

Grammar Formalisms for Natural Languages

2.1 Context-Free Grammars and Natural Languages

2.1.1 The Generative Capacity of CFGs

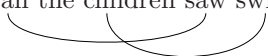
For a long time there has been a debate about whether CFGs are sufficiently powerful to describe natural languages. Several approaches have used CFGs, oftentimes enriched with some additional mechanism of transformation (Chomsky, 1956) or with features (Gazdar et al., 1985) for natural languages. These approaches were able to treat a large range of linguistic phenomena.

However, in the 1980s Stuart Shieber was able to prove in (1985) that there are natural languages that cannot be generated by a CFG. Before that, Bresnan et al. (1982) made a similar argument but their proof is based on the tree structures obtained with CFGs while Shieber argues on the basis of weak generative capacity, i.e., of the string languages.

The phenomena considered in both papers are cross-serial dependencies. Bresnan et al. (1982) argue that CFGs cannot describe cross-serial dependencies in Dutch while Shieber (1985) argues the same for Swiss German. Swiss German has case marking; therefore dependencies are visible on the strings and one can show that the string languages are not context-free.

Let us first consider the Dutch data from (Bresnan et al., 1982).

- (3) ... dat Jan de kinderen zag zwemmen
 ... that Jan the children saw swim



‘... that Jan saw the children swim’

In (3), we have two verbs and two noun phrases. The links mark the dependencies between these: *the children* is an argument of *swim* while *Jan* is an argument of *saw*. The dependency links are in a crossing configuration. This phenomenon can be iterated, as shown in (4) and (5).

- (4) ... dat Jan Piet de kinderen zag helpen zwemmen
 ... that Jan Piet the children saw help swim



‘... that Jan saw Piet help the children swim’

- (5) ... dat Jan Piet Marie de kinderen zag helpen leren zwemmen
 ... that Jan Piet Marie the children saw help teach swim



‘... that Jan saw Piet help Marie teach the children to swim’

In principle, an unbounded number of crossed dependencies is possible. However, except for the first and last verb any permutation of the NPs and the verbs is grammatical as well (even though with a completely different dependency structure since the dependencies are always cross-serial). Therefore, the string language of Dutch cross-serial dependencies amounts roughly to $\{n^k v^k \mid k > 0\}$, which is a context-free language.

Bresnan et al. (1982) argue that the strong generative capacity of CFGs is too limited for the Dutch examples. A weakness of the argument is however that an argument about syntactic structure makes always certain theoretical stipulations. Although it is very probable, it does not absolutely prove that, even using different syntactic theories, there is no context-free analysis for the Dutch examples. It only shows that the syntactic structures Bresnan et al. (1982) think the appropriate ones cannot be obtained with a CFG.

Shieber’s argument about Swiss German cross-serial dependencies is more convincing since it relies only on the string language, i.e., it concerns the weak generative capacity of CFGs. Swiss German displays the same dependency patterns as Dutch in examples such as (3)–(5). The crucial difference is that Swiss German has case marking. Let us consider the Swiss German data.

- (6) ... das mer em Hans es huus hälfed aastriiche
 ... that we Hans_{DAT} house_{ACC} helped paint



‘... that we helped Hans paint the house’

- (7) ... das mer d’chind em Hans es huus lönd hälfe aastriiche
 ... that we the children_{ACC} Hans_{DAT} house_{ACC} let help paint



‘... that we let the children help Hans paint the house’

In Swiss German, as in Dutch, the dependencies are always cross-serial in these examples. But, since we have case marking, permutations of the noun phrases would lead to ungrammatical sentences. This is why Shieber was able to show that Swiss German (as a string language) is not context-free.

Proposition 2.1.

The language L of Swiss German is not context-free (Shieber, 1985).

The argumentation of the proof goes as follows: We assume that L is context-free. Then the intersection of a regular language with the image of L under a homomorphism must be context-free as well. We find a particular homomorphism and a regular language such that the result obtained in this way is a non-context-free language. This is a contradiction to our assumption and, consequently, the assumption does not hold.

Shieber considers sentences of the following form:

- (8) ... das mer d'chind em Hans es huus haend
 ... that we the children_{ACC} Hans_{DAT} house_{ACC} have
 wele laa h  lfe aastr  che
 wanted let help paint
 ‘... that we have wanted to let the children help Hans paint the house’

Swiss German allows constructions of the form *(Jan s  it)* (‘Jan says’) *das mer (d'chind)ⁱ (em Hans)^j es huus haend wele (laa)ⁱ (h  lfe)^j aastr  che*. In these constructions the number of accusative NPs *d'chind* must equal the number of verbs (here *laa*) selecting for an accusative and the number of dative NPs *em Hans* must equal the number of verbs (here *h  lfe*) selecting for a dative object. Furthermore, the order must be the same in the sense that if all accusative NPs precede all dative NPs, then all verbs selecting an accusative must precede all verbs selecting a dative.

The following homomorphism f separates the iterated noun phrases and verbs in these examples from the surrounding material:

$$\begin{aligned} f(\text{“d'chind”}) &= a & f(\text{“Jan s  it das mer”}) &= w \\ f(\text{“em Hans”}) &= b & f(\text{“es huus haend wele”}) &= x \\ f(\text{“laa”}) &= c & f(\text{“aastr  che”}) &= y \\ f(\text{“h  lfe”}) &= d & f(s) &= z \text{ otherwise} \end{aligned}$$

To make sure we concentrate only on the constructions of the described form, we intersect $f(L)$ with the regular language $wa^*b^*xc^*d^*y$. Whenever we have a sentence whose image under f is in the intersection, this sentence has the form *(Jan s  it) das mer (d'chind)ⁱ (em Hans)^j es huus haend wele (laa)^k (h  lfe)^l aastr  che* for some $i, j, k, l \geq 0$. Furthermore, because of the constraints we observe in Swiss German, $i = k$ and $j = l$. Therefore, the result of this intersection is $\{wa^ib^jxc^id^jy \mid i, j \geq 0\}$, a language that is not

context-free.¹ Consequently, the original language L , Swiss German, is not context-free either.

Alternatively, one can also reduce Swiss German to the copy language $\{ww \mid w \in \{a, b\}^*\}$ by appropriate homomorphisms and an intersection with a regular language (see Problem 2.2 for more details). For grammar formalisms whose language classes are closed under homomorphisms and intersection with regular languages, this means the following: If such a formalism cannot generate the copy language, then it is not powerful enough to describe all natural languages. Therefore, the fact that a formalism can generate the copy language is often considered a necessary condition for the ability to describe natural languages.

2.1.2 CFGs and Lexicalization

Besides the fact that the generative capacity of CFGs is too weak to describe all natural languages, CFGs cannot be strongly lexicalized. A set of grammars can be strongly lexicalized if, for every grammar in this set, we can find a strongly equivalent lexicalized grammar in the same set. This property is sometimes claimed useful for formalisms intended to describe natural languages (Schabes, 1990; Joshi and Schabes, 1997).

Lexicalized grammars are grammars where each rewriting rule contains at least one terminal. On the one hand, lexicalized grammars are computationally interesting since in a lexicalized grammar the number of analyses for a sentence is finite (if the grammar is finite of course). On the other hand, they are linguistically interesting since, if we assume that each lexical item comes with the possibility of certain partial syntactic constructions, we would like to associate it with a set of such structures.

Another linguistic aspect of lexicalized grammars is that they relate oftentimes immediately to dependency structures since combinations during derivation can be interpreted as dependencies. This link is investigated in detail in (Kuhlmann, 2007).

Lexicalization is particularly useful for parsing since the lexical elements give us a strong indication for which rewriting rules to use, i.e., they help to restrict the search space during parsing.

A lexicalized grammar can never generate the empty word ε . Therefore, in the following we consider only languages that do not contain ε .

Definition 2.2 (Lexicalized Grammar). *A grammar is lexicalized if it consists of*

- *a finite set of elementary objects of finite size each associated with a non-empty lexical item (called its anchor),*

¹ To see that, we can intersect this language with the regular language $a^*b^*c^*d^*$, which leads to $\{a^ib^jc^id^j \mid i, j \geq 0\}$. This language can be shown to be non-context-free using the pumping lemma for context-free languages.

CFG rewriting step $\alpha A \beta \Rightarrow \alpha X_1 \dots X_k \beta$ with production $A \rightarrow X_1 \dots X_k$

Corresponding tree substitution:

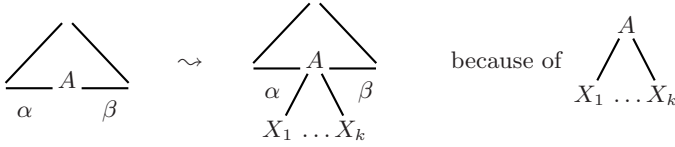


Fig. 2.1. Context-free derivation steps as substitution

- and an operation/operations for composing these structures that do not copy, erase or restructure unbounded components of their arguments.

The objects might be for instance productions as in CFG or trees as in TAG or tree descriptions (“quasi trees”) as in D-Tree Substitution Grammar (Rambow, Vijay-Shanker, and Weir, 2001).

An elementary object can contain more than one lexical item. We then call the set of its lexical items a *multicomponent anchor*.

Lexicalized grammars are *finitely ambiguous*, i.e., no sentence of finite length can be analyzed in an infinite number of ways. Consequently the recognition problem for lexicalized grammars is decidable.

Definition 2.3 (Lexicalization).

A formalism F can be strongly (weakly) lexicalized by a formalism F' if for any finitely ambiguous grammar G in F there is a lexicalized grammar G' in F' such that G and G' are strongly (weakly) equivalent.

CFG can be weakly lexicalized by CFG since for each CFG whose string language does not contain ε , a weakly equivalent lexicalized CFG can be found, namely the one in Greibach Normal Form (GNF) (see Hopcroft and Ullman (1979)).² However, the derivation trees obtained with the original CFG and the one in Greibach Normal Form are different in general.

In order to show that CFGs cannot be strongly lexicalized by CFGs, we show that they cannot be strongly lexicalized by Tree Substitution Grammars, a formalism that is strongly equivalent to CFG. Therefore, we now introduce Tree Substitution Grammars.

We can consider context-free derivation steps as tree substitutions since a non-terminal leaf is replaced with a tree of height 1 (one mother node and n daughters) as depicted in Figure 2.1.

Extending the height of the trees permitted leads to Tree Substitution Grammars:

² A CFG is in Greibach Normal Form if each production is of the form $A \rightarrow ax$ with $A \in N, a \in T, x \in (N \cup T)^*$.

Definition 2.4 (Tree Substitution Grammar).

A *Tree Substitution Grammar (TSG)* consists of a quadruple $\langle T, N, I, S \rangle$ such that

- T and N are disjoint alphabets, the terminals and non-terminals,
- I is a finite set of syntactic trees, and
- $S \in N$ is the start symbol.

We call the syntactic trees in I the *elementary trees*.

Every elementary tree is a *derived* tree and we can obtain larger derived trees from existing ones by replacing some of the non-terminal leaves with elementary trees having the same non-terminal as root label. Such operations are called *substitution*.

Definition 2.5 (Substitution).

Let $\gamma = \langle V, E, r \rangle$ be a syntactic tree, $\gamma' = \langle V', E', r' \rangle$ an initial tree and $v \in V$. $\gamma[v, \gamma']$, the result of substituting γ' into γ at node v is defined as follows:

- if v is no leaf or $l(v) \neq l(r')$, then $\gamma[v, \gamma']$ is undefined;
- otherwise, $\gamma[v, \gamma'] := \langle V'', E'', r'' \rangle$ with $V'' = V \cup V' \setminus \{v\}$ and $E'' = (E \setminus \{\langle v_1, v_2 \rangle \mid v_2 = v\}) \cup E' \cup \{\langle v_1, r' \rangle \mid \langle v_1, v \rangle \in E\}$.

A leaf that has a non-terminal label is called a *substitution node*.

A sample substitution is shown in Figure 2.2 where the *John*-tree with root node label NP is substituted into the NP substitution node in the *laughs* tree.

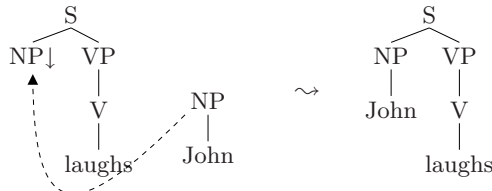


Fig. 2.2. Sample substitution

A tree is *completed* if all leaves are labeled by terminals. The tree language $T(G)$ of a TSG G is the set of all completed derived trees that have the root label S . The string language of G is then the set of strings yielded by the trees in the tree language.

TSGs are weakly equivalent to CFGs and each CFG is a TSG.

Proposition 2.6. *CFG cannot be strongly lexicalized by TSG (Schabes, 1990; Joshi and Schabes, 1997).*

Proof. Consider the CFG G with productions $S \rightarrow SS$, $S \rightarrow a$. Assume that there is a strongly equivalent lexicalized TSG G' . Then each tree in the tree language is derived from some initial tree t with a leaf labeled with a such that the path between this leaf and the root has a constant length n . Below this leaf nothing can be added, i.e., each tree derived from t still has a path of length n . Let n_{max} be the maximal path length between root and leaf with label a in the initial trees of G' . Then there is no derived tree in the tree language of G' such that all paths have a length $> n_{max}$. But such trees exist in the tree language of G . Contradiction. \square

Then, trivially, CFGs cannot strongly lexicalize CFGs either.

The reason why TSG cannot strongly lexicalize CFG is that in a TSG we always add material below one of the leaves. Consequently, TSGs do not permit the distance between two nodes in the same elementary tree to increase. One way to overcome this is to allow not only leaves but also internal nodes to be replaced with new elementary trees. This leads to tree-rewriting grammars with adjunction, i.e., to Tree Adjoining Grammars.

2.1.3 Mild Context-Sensitivity

Once it was clear that CFGs were not powerful enough to describe all natural language phenomena, the question of the appropriate context-sensitive formalism for natural languages arose. In an attempt to characterize the amount of context-sensitivity required, Aravind Joshi introduced the notion of mild context-sensitivity (1985). This is a term that refers to classes of languages, not to formalisms.

Definition 2.7 (Mildly context-sensitive).

1. A set \mathcal{L} of languages is mildly context-sensitive iff
 - a) \mathcal{L} contains all context-free languages.
 - b) \mathcal{L} can describe cross-serial dependencies: There is an $n \geq 2$ such that $\{w^k \mid w \in T^*\} \in \mathcal{L}$ for all $k \leq n$.
 - c) The languages in \mathcal{L} are polynomially parsable, i.e., $\mathcal{L} \subset PTIME$.
 - d) The languages in \mathcal{L} have the constant growth property.
2. A formalism F is mildly context-sensitive iff the set $\{L \mid L = L(G) \text{ for some } G \in F\}$ is mildly context-sensitive.

The constant growth property roughly means that, if we order the words of a language according to their length, then the length grows in a linear way. E.g., $\{a^{2^n} \mid n \geq 0\}$ does not have the constant growth property. The following definition is from Weir (1988).

Definition 2.8 (Constant Growth Property).

Let X be an alphabet and $L \subseteq X^*$. L has the constant growth property iff there is a constant $c_0 > 0$ and a finite set of constants $C \subset \mathbb{N} \setminus \{0\}$ such that for all $w \in L$ with $|w| > c_0$, there is a $w' \in L$ with $|w| = |w'| + c$ for some $c \in C$.

As already mentioned, mild context-sensitivity is introduced as a property of a set of languages. So far, it has not been possible to identify a grammar formalism that generates the largest possible mildly context-sensitive set of string languages. The closest approximation we know of are *Linear Context-Free Rewriting Systems (LCFRSs)*, introduced in (Vijay-Shanker, Weir, and Joshi, 1987; Weir, 1988), and equivalent formalisms such as *set-local Multicomponent Tree Adjoining Grammars (MCTAGs)* (Weir, 1988), *Multiple Context-Free Grammars (MCFGs)* (Seki et al., 1991) and *simple Range Concatenation Grammars (simple RCGs)* (Boullier, 2000b). However, recent research on certain types of MCTAG suggests that there might be mildly context-sensitive grammar formalisms that are not comparable with LCFRS and equivalent formalisms, i.e., that generate languages that cannot be generated by LCFRS and vice versa (Kallmeyer and Satta, 2009).

There are different ways to show the constant growth property for a specific formalism. Oftentimes, constant growth follows from a pumping lemma. If there is no pumping lemma, then one might show the constant growth property of a language class by showing the semilinearity (Parikh, 1966) of the languages. Constant growth follows from semilinearity.

Let us introduce semilinearity.

First, we define for $\langle a_1, \dots, a_n \rangle, \langle b_1, \dots, b_n \rangle \in \mathbb{N}^n$ and $m \in \mathbb{N}$ that $\langle a_1, \dots, a_n \rangle + \langle b_1, \dots, b_n \rangle := \langle a_1 + b_1, \dots, a_n + b_n \rangle$ and $m\langle a_1, \dots, a_n \rangle := \langle ma_1, \dots, ma_n \rangle$.

A Parikh mapping is a function counting for each letter of an alphabet the occurrences of this letter in a word w :

Definition 2.9 (Parikh mapping).

Let $X = \{a_1, \dots, a_n\}$ be an alphabet with some (arbitrary) fixed order of the elements. The Parikh mapping $p : X^* \rightarrow \mathbb{N}^n$ (with respect to this order) is defined as follows:

- For all $w \in X^* : p(w) := \langle |w|_{a_1}, \dots, |w|_{a_n} \rangle$ where $|w|_{a_i}$ is the number of occurrences of a_i in w .
- For all languages $L \subseteq X^* : p(L) := \{p(w) \mid w \in L\}$ is the Parikh image of L .

Two words are *letter equivalent* if they contain equal number of occurrences of each terminal symbol, and two languages are letter equivalent if every string in one language is letter equivalent to a string in the other language and vice-versa.

Definition 2.10 (Letter equivalent).

Let X be an alphabet.

1. Two words $w_1, w_2 \in X^*$ are letter equivalent if there is a Parikh mapping p such that $p(w_1) = p(w_2)$.
2. Two languages $L_1, L_2 \subseteq X^*$ are letter equivalent if there is a Parikh mapping p such that $p(L_1) = p(L_2)$.

Definition 2.11 (Semilinear).

1. Let x_0, \dots, x_m with $m \geq 0$ be in \mathbb{N}^n for some $n \geq 0$.
The set $\{x_0 + n_1x_1 + \dots + n_mx_m \mid n_i \in \mathbb{N} \text{ for } 1 \leq i \leq m\}$ is a linear subset of \mathbb{N}^n .
2. The union of finitely many linear subsets of \mathbb{N}^n is a semilinear subset of \mathbb{N}^n .
3. A language $L \subseteq X^*$ is semilinear iff there is a Parikh mapping p such that $p(L)$ is a semilinear subset of \mathbb{N}^n for some $n \geq 0$.

Lemma 2.12. *The constant growth property holds for semilinear languages.*

Proof. Assume $L \subseteq X^*$ is semilinear and $p(L)$ is a semilinear Parikh image of L where $p(L)$ is the union of the linear sets M_1, \dots, M_l . Then the constant growth property holds for L with

$$c_0 := \max\{\Sigma_{i=1}^n y_i \mid \text{there are } x_1, \dots, x_m \text{ such that} \\ \langle y_1, \dots, y_n \rangle + n_1x_1 + \dots + n_mx_m \mid n_i \in \mathbb{N}\} \\ \text{is one of the sets } M_1, \dots, M_l\} \text{ and}$$

$$C := \{\Sigma_{i=1}^n y_i \mid \text{there are } x_1, \dots, x_m \text{ such that} \\ \{x_1 + n_1\langle y_1, \dots, y_n \rangle + \dots + n_mx_m \mid n_i \in \mathbb{N}\} \\ \text{is one of the sets } M_1, \dots, M_l\}.$$

□

Parikh has shown that a language is semilinear if and only if it is letter equivalent to a regular language. The proof is given in (Kracht, 2003, p. 151). As a consequence, we obtain that context-free languages are semilinear.

Proposition 2.13 (Parikh Theorem).

Each context-free language is semilinear (Parikh, 1966).

Furthermore, each language that is letter equivalent to a semilinear language is semilinear as well since the Parikh images of the two languages are equal. Therefore, in order to show the semilinearity (and constant growth) of a language, it is sufficient to show letter equivalence to a context-free language.

As far as we know, Joshi's hypothesis that natural languages are mildly context-sensitive has been questioned only by two natural language phenomena that have been claimed to be non-semilinear, namely case stacking in Old Georgian (Michaelis and Kracht, 1997) and Chinese number names (Radzinski, 1991). The analyses of Old Georgian, however, are based on very few data since there are no speakers of Old Georgian today. Therefore, it is hard to tell whether there is really an infinite progression of case stacking possible. Concerning Chinese number names, it is not totally clear to what extent this constitutes a syntactic phenomenon. Therefore, even with these counterexamples, there is still good reason to assume that natural languages are mildly context-sensitive.

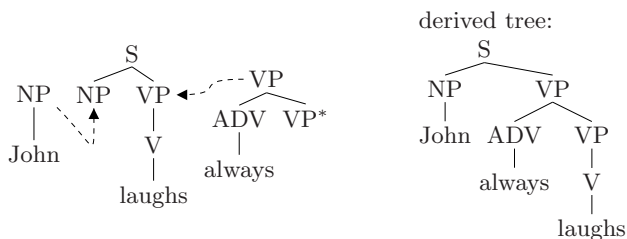


Fig. 2.3. TAG derivation for *John always laughs*

2.2 Grammar Formalisms Beyond CFG

We have seen that CFGs are not powerful enough to deal with all natural language phenomena. This is one of the reasons why we are interested in investigating extensions of CFG. We now introduce the different formalisms that will be treated in this book. The formalisms presented in this section will be defined in detail in the corresponding chapters on parsing. This section aims only at providing an intuition of how these formalisms extend CFG, how they model natural language phenomena and how they are related to each other.

2.2.1 Tree Adjoining Grammars

The Formalism

Starting from Tree Substitution Grammars, if we allow also for replacing internal nodes with new trees, we obtain Tree Adjoining Grammars. Tree Adjoining Grammar (TAG, Joshi, Levy, and Takahashi (1975; Joshi and Schabes (1997))) is a tree-rewriting formalism. A TAG consists of a finite set of syntactic trees (so-called *elementary trees*). Starting from the elementary trees, larger trees are derived by substitution (replacing a leaf with a new tree) and adjunction (replacing an internal node with a new tree). In case of an adjunction, the tree being adjoined has exactly one leaf that is marked as the *foot node* (marked with an asterisk). Such a tree is called an *auxiliary tree*. When adjoining it to a node n , in the resulting tree, the subtree with root n from the old tree is attached to the foot node of the auxiliary tree. Non-auxiliary elementary trees are called *initial trees*. A derivation starts with an initial tree. In a final derived tree, all leaves must have terminal labels.

For a sample derivation see Figure 2.3 where the tree for *John* is substituted for the subject NP slot while the auxiliary tree for the modifier *always* adjoins to the VP node in the tree of *laughs*.

The internal nodes in $I \cup A$ can be marked as *OA* (obligatory adjunction) and *NA* (null adjunction, i.e., no adjunction allowed). Furthermore, for nodes

that are not *NA*, one can specify the set of auxiliary trees that can be adjoined at that node.

As a second example, Figure 2.4 shows a TAG for the copy language and Figure 2.5 shows a sample derivation using the trees from this grammar. (*NA* stands for “null adjunction”, i.e., no adjunction allowed at that node. *OA* stands for “obligatory adjunction”, i.e., adjunction mandatory at that node.) In this TAG, the *NA* constraints are crucial since they make sure that the adjunction always targets the middle *S* node. Without adjunction constraints, it is not possible for TAG to generate the copy language.

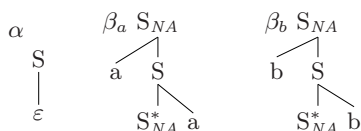


Fig. 2.4. TAG for the copy language

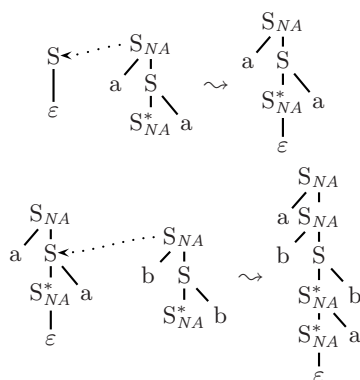


Fig. 2.5. A sample derivation of a word in the copy language

TAG derivations are represented by derivation trees (unordered trees) that record the history of how the elementary trees are put together. A derived tree is the result of carrying out the substitutions and adjunctions, i.e., the derivation tree describes uniquely the derived tree. Each edge in a derivation tree stands for an adjunction or a substitution. The edges are labeled with Gorn addresses. E.g., the derivation tree in Figure 2.6 indicates that the elementary tree for *John* is substituted for the node at address 1 and *always* is adjoined at node address 2 (the fact that the former is an adjunction and the latter is

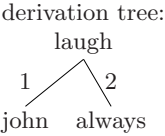


Fig. 2.6. TAG derivation tree for *John always laughs*

a substitution can be inferred from the fact that the node at address 1 is a leaf that is not a foot node while the node at address 2 is an internal node).

The fact that TAGs are able to generate the copy language indicates that they are powerful enough to describe cross-serial dependencies. An actual analysis has been proposed in (Kroch and Santorini, 1991); it is shown in Figures 2.7 and 2.8.

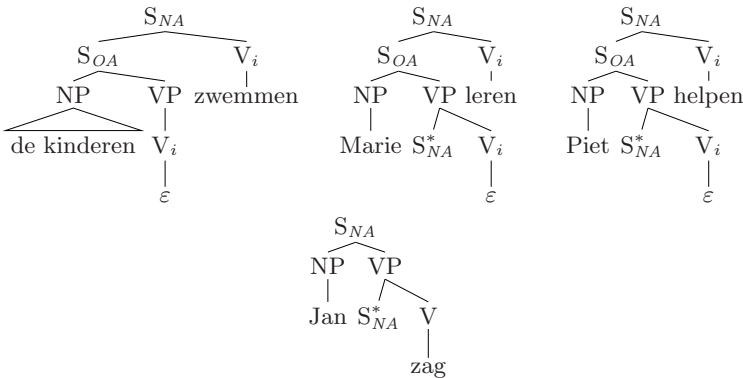


Fig. 2.7. TAG for cross-serial dependencies

Lexicalization

As we have seen in Section 2.1.2, in order to lexicalize CFGs one has to extract recursive sub-trees (with root and some leaf having the same non-terminal symbol) and put them into extra structures. This leads to a set of trees with an adjunction operation, i.e., to a TAG.

As an example, consider again the CFG in Figure 2.9 that cannot be lexicalized using only substitution. With adjunction, a lexicalization of this CFG is possible. The corresponding TAG is given in Figure 2.9.

In general it can be shown that CFGs can be lexicalized by TAGs and, furthermore, TAGs are closed under strong lexicalization. I.e., for each grammar that is a CFG or a TAG, there is a strongly equivalent lexicalized TAG (LTAG) (Schabes, 1990; Joshi and Schabes, 1997).

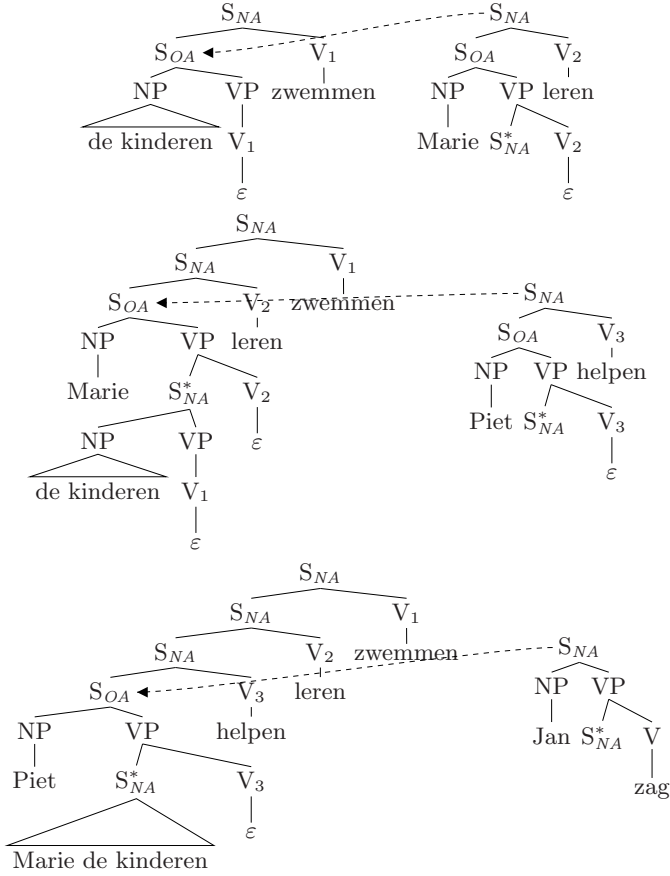


Fig. 2.8. Derivation of (5) using adjunction

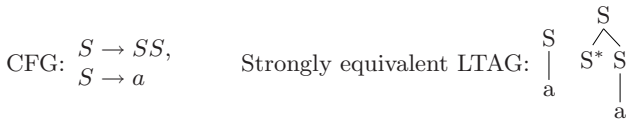


Fig. 2.9. CFG and strongly equivalent lexicalized TAG

Extended domain of locality and factoring of recursion

Because of the move to larger trees (compared to CFGs) and the addition of adjunction, TAGs have some properties that make them particularly interesting for natural language processing.

TAG elementary trees allow to express locally dependencies such as filler-gap dependencies, even if they are ‘unbound’. This is why TAG is said to have an *extended domain of locality*. Two properties are crucial for obtaining this extended domain of locality: TAG elementary trees can be arbitrarily large (but have to be finite), and recursion can be factored away because of adjunction. Consequently, even so-called unbounded dependencies can be captured locally, i.e., inside single elementary trees (Kroch, 1987; Frank, 1992; Frank, 2002). Because of the constraints that hold for adjunction, in many cases one gets locality constraints for unbounded dependencies for free.

- (9) a. **whom**_{*i*} did John tell Sam **that** Bill likes *t*_{*i*}
- b. **whom**_{*i*} did John tell Sam that Mary said **that** Bill likes *t*_{*i*}

As an example that illustrates this property of TAG, consider the derivation of (9a.) in Figure 2.10 with the recursive part being put in a separate tree that gets adjoined.

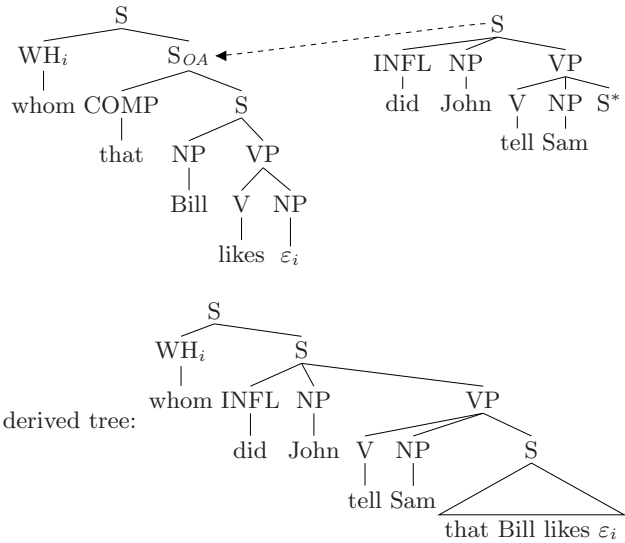


Fig. 2.10. Derivation for an unbounded dependency

When dealing with natural languages, one always uses *Lexicalized Tree Adjoining Grammars (LTAGs)*. The linguistic theory implemented within LTAG

is roughly as follows. The grammar contains extended projections of each lexical item (the elementary trees anchored by this lexical item). These extended projections satisfy certain linguistic principles that are not part of the TAG formalism itself. The extended projections are minimal in the sense that they contain slots only for the arguments of their lexical head. Recursion is factored away. Consequently, the set of elementary structures in the grammar is finite. Every constraint concerning larger structures (constraints on “unbounded dependencies”) does not need to be stipulated but, instead, follows from the possibilities of adjunction in the extended projections.

We will give a more detailed discussion of LTAG for natural languages in Chapter 4.

2.2.2 Linear Indexed Grammars

Indexed grammars (IGs) were introduced by (Aho, 1968). An indexed grammar looks like a CFG except that the non-terminals are equipped with stacks of indices, i.e., besides the non-terminals N and the terminals T , we have an alphabet I of indices. In a derived sentential form x , non-terminals can be equipped with stacks of indices, i.e., $x \in (NI^* \cup T)^*$.

The productions in an IG have the form (i) $A \rightarrow \alpha$ or (ii) $A \rightarrow Bf$ or (iii) $Af \rightarrow \alpha$ with $A, B \in N, f \in I, \alpha \in (N \cup T)^*$. The first kind of production works like context-free productions while copying the stack of A to all non-terminals in α . The second kind of production adds a symbol to the stack of A while replacing A with B . The third kind of production deletes a symbol f from the stack of A and then works like the first kind of production.

As an example consider the IG for $\{a^{2^n} \mid n \geq 0\}$ with $N := \{S, A, B\}, I := \{f, g\}, T := \{a\}$ and productions $P := \{S \rightarrow a, S \rightarrow Ag, A \rightarrow Af, A \rightarrow B, Bf \rightarrow BB, Bg \rightarrow aa\}$. This grammar works as follows: For a word a^{2^n} with $n \geq 1$, we first apply the production $S \rightarrow Ag$ and then n times the production $A \rightarrow Af$. This leads to a non-terminal A with a stack of length n . Then the A is turned into a B , and, while reducing the stack, the B is doubled (with the production $Bf \rightarrow BB$). This happens $n - 1$ times. Then we reach the last stack symbol and, while reducing this as well, we finally generate two terminals aa . Crucially, when doubling the B with $Bf \rightarrow BB$, the remaining stack is passed to both B s in the right-hand side of the production. This guarantees that the two parts have the same number of a s (since they have the same stacks). Figure 2.11 shows a sample derivation with this grammar.

An indexed grammar is called a *linear indexed grammar (LIG)* (Gazdar, 1988; Vijay-Shanker, 1987) if in a production $A \rightarrow \alpha$ or $Af \rightarrow \alpha$ the stack of A is copied only to one non-terminal in α .

We write the productions in a LIG as follows:

- $A[\dots] \rightarrow X_1 \dots X_i[\dots] \dots X_n$ with $X_j \in N \cup T$ for $j \neq i$, $X_i \in N$.
- $A[\dots] \rightarrow B[f \dots]$
- $A[f \dots] \rightarrow X_1 \dots X_i[\dots] \dots X_n$ with $X_j \in N \cup T$ for $j \neq i$, $X_i \in N$.

$S \Rightarrow Ag$	production $S \rightarrow Ag$
$\Rightarrow Afg$	production $A \rightarrow Af$
$\stackrel{*}{\Rightarrow} Afffg$	
$\Rightarrow Bffffg$	production $A \rightarrow B$
$\Rightarrow BffgBffg$	production $Bf \rightarrow BB$
$\stackrel{*}{\Rightarrow} BfgBfgBfgBfg$	
$\stackrel{*}{\Rightarrow} BgBgBgBgBgBgBgBg$	
$\stackrel{*}{\Rightarrow} aaaaaaaaaaaaaaaaaa$	production $Bg \rightarrow aa$

Fig. 2.11. IG derivation for $a^{2^4} = a^{16}$

As an example consider the LIG for the copy language from Figure 2.12.

$$\begin{aligned}
S_0 &\rightarrow S[\#] \\
S[.] &\rightarrow aS_a[.] \quad S_a[.] \rightarrow S[a.] \\
S[.] &\rightarrow bS_b[.] \quad S_b[.] \rightarrow S[b.] \\
S &\rightarrow T \\
T[a..] &\rightarrow T[.]a \quad T[b..] \rightarrow T[.]b \\
T[\#] &\rightarrow \varepsilon
\end{aligned}$$

Fig. 2.12. LIG for the copy language

It has been shown that LIG and TAG are weakly equivalent (Vijay-Shanker, 1987; Vijay-Shanker and Weir, 1994).

When constructing a LIG that is equivalent to a given TAG, whenever an adjunction is performed, while traversing the adjoined tree, the stack can be used to keep track of the tree one has to go back to once the adjunction is finished. It needs to be passed along the path from the root to the foot node. Figure 2.13 shows the LIG one obtains when constructing an equivalent LIG for the TAG for the copy language given in Figure 2.4 along these lines.

$$\begin{aligned}
\langle S, \alpha \rangle &\rightarrow \varepsilon \\
\langle S, \alpha \rangle &\rightarrow \langle S_1, \beta_a \rangle [\langle \alpha, 0 \rangle] & \langle S, \alpha \rangle &\rightarrow \langle S_1, \beta_b \rangle [\langle \alpha, 0 \rangle] \\
\langle S_1, \beta_a \rangle [..] &\rightarrow a \langle S_2, \beta_a \rangle [..] & \langle S_1, \beta_b \rangle [..] &\rightarrow b \langle S_2, \beta_b \rangle [..] \\
\langle S_2, \beta_a \rangle [..] &\rightarrow \langle S_3, \beta_a \rangle [..] a & \langle S_2, \beta_b \rangle [..] &\rightarrow \langle S_3, \beta_b \rangle [..] b \\
\langle S_2, \beta_a \rangle [..] &\rightarrow \langle S_1, \beta_a \rangle [\langle \beta_a, 2 \rangle \dots] & \langle S_2, \beta_a \rangle [..] &\rightarrow \langle S_1, \beta_b \rangle [\langle \beta_a, 2 \rangle \dots] \\
\langle S_2, \beta_b \rangle [..] &\rightarrow \langle S_1, \beta_a \rangle [\langle \beta_b, 2 \rangle \dots] & \langle S_2, \beta_b \rangle [..] &\rightarrow \langle S_1, \beta_b \rangle [\langle \beta_b, 2 \rangle \dots] \\
\langle S_3, \beta_a \rangle [\langle \alpha, 0 \rangle \dots] &\rightarrow \langle S, \alpha \rangle [..] & \langle S_3, \beta_b \rangle [\langle \alpha, 0 \rangle \dots] &\rightarrow \langle S, \alpha \rangle [..] \\
\langle S_3, \beta_a \rangle [\langle \beta_a, 2 \rangle \dots] &\rightarrow \langle S_2, \beta_a \rangle [..] & \langle S_3, \beta_b \rangle [\langle \beta_a, 2 \rangle \dots] &\rightarrow \langle S_2, \beta_a \rangle [..] \\
\langle S_3, \beta_a \rangle [\langle \beta_b, 2 \rangle \dots] &\rightarrow \langle S_2, \beta_b \rangle [..] & \langle S_3, \beta_b \rangle [\langle \beta_b, 2 \rangle \dots] &\rightarrow \langle S_2, \beta_b \rangle [..]
\end{aligned}$$

Fig. 2.13. Equivalent LIG for the TAG from Figure 2.4

Productions of the Generalized CFG (start symbol is S):

$$\begin{aligned} S &\rightarrow f_1(A, B, C) & A &\rightarrow f_2(A) & B &\rightarrow f_3(B) & C &\rightarrow f_4(C) \\ & & A &\rightarrow f_5() & B &\rightarrow f_5() & C &\rightarrow f_5() \end{aligned}$$

Strings $\phi(t)$ yielded by the terms t :

$$\begin{aligned} \phi(f_5()) &:= \langle \varepsilon, \varepsilon \rangle, \\ \phi(f_2(t)) &:= \langle aw_1, aw_2 \rangle \text{ where } \langle w_1, w_2 \rangle = \phi(t), \\ \phi(f_3(t)) &:= \langle bw_1, bw_2 \rangle \text{ where } \langle w_1, w_2 \rangle = \phi(t), \\ \phi(f_4(t)) &:= \langle cw_1, cw_2 \rangle \text{ where } \langle w_1, w_2 \rangle = \phi(t), \\ \phi(f_1(t_1, t_2, t_3)) &:= \langle w_1u_1v_1w_2u_2v_2 \rangle \\ &\text{where } \langle w_1, w_2 \rangle = \phi(t_1), \langle u_1, u_2 \rangle = \phi(t_2), \langle v_1, v_2 \rangle = \phi(t_3) \end{aligned}$$

Fig. 2.14. An LCFRS for $\{a^n b^m c^k a^n b^m c^k \mid n, m, k \geq 0\}$

LIGs themselves are not used for natural languages. Their interest lies in their relations to other formalisms, in particular in their equivalence to TAGs. Because of this equivalence, LIGs have been proposed for TAG parsing (Vijay-Shanker and Weir, 1993; Boullier, 1996).

2.2.3 Linear Context-Free Rewriting Systems

Linear Context-Free Rewriting Systems (LCFRSs) are introduced in (Vijay-Shanker, Weir, and Joshi, 1987; Weir, 1988). They are grammars that have an underlying context-free structure. More concretely, an LCFRS consists of

1. a generalized context-free grammar (GCFG) that generates a set of terms,
2. a yield function that specifies the strings yielded by these structures.

LCFRS is more powerful than TAG and LIG. More concretely, every TAG can be written as an LCFRS.

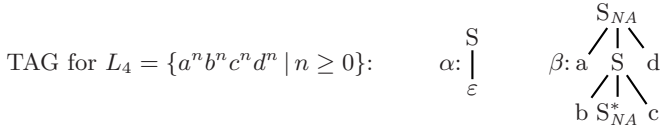
In an LCFRS, the yield $\phi(t)$ of a term t is a sequence of strings. A unique equation is associated with each production $A \rightarrow f(A_1, \dots, A_n)$ in C . It describes how to compute the yield of a term $f(t_1, \dots, t_n)$ from the yields of t_1, \dots, t_n and a bounded collection of new terminals. When computing the yield of a left-hand side from the yields of a right-hand side, we must neither copy nor erase.

As an example consider the LCFRS in Figure 2.14.

The languages generated by these grammars are mildly context-sensitive and they properly contain the languages generated by TAG. Figure 2.15 shows an example of an equivalent LCFRS for a given TAG.

2.2.4 Multicomponent Tree Adjoining Grammars

Multicomponent Tree Adjoining Grammars (MCTAGs) were first introduced in (Joshi, Levy, and Takahashi, 1975) as *simultaneous TAGs*, later redefined as *multicomponent TAGs (MCTAGs)* in (Weir, 1988; Joshi, 1985). The underlying linguistic motivation is the idea to separate the contribution of a lexical



Productions of the corresponding Generalized CFG (start symbol is α):

$$\begin{aligned} \alpha &\rightarrow f_\alpha(), \beta \rightarrow f_\beta() \text{ (no adjunctions),} \\ \alpha &\rightarrow f_{\alpha:\varepsilon}(\beta), \beta \rightarrow f_{\beta:1}(\beta) \text{ (adjunctions of } \beta). \end{aligned}$$

Strings $\phi(t)$ yielded by the terms t :

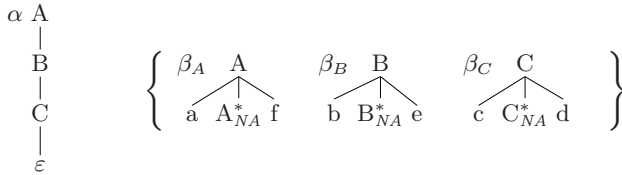
$$\begin{aligned} \phi(f_\alpha()) &:= \varepsilon, \\ \phi(f_\beta()) &:= \langle ab, cd \rangle, \\ \phi(f_{\alpha:\varepsilon}(t)) &:= \langle w_1 w_2 \rangle \text{ where } \langle w_1, w_2 \rangle = \phi(t), \\ \phi(f_{\beta:1}(t)) &:= \langle aw_1 b, cw_2 d \rangle \text{ where } \langle w_1, w_2 \rangle = \phi(t). \end{aligned}$$

Fig. 2.15. An LCFRS for a given TAG

item into several components. Instead of single trees, these grammars contain (finite) sets of trees. In each derivation step, a new set is picked and all trees from the set are added simultaneously, i.e., they are attached (by substitution or adjunction) to different nodes in the already derived tree.

As in TAG, a derivation starts from an initial tree and in the end, in the final derived tree, all leaves must have terminal labels (or the empty word) and there must not be any *OA* constraints left.

A sample MCTAG with a derivation is shown in Figure 2.16.



Derivation for $aabbccddeeff$:

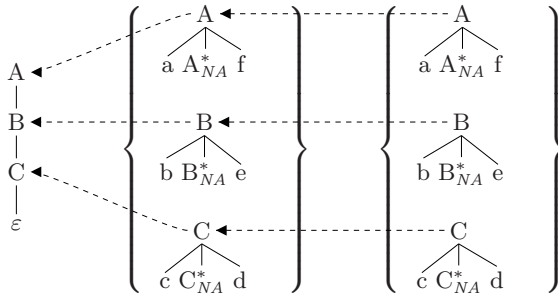


Fig. 2.16. MCTAG for $L_6 = \{a^n b^n c^n d^n e^n f^n \mid n \geq 0\}$ with sample derivation

MCTAGs are linguistically interesting because they extend the domain of locality since the contributions of single lexical elements are separated into different trees. As an example, consider extractions out of complex NPs (Kroch, 1989) as in (10). A possible MCTAG analysis is shown in Figure 2.17.

(10) which painting_{*i*} did you see a picture of *t_i*

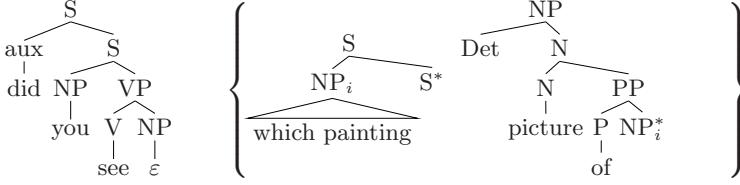


Fig. 2.17. MCTAG elementary trees for extraction from NP

An MCTAG is called *tree-local* iff in each derivation step, the nodes the new trees attach to belong to the same elementary tree. It is called *set-local* iff in each derivation step, the nodes the new trees attach to belong to the same elementary tree set. Otherwise it is called *non-local*. The derivation in Figure 2.16 for example is a set-local derivation. Usually, the term “MCTAG” without specification of the locality means “set-local MCTAG”.

Concerning the respective generative capacity, it has been shown that tree-local MCTAGs are strongly equivalent to TAGs while set-local MCTAGs are weakly equivalent to LCFRSs. As an example, Figure 2.18 shows the LCFRS that is equivalent to the set-local MCTAG from Figure 2.16.

GCFG productions:

$$\begin{aligned} \alpha &\rightarrow f_\alpha(), \alpha \rightarrow g_\alpha(\beta_{A,B,C}), \\ \beta_{A,B,C} &\rightarrow f_{A,B,C}(), \\ \beta_{A,B,C} &\rightarrow g_{A,B,C}(\beta_{A,B,C}). \end{aligned}$$

Yield function ϕ :

$$\begin{aligned} \phi(f_\alpha()) &:= \langle \varepsilon \rangle, \\ \phi(f_{A,B,C}()) &:= \langle a, b, c, d, e, f \rangle, \\ \phi(g_\alpha(t)) &:= \langle w_1 w_2 w_3 w_4 w_5 w_6 \rangle \text{ where } \langle w_1, w_2, w_3, w_4, w_5, w_6 \rangle = \phi(t), \\ \phi(g_{A,B,C}(t)) &:= \langle w_1 a, w_2 b, w_3 c, dw_4, ew_5, fw_6 \rangle \\ &\text{where } \langle w_1, w_2, w_3, w_4, w_5, w_6 \rangle = \phi(t) \end{aligned}$$

Fig. 2.18. LCFRS for $L_6 = \{a^n b^n c^n d^n e^n f^n \mid n \geq 0\}$

2.2.5 Multiple Context-Free Grammars

Multiple Context-Free Grammars (MCFGs), (Seki et al., 1991), are very similar to LCFRSs. The non-terminals in an MCFG, in contrast to CFG, can yield sequences of terminals, i.e., their span can be discontinuous in the input. Each non-terminal has a fixed dimension that determines the number of components in its span. In other words, from a non-terminal of dimension k , k -tuples of terminal strings are derived. The dimension of the start symbol S is 1.

An MCFG, similar to the GCFG of an LCFRS, contains productions of the form $A_0 \rightarrow f[A_1, \dots, A_k]$ where f is a function from a given set of functions F . The idea is that f describes how to compute the yield of A_0 (a $\dim(A_0)$ -tuple of terminal strings) from the yields of A_1, \dots, A_k . f must be linear in the sense that each of its arguments is used at most once to compute the new string tuple. Note that the functions f are not required not to delete parts of their input as in the case of LCFRS. In other words, it might be the case that some of the arguments in the right-hand side of a production are not used to compute the yield of the left-hand side. However, even though deletion in the yield computation is allowed in MCFG and not in LCFRS, the two formalisms are weakly equivalent (Seki et al., 1991).

As an example consider the MCFG in Figure 2.19 that generates the language $\{a^n b^n c^n d^n \mid n \geq 1\}$.

Productions:

$$S \rightarrow f[A], A \rightarrow g[A], A \rightarrow h[].$$

Yield functions:

$$h[] = (ab, cd), g[(x_1, x_2)] = (ax_1b, cx_2d), f[(x_1, x_2)] = (x_1x_2)$$

Fig. 2.19. MCFG for $\{a^n b^n c^n d^n \mid n \geq 1\}$

MCFGs have been investigated mainly in the context of biological applications such as the modeling of RNA pseudoknotted structures (Kato, Seki, and Kasami, 2006). However, because of their equivalence to LCFRSs and set-local MCTAGs, they are useful for natural language processing as well.

2.2.6 Range Concatenation Grammars

If we incorporate the definitions of the yield functions in MCFG and LCFRS into the productions themselves, and, in addition, if we relax the conditions on the yield functions, we obtain *Range Concatenation Grammars (RCGs)* (Boullier, 2000b).

The idea of RCGs is roughly that the productions of RCGs (called *clauses*) rewrite predicates ranging over parts of the input by other predicates. As an example consider the clause $S(axb) \rightarrow S(X)$. This clause signifies that the

predicate S (a unary predicate) holds for a part of the input if (i) this part starts with an a and ends with a b and (ii) S also holds for the part between the a and the b .

The RCG with clauses $S(aXb) \rightarrow S(X), S(c) \rightarrow \varepsilon$ for example generates the language $\{a^n cb^n \mid n \geq 0\}$.

An RCG consists of an alphabet N of non-terminals (called predicates) of a fixed arity (this corresponds to the dimension from MCFG) where the special predicate S has arity 1. Furthermore, it has a terminal alphabet T and an alphabet of variables V . The clauses have the form

$$A(\alpha_1, \dots, \alpha_{\dim(A)}) \rightarrow \varepsilon$$

or

$$A(\alpha_1, \dots, \alpha_{\dim(A)}) \rightarrow A_1(\alpha_1^{(1)}, \dots, \alpha_{\dim(A_1)}^{(1)}) \dots A_n^{(n)}(\alpha_1, \dots, \alpha_{\dim(A_n)}^{(n)})$$

where the predicates are from N and their arguments are words over $(T \cup V)$.

For a given clause, an instantiation with respect to a string $w = t_1 \dots t_n$ maps all variables and all occurrences of terminals in the clause to ranges $\langle i, j \rangle$ with $0 \leq i \leq j \leq |w|$. A range $\langle i, j \rangle$ denotes the part of w between positions i and j . An instantiation must be such that all occurrences of a terminal t are mapped to a range whose yield is a t , and adjacent variables/occurrences of terminals in one of the arguments are mapped on adjacent ranges, i.e., ranges $\langle i, j \rangle, \langle k, l \rangle$ with $j = k$.

A derivation step consists of replacing the left-hand side of an instantiated clause with its right-hand side. The language of an RCG G is the set of strings w that satisfy the start predicate S , in other words, the set of w such that ε can be derived from $S(\langle 0, |w| \rangle)$.

RCGs are called *simple* if (i) the arguments in the right-hand sides of the clauses are single variables, (ii) no variable appears more than once in the left-hand side of a clause or more than once in the right-hand side of a clause, and (iii) each variable occurring in the left-hand side of a clause occurs also in its right-hand side and vice versa.

Simple RCGs are weakly equivalent to LCFRSs and MCFGs. RCGs in general however are more powerful; they generate exactly the class PTIME of polynomially parsable languages (Bertsch and Nederhof, 2001). They properly include the set of languages generated by LCFRS and even the maximal set of mildly context-sensitive languages. An example of a language that can be generated by a RCG but that is not semilinear is the language from Figure 2.20.

RCGs are equivalent to a restricted form of *Literal Movement Grammars* (LMGs) (Groenink, 1996), so-called *simple* LMGs. These grammars have rewriting rules that are like the ones in RCG with the additional constraints that (i) the arguments in the right-hand sides of the clauses are single variables, (ii) no variable appears more than once in the left-hand side of a clause and (iii) each variable occurring in the right-hand side of a clause occurs also

RCG for the language $\{a^{2^n} \mid n \geq 0\}$:

$$\begin{aligned} S(XY) &\rightarrow S(X)eq(X, Y) \\ S(a) &\rightarrow \varepsilon \\ eq(aX, aY) &\rightarrow eq(X, Y) \\ eq(a, a) &\rightarrow \varepsilon \end{aligned}$$

A sample derivation (reduction to ε) for $w = aaaa$:

$$\begin{array}{c} S(X \quad Y) \rightarrow S(X) \quad eq(X, \quad Y) \\ \downarrow \quad \downarrow \quad \quad \downarrow \quad \downarrow \quad \downarrow \\ \langle 0, 2 \rangle \quad \langle 2, 4 \rangle \quad \langle 0, 2 \rangle \quad \langle 0, 2 \rangle \quad \langle 2, 4 \rangle \\ aa \quad aa \quad aa \quad aa \quad aa \\ \text{With this instantiation, } S(\langle 0, 4 \rangle) \Rightarrow S(\langle 0, 2 \rangle)eq(\langle 0, 2 \rangle, \langle 2, 4 \rangle). \\ S(X \quad Y) \rightarrow S(X) \quad eq(X, \quad Y) \quad \quad S(a) \rightarrow \varepsilon \quad \quad eq(a, \quad a) \rightarrow \varepsilon \\ \downarrow \quad \downarrow \quad \quad \downarrow \quad \downarrow \quad \downarrow \quad \quad \downarrow \quad \downarrow \quad \downarrow \\ \langle 0, 1 \rangle \quad \langle 1, 2 \rangle \quad \langle 0, 1 \rangle \quad \langle 0, 1 \rangle \quad \langle 1, 2 \rangle \quad \quad \langle 0, 1 \rangle \quad \quad \langle 0, 1 \rangle \quad \langle 1, 2 \rangle \\ a \quad a \quad a \quad a \quad a \quad \quad a \quad \quad a \quad a \\ \text{leads to } S(\langle 0, 2 \rangle) \Rightarrow S(\langle 0, 1 \rangle)eq(\langle 0, 1 \rangle, \langle 1, 2 \rangle) \xRightarrow{*} \varepsilon \\ eq(a \quad X \quad a \quad Y) \rightarrow eq(X, \quad Y) \quad \quad eq(a, \quad a) \rightarrow \varepsilon \\ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \quad \downarrow \quad \downarrow \quad \quad \downarrow \quad \downarrow \\ \langle 0, 1 \rangle \quad \langle 1, 2 \rangle \quad \langle 2, 3 \rangle \quad \langle 3, 4 \rangle \quad \langle 1, 2 \rangle \quad \langle 3, 4 \rangle \quad \quad \langle 1, 2 \rangle \quad \langle 3, 4 \rangle \\ a \quad a \quad a \quad a \quad a \quad a \quad \quad a \quad a \\ \text{leads to } eq(\langle 0, 2 \rangle, \langle 2, 4 \rangle) \Rightarrow eq(\langle 1, 2 \rangle, \langle 3, 4 \rangle) \Rightarrow \varepsilon \end{array}$$

Fig. 2.20. RCG for $\{a^{2^n} \mid n \geq 0\}$

in its left-hand side. In contrast to RCG, an instantiation in a LMG maps variables to strings of terminals. Consequently, the terminals in a clause need not have corresponding terminals in the input and different occurrences of the same variable can be mapped to different occurrences of the same string. This is why with this restricted form of clauses one obtains a grammar formalism with the same generative capacity as RCGs.

2.3 Summary

In this chapter, we have given an overview of the different grammar formalisms that we will deal with in the course of this book.

The starting point was the observation that CFGs do not have enough expressive power to deal with natural languages. A formal proof of this fact has been given by Shieber (1985), showing that Swiss German is not context-free because of its cross-serial dependencies. Shieber was able to make an argument even on the basis of the weak generative capacity since Swiss German has case marking and therefore dependencies are visible even in the string language.

Another property that has been argued as being desirable for an adequate grammar formalism for natural languages is lexicalization. It has been shown that in general, CFGs cannot be strongly lexicalized.

From these shortcomings arises the need for more powerful formalisms. This has led to a rich variety of grammar formalisms that can be seen as more and more extending the properties of context-free grammars. In TAG, we allow not only replacing leaves with new trees as in CFG but we also allow internal nodes to be replaced with new trees. In LCFRS, we allow the yields of non-terminals to consist not only of single strings but of tuples of non-adjacent strings. In RCG, we even allow strings to be used several times in different contexts. All these grammar frameworks, and their respective equivalent formalisms, constitute a hierarchy of string languages as shown in Figure 2.21.

A notion that has proved an important concept in the characterization of grammar formalisms with respect to their relevance for natural languages is the notion of mild context-sensitivity, introduced by Joshi (1985). A class of languages is mildly context-sensitive if it contains all context-free languages, if it can describe cross-serial dependencies, if it contains only polynomial languages and if its languages are of constant growth. The language classes of TAG and of LCFRS in our hierarchy are mildly context-sensitive.

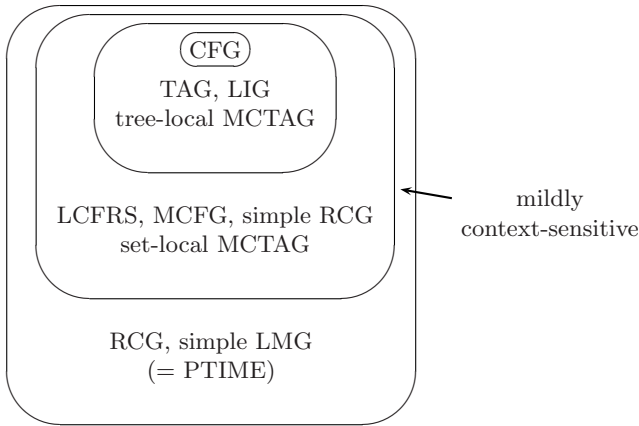


Fig. 2.21. The language hierarchy of the different grammar formalisms

Problems

2.1. Consider the language $L_2 = \{a^n b^n \mid n \geq 0\}$.

1. Give a CFG for L_2 with nested dependencies, i.e., such that for each word $a_1 \dots a_n b_1 \dots b_n$ (the subscripts mark the occurrences of the a s and b s respectively) a_i and b_{n+1-i} are added in the same derivation step for all $1 \leq i \leq n$.

2. Show that for L_2 there is no CFG displaying cross-serial dependencies, i.e., no CFG such that for each word $a_1 \dots a_n b_1 \dots b_n$, a_i and b_i are added in the same derivation step for all $1 \leq i \leq n$ and, furthermore, different a s are added in different derivation steps.

2.2. Similar to the argument of Shieber (1985) for Swiss German, one can apply first a homomorphism f , then intersect the result with some regular language, and then apply another homomorphism g in order to reduce the language of Swiss German to the copy language $\{ww \mid w \in \{a, b\}^*\}$. Find the corresponding homomorphisms and the regular language.

2.3. Consider the following CFG:

$S \rightarrow NP VP$ $NP \rightarrow \text{John}$
 $VP \rightarrow ADV VP$ $ADV \rightarrow \text{always}$
 $VP \rightarrow V$ $V \rightarrow \text{laughs}$

Find a TSG that strongly lexicalizes this grammar.

Why is this lexicalization not satisfying from a linguistic point of view?

- 2.4.** 1. Show that the copy language $\{ww \mid w \in T^*\}$ for some alphabet T is semilinear using the Parikh Theorem.
2. Show that $\{a^{2^n} \mid n \geq 0\}$ is not semilinear.



<http://www.springer.com/978-3-642-14845-3>

Parsing Beyond Context-Free Grammars

Kallmeyer, L.

2010, XII, 248 p. 76 illus., Hardcover

ISBN: 978-3-642-14845-3