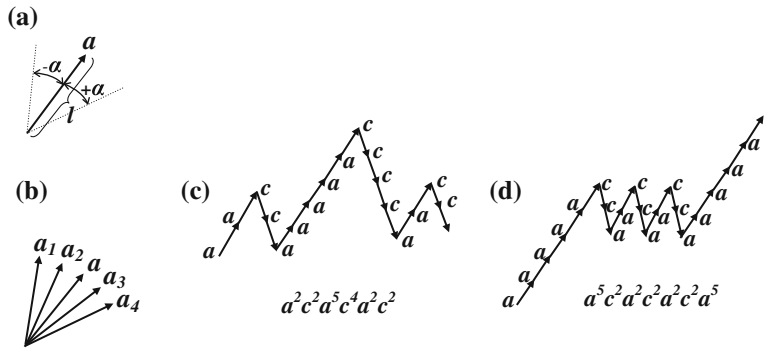

Fig. 8.4 **a** Ascribing attributes to a primitive, **b** a model primitive and patterns of its deviations, **c**-**d** structural representations of stock chart patterns

Syntactic-pattern-recognition-based AI systems have been used in various application areas. In medicine these include, apart from ECG, also EEG (monitoring the brain's electrical activity), PWA (pulse wave analysis), ABR (recording an auditory brainstem response for determining hearing levels), etc. Analysis of economic phenomena is another area of syntactic pattern recognition applications. For example, structural representations of stock chart patterns used for technical analysis are shown in Figs. 8.4c, d. A structural representation of the Head and Shoulders formation, which occurs when a trend is in the process of reversal, is shown in Fig. 8.4c, whereas a representation of the Flag formation, which is a trend continuation pattern, is shown in Fig. 8.4d.

In practical applications, if there are a lot of exemplary patterns (i.e., exemplary sentences of a language) then defining a grammar by hand is very difficult, sometimes impossible, because a (human) designer is not able to comprehend the whole set of sample patterns. Therefore, methods for automatic construction of a grammar on the basis of sample patterns have been developed. We discuss them in the next section.

8.4 Induction of Generative Grammars

We present the main idea of *grammar induction (grammatical inference)* with the help of a simple method of formal derivatives for regular grammars [104]. Firstly, we introduce the notion of a *formal derivative*. Let $A^{(0)}$ be a set of expressions built of terminal symbols. The formal derivative of a set $A^{(0)}$ with respect to a terminal symbol a, denoted $D_{\mathsf{a}}A^{(0)}$, is a set of expressions which are constructed from expressions of $A^{(0)}$ by removing a symbol a occurring at the beginning of these expressions. In other words, $D_{\mathsf{a}}A^{(0)}$ is the set of expressions x such that $\mathsf{a}x$ belongs to $A^{(0)}$.
For example, let there be given a set

$$A^{(0)} = \{\text{Jack cooks well, Jack runs quickly}\}.$$

Then

$$D_{\text{Jack}} A^{(0)} = \{\text{cooks well, runs quickly}\} = A^{(1)}.$$

We can continue by computing a formal derivative for the set $A^{(1)}$:

$$D_{\text{cooks}} A^{(1)} = \{\text{well}\} = A^{(2)}.$$

Now, if we compute a formal derivative once more, this time for the set $A^{(2)}$, then we obtain a set containing the empty word λ only:

$$D_{\text{well}} A^{(2)} = \{\lambda\} = A^{(3)}.$$

In fact, if a symbol well is attached to the empty word, then we obtain an expression well, which belongs to the set $A^{(2)}$.

Let us notice that computing a formal derivative can give the empty set as a result. For example, if we compute a formal derivative of the set $A^{(3)}$ with respect to any symbol, e.g., with respect to the symbol quickly, then we obtain:

$$D_{\text{quickly}} A^{(3)} = \emptyset,$$

because there is no such expression, which gives the empty word after attaching the symbol quickly (or any other symbol).

In this method we have to compute all the formal derivatives. Thus, let us do so:

$$D_{\text{runs}} A^{(1)} = \{\text{quickly}\} = A^{(4)},$$
$$D_{\text{quickly}} A^{(4)} = \{\lambda\} = A^{(5)}.$$

Computing any derivative of the set $A^{(5)}$ gives the empty set.

After computing all formal derivatives we can define the productions of a regular grammar which generates expressions belonging to the set $A^{(0)}$. A symbol $A^{(0)}$ is the start symbol of the grammar. Productions are defined according to the following two rules.

1. If the formal derivative of a set $A^{(n)}$ with respect to a symbol a is equal to a set $A^{(k)}$, i.e., $D_{\text{a}} A^{(n)} = A^{(k)}$, and the set $A^{(k)}$ does not consist of the empty word, then add a production $A^{(n)} \to \text{a}A^{(k)}$ to the set of productions of the grammar.
2. If the formal derivative of a set $A^{(n)}$ with respect to a symbol a is equal to a set $A^{(k)}$, i.e., $D_{\text{a}} A^{(n)} = A^{(k)}$, and the set $A^{(k)}$ consists of the empty word, then add a production $A^{(n)} \to \text{a}$ to the set of productions of the grammar.

As one can easily check, after applying these rules we obtain the following set of productions.

(1) $A^{(0)} \rightarrow$ Jack $A^{(1)}$
(2) $A^{(1)} \rightarrow$ cooks $A^{(2)}$
(3) $A^{(2)} \rightarrow$ well
(4) $A^{(1)} \rightarrow$ runs $A^{(4)}$
(5) $A^{(4)} \rightarrow$ quickly .

The method introduced above is used for the induction of a grammar, which generates only a given sample of a language. Methods which try to *generalize* a sample to the whole language are defined in syntactic pattern recognition, as well. Let us notice that such an induction of a grammar corresponds to inductive reasoning (see Appendix F.2). In fact, we go from individual cases (a sample of sentences) to their generalization of the form of a grammar.

In case of the Chomsky generative grammars a lot of induction methods have been defined for regular languages. Research results in the case of context-free languages are still unsatisfactory. However, the induction of graph grammars, which are introduced in the next section, is a real challenge.

8.5 Graph Grammars

As we have mentioned in Chap. 6, reasoning as symbolic computation is based on Abstract Rewriting Systems, ARSs, which can be divided into Term Rewriting Systems, TRSs (e.g., lambda calculus introduced in Sect. 6.5), String Rewriting Systems, SRSs, which have been discussed in previous sections with the help of the example of the Chomsky generative grammars, and *Graph Rewriting Systems, GRSs*. The last ones are used for rewriting (transforming) structures in the form of graphs. *Graph grammars*, which are introduced in this section, are the most popular kind of Graph Rewriting Systems.

Graphs are widely used in Artificial Intelligence (and in general, in computer science), because they are the most general structures used for representing aspects of the world. AI representations such as semantic networks, frames, scripts, structures used for semantic interpretation in First Order Logic, Bayesian networks, structures used in model-based reasoning—all of them are graphs. Therefore, graph grammars are an important formalism for generating (in general, transforming) such representations. First of all, we show how they can be applied for modeling (describing) processes (phenomena) of the world. We consider the example of an intelligent system for integrating areas of *Computer-Aided Design* and *Computer-Aided Manufacturing*.[33] The definition of such a representation of a mechanical part which can be *translated* automatically into the *language* of technological operations performed by manufacturing equipment is a crucial problem in this area.

[33]The example is based on a model introduced in: Flasiński M.: Use of graph grammars for the description of mechanical parts. *Computer-Aided Design 27* (1995), pp. 403–433, Elsevier.

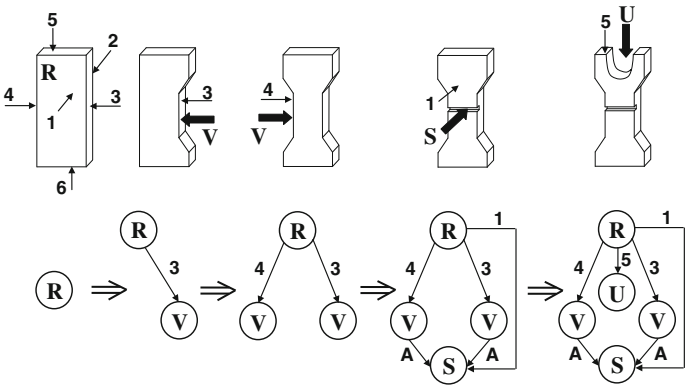

Fig. 8.5 An example of a graph grammar derivation, which represents modeling a part in a CAD system and its manufacturing controlled by a CAM system

The derivation of a graph with a graph grammar which corresponds to both a design process and a technological process is shown in Fig. 8.5. Raw material in the form of a rectangular cuboid is represented by a graph node labeled by *R*. The faces of the cuboid are indexed as shown in the figure. An application of the first production results in replacing the node *R* by a graph that consists of nodes *R* and *V*, which are connected with an edge labeled with *3*. This production corresponds to embedding a feature called a *V*-slot in the face indexed with *3* of the solid *R*.[34] In the second step of the derivation, a *V*-slot is embedded in the face indexed with *4* of the solid *R*. Then, a *S*lot is embedded in the face indexed with *1* of the solid *R*. Let us notice that this *S*lot is adjacent to both *V*-slots, which is represented by edges labeled with *A*. Finally, a *U*-slot is embedded in the face indexed with *5* of the solid *R*.

Defining a way of replacing a graph of the left-hand side of a production by a graph of the right-hand side of the production is a fundamental problem of graph grammars. (In the example above we see only the result of such a replacement.) This operation is performed with the help of the *embedding transformation*. On one hand, the embedding transformation complicates a derivation. On the other hand, it is the source of the very great descriptive power of graph grammars. It is so important that a taxonomy of graph grammars is defined on its basis. We present the embedding transformation, which was introduced by the research team of Grzegorz Rozenberg[35] for *edNLC graph grammars* in the 1980s [149].

[34] During a technological process this corresponds to milling a V-slot in the raw material.

[35] Grzegorz Rozenberg—a professor of Leiden University, the University of Colorado at Boulder, and the Polish Academy of Sciences in Warsaw, an eminent computer scientist and mathematician. His research concerns formal language theory, concurrent systems, and natural computing. Prof. G. Rozenberg was the president of the European Association for Theoretical Computer Science for 11 years.

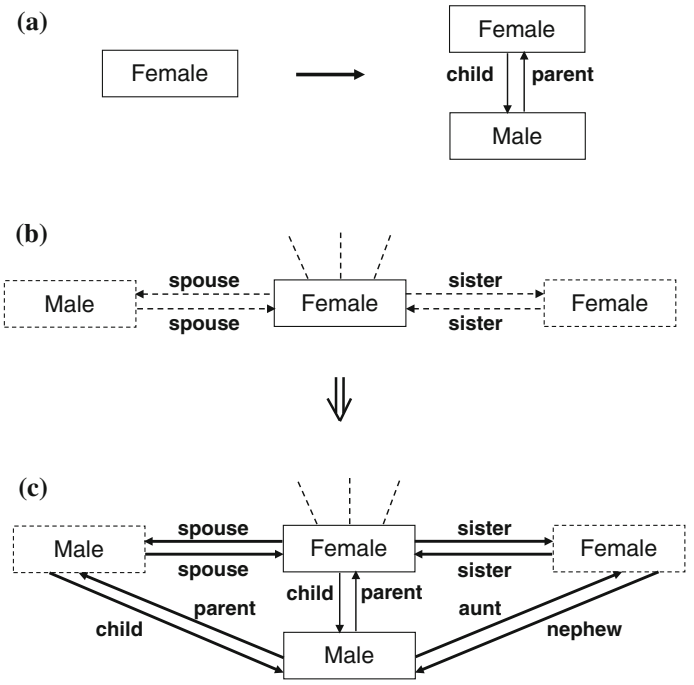

Fig. 8.6 **a** An example of a graph grammar production which is used for transforming a semantic network, **b**-**c** an application of a graph grammar production for transforming a semantic network

Let us transform a semantic network which represents family relations[36] with an edNLC graph grammar. Graphs of the left-hand side and the right-hand side of a production which represents the birth of a male child are shown in Fig. 8.6a. An object of the class Female is replaced by itself with an object of the class Male attached with the help of relations child-parent. A part of a semantic network before applying this production (i.e., before the birth) is shown in Fig. 8.6b and a part of the network after applying the production (i.e. after the birth) is shown in Fig. 8.6c. Let us notice that firstly, the production has to *reconstruct* (horizontal) edges connecting a (happy) mother with her husband and with her sister, because we have destroyed these edges in removing the node Female (mother) corresponding to the left-hand side of the production. Secondly, the production has to establish new edges between the child and his father as well as between the child and his aunt. All the reconstructed edges in Fig. 8.6c are bold. This reconstruction is performed by the production with the help of the embedding transformation, which is defined in the following way.

[36] A similar semantic network has been introduced in Sect. 7.1.

$$C(\text{spouse}, out) = \{(\text{Female}, \text{Male}, \text{spouse}, out), \quad (8.19)$$
$$(\text{Male}, \text{Male}, \text{parent}, out), \quad (8.20)$$
$$(\text{Male}, \text{Male}, \text{child}, in)\} \quad (8.21)$$
$$C(\text{sister}, out) = \{(\text{Female}, \text{Female}, \text{sister}, out), \quad (8.22)$$
$$(\text{Male}, \text{Female}, \text{aunt}, out), \quad (8.23)$$
$$(\text{Male}, \text{Female}, \text{nephew}, in)\} \quad (8.24)$$
$$C(\text{spouse}, in) = \{(\text{Female}, \text{Male}, \text{spouse}, in)\} \quad (8.25)$$
$$C(\text{sister}, in) = \{(\text{Female}, \text{Female}, \text{sister}, in)\}. \quad (8.26)$$

For example, formula (8.24) is interpreted in the following way.

- Each edge before the production application, which:
  - has been labeled by sister—C(sister, ...) and
  - has gone out (out) from the left-hand side of the production—C(....., out)

  should be replaced by
- the new edge, which:
  - connects a node of the right-hand side graph labeled by Male—(Male, ..., ..., ...),
  - with a node of the context of the production, which has been pointed out by the old edge[37] and which has been labeled by Female—(....., Female, ..., ...),
  - is labeled with nephew—(....., ..., nephew, ...)
  - and comes into (in) this node of the right-hand side graph—(....., ..., ..., in).

One can easily notice that formulas (8.19), (8.22), (8.25), and (8.26) reconstruct only the old edges, i.e., the edges, which previously existed in the semantic network. On the other hand, the remaining formulas establish new relations between the child and his father as well as between the child and his aunt.

In the case of the use of graph languages in AI we are interested in their analysis more than in their generation. Unfortunately, the construction of an efficient graph automaton is very difficult.[38] At the end of the twentieth century the *ETPL(k)* subclass of edNLC grammars with efficient automata was defined [93, 94]. ETPL(k) graph grammars have been applied for various AI areas such as transforming semantic networks in real-time expert systems, scene analysis in robotic systems, reasoning in multi-agent systems, intelligent integrators for CAD/CAM/CAPP, sign language recognition, model-based reasoning in diagnostic expert systems, etc. The problem of grammar induction, introduced in the previous section, has been solved for these grammars as well [96].

[37] The old edge has pointed out an *aunt*—C(sister, out).

[38] This was shown in the 1980s during research into the *membership problem* for graph languages, which was led (independently) by G. Turan and F.J. Brandenburg.

Bibliographical Note

Monographs [41, 104, 113, 215] are good introductions to syntactic pattern recognition.