UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Feature grammar systems. Incremental maintenance of indexes to digital media warehouses

Windhouwer, M.A.

Publication date
2003

Link to publication

Citation for published version (APA):
Windhouwer, M. A. (2003). *Feature grammar systems. Incremental maintenance of indexes to digital media warehouses*. [Thesis, externally prepared, Universiteit van Amsterdam].

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:18 Nov 2026

Chapter 2

Feature Grammar Systems

A wise man once said
that everything could be explained with mathematics
He has denied
His feminine side
Now where is the wisdom in that?

Marillion – *This is the 21st century*

The core of incremental maintenance of the annotation index lies in the understanding and analysis of the dependency description. In this chapter the dependency descriptions used by the Acoi system architecture, called feature grammar systems, are introduced.

As the name suggests feature grammar systems are related to grammars known from natural languages, *e. g.* English or Dutch. Sentences in these languages are constructed from basic building blocks: words. Each language has a set of rules which describe how words can be put in a valid order, and what the semantic meaning of this order is. These rules form the grammar of the language, *e. g.* like the following grammar rules for a subset of the English language.

Example 2.1.

$$S \rightarrow NP\,VP$$

$$NP \rightarrow John$$
$$NP \rightarrow Mary$$

$$VP \rightarrow V_i$$
$$VP \rightarrow V_t\,NP$$

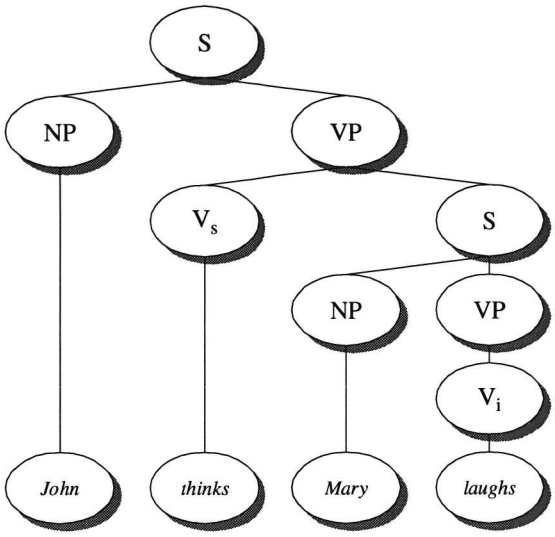


Figure 2.1: A parse tree

$$VP \rightarrow V_s \; S$$

$$V_i \rightarrow laughs$$
$$V_t \rightarrow loves$$
$$V_s \rightarrow thinks$$

□

Sentences can now be tested for membership of the language induced by these specific grammar rules. A valid sentence S consists of two parts: a noun phrase NP and a verb phrase VP. Each of these phrases may (again) consist of other parts, *e. g.* a specific verb type. This can be repeated until the individual words of the sentence are reached. The result of such a process, also known as *parsing*, is the *parse tree* of a sentence. Figure 2.1 shows the parse tree for this sentence: *John thinks Mary laughs*. Notice that the complete parsing context is captured by this tree.

The fundamental idea behind feature grammar systems is that the same process of deriving a parse tree, and thus the underlying formal theories and practices, can be used as a driving force to produce the annotation of a multimedia object. To start once more with the basis: annotation items are seen as the words. Sentences formed by combinations of these words should be valid in a feature grammar system, *i. e.* a specialized dependency description.

As stated in the previous chapter two kinds of dependencies should be captured in these descriptions: output/input and contextual dependencies. Looking at the structure of the grammar rules it is easy to see that the right-hand sides of the rules capture contextual dependencies: in the context of a specific verb phrase VP a verb V_t should always be followed by a noun phrase NP. Output/input dependencies are directly related to annotation extraction algorithms. But those are not found in the grammar rules. The addition of feature grammar systems is that these algorithms are bound to specific symbols in the grammar. Upon encountering such a special symbol during the derivation process the output/input dependencies can be resolved by using the contexts stored in the gradually build parse tree. In fact the output/input dependencies are associated with the context of the left-hand side of a rule.

The first part of this chapter is devoted to embedding the additions of feature grammar systems into results of formal language theory. To support the understanding of this embedding the next section shortly recalls some relevant grammar theory. Readers which are familiar with grammar theory, *i. e.* the Chomsky hierarchy, the (regulated) rewriting process and grammar systems, may skip to Section 2.2.

A feature grammar itself is a valid sentence in a meta language: the *feature grammar language*. The next chapter describes how the expressive power of feature grammar systems is captured by the feature grammar language. Using this theoretical basis Chapters 4 and 6 will describe appropriate adaptations of formal language technologies used by the annotation subsystem to maintain the database. This database, as will be described in Chapter 5, stores a collection of the annotation sentences, *i. e.* a subset of all possible sentences in the language induced by the feature grammar, along with their parse trees.

2.1 A Grammar Primer

Grammars are a key concept in computer science and many theoretical and practical issues related to them have been studied extensively. The theoretical implications of grammars are studied in the field of formal language theory (see [Lin97, HMU01] for an introduction) and form the basis of this chapter. Parsing algorithms (see [GJ98] for an overview), *i. e.* how to efficiently determine if a sentence is a valid member of a language, is one of the more practical issues and will play an important role in Chapter 4.

2.1.1 A Formal Specification of Languages and Grammars

The formal specification starts with a language L. L consists of sentences constructed by concatenation of symbols from the alphabet Σ, *i. e.* $L \subseteq \Sigma^*$. A grammar G describes the language $L(G)$ and is defined as follows.

Definition 2.1. *A grammar G is defined as a quadruple $G = (N, T, P, S)$, where*

1. N *is a, non-empty, finite set of symbols called* non-terminals *or* variables,
2. T *is a, possibly empty, finite set of symbols called* terminals, *i. e.* $T \subseteq \Sigma$,
3. $N \cap T = \emptyset$,
4. $V = N \cup T$,
5. P *is a finite set of rules of the form* $(L \rightarrow R)$, *called* productions, *such that*
   (a) $L \in V^+$ *is the left-hand side (LHS) of a production and*
   (b) $R \in V^*$ *is the right-hand side (RHS) of a production, and*
6. $S \in N$ *is a special symbol called the* start *variable or* axiom.

□

A production rule in P where the RHS is an empty string is written as: $(L \rightarrow \lambda)$, *i. e.* λ represents the empty string. Such a production rule is also called an erasing production.

The Chomsky Hierarchy

Grammars which apply to Definition 2.1 are called *recursively enumerable* (RE) or Type 0 grammars. These grammars permit the description of a large set of languages[1], however, they are also unmanageable, *e. g.* there is no general efficient parsing algorithm. This problem led to a key work in formal language theory: the Chomsky hierarchy of grammars [Cho59]. In this hierarchy grammars are organized on the basis of their expressive power. The hierarchy starts with phrase structure grammars, and on each subsequent level restrictions are added. The restrictions result in gradually easier to "understand" or to parse grammars, but these grammars become also gradually less expressive. The following other grammar types belong to the Chomsky hierarchy [GJ98].

Context-sensitive (CS) or Type 1 grammars A grammar is context-sensitive if each production rule is context-sensitive. A rule is context-sensitive if actually only one (non-terminal) symbol in its LHS gets replaced by other symbols, while the others are found back undamaged and in the same order in the RHS. This rule is for example context-sensitive, where L is the left and R is the right context of S:

$$L\,S\,R \rightarrow L\,W\,R$$

[1] [Lin97] contains a proof that there are languages which are not in $\mathcal{L}(RE)$.

Context-free (CF) or Type 2 grammars A context-free grammar is like a context-sensitive grammar, except that it may contain only rules that have a single non-terminal in their LHS, *i. e.* there are no left and right contexts as shown in this rule:

$$S \rightarrow X\ Y\ Z$$

Regular (REG) or Type 3 grammars A regular grammar contains two kinds of production rules: (1) a non-terminal produces zero or more terminals and (2) a non-terminal produces zero or more terminals followed by one non-terminal. Examples for both kinds of rules are shown here:

$$S \rightarrow x\ y\ z$$
$$S \rightarrow v\ W$$

Due to the specific forms of rules in a REG grammar a more compact notation, a regular expression, is often used. Next to the alphabet T the notation supports: parentheses and the operators union ($+$), concatenation ($\cdot$) and star-closure ($*$). For example, the expression $((a+b\cdot c)^*)$ stands for the star-closure of $\{a\} \cup \{bc\}$, that is, the language $\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, \ldots\}$. More extended regular expression languages, and thus more compact, are used in practice [Fri02]. As a convenience for later on the *period* operator (.) is already introduced. This operator matches any symbol of the alphabet, *i. e.* $(.\cdot a\cdot .)$ describes the language where the second symbol is always a, while the first and third symbol may be any symbol (including a).

Mild Context-sensitivity

Just like RE grammars, CS grammars turned out to be impractical. Although a generic parsing algorithm can be defined, *i. e.* in the construction of a linear bounded automaton (LBA) [Lin97, RS97a], specifying a CS grammar remains a non-trivial task, even for a small language, resulting in incomprehensible grammars (see for an example [GJ98]). For this reason most practical attention has gone to CF and REG grammars. However, it was early discovered that "the world is not context-free", *i. e.* there are many circumstances where naturally non-CF languages appear. Linguists seem to agree [Man94] that "all" natural languages contain constructions which cannot be described by CF grammars. Three of such non-CF constructions are [RS97b]:

1. *reduplication*, leading to languages of the form $\{xx | x \in V^*\}$;
2. *multiple agreements*, modeled by languages of the form $\{a^n b^n c^n | n \geq 1\}$, $\{a^n b^n c^n d^n | n \geq 1\}$, etc.;

3. *crossed agreements*, as modeled by $\{a^n b^m c^n d^m | n, m \geq 1\}$.

Artificial languages, *e. g.* a programming language like *Algol 60*, have also non-CF properties [Flo62]. Seven examples of non-CF areas where languages are found are described in Section 0.4 of [DP89], and the section also concludes with the remark: "the world seems to be non-context-free ...". The same book describes 25 different mechanisms for regulated rewriting. Using such a mechanism a "mild" subfamily of CS languages is created. A mild CS language has as many CF-like properties as possible, but is able to cover the required non-CF constructions.

Before some mechanisms of regulated rewriting are introduced the derivation process (also known as rewriting) for CF and REG grammars is formalized.

2.1.2 The Derivation Process

If the sentence $w \in L(G)$, then the following derivation exists:

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$$

The strings $S, w_1, w_2, \ldots, w_n$, which may contain variables as well as terminals, are called *sentential forms* of the derivation.

This application of the direct derivation can be defined as follows:

Definition 2.2. *The application of the direct derivation $x_1 w x_2 \Rightarrow x_1 z x_2$ is allowed iff $(w \rightarrow z) \in P$, where P is the set of productions from grammar G.*

□

By using a subscript to the direct derivation ($\Rightarrow$) a specific grammar or a (labeled) production rule may be indicated, *e. g.* $\Rightarrow_G$.

Using the transitive closure of the direct derivation the set of sentences in the language L can be defined:

$$L(G) = \{w \in T^* | S \stackrel{*}{\Rightarrow} w\}$$

This set of sentences may be further limited by specific forms of derivation. A practical form, which will be encountered later on, is *leftmost* derivation. In this case each rule used in the derivation process rewrites the leftmost non-terminal in the current sentential form, *i. e.* $x_1 \in T^*$. This specific mode of derivation is indicated as follows:

$$L(G) = \{w \in T^* | S \underset{lm}{\stackrel{*}{\Rightarrow}} w\}$$

Bidirectional Grammars

Until now grammars are used in generating mode, *i. e.* by starting with the start symbol a specific sentence is gradually constructed. Using this process all sentences belonging to the language can be enumerated, hence all grammars are RE grammars. Grammars can also be used in accepting mode. In this mode a specific sentence is checked for membership of the language. This is done by flipping the left- and right-hand sides of the production rules. This derivation process then describes the acceptance of the sentence w: $w \stackrel{*}{\Rightarrow} S$.

The mode of the grammar is indicated by G^{gen} or G^{acc} for respectively generative and accepting mode. When no mode is indicated (G) the generative mode is used. A specific grammar may be usable in both modes and is then called *bidirectional* [App87, Neu91].

Notice that membership of a sentence can be resolved in either accepting or generating mode. In the latter case one enumerates all sentences until the sentence is found (or not), although this process may be endless. The optimization of this search process is the main target of parsing algorithms, which will be discussed in Chapter 4.

Parse Trees

By keeping track of the derivation steps a parse tree of the sentence, like the one in Figure 2.1, can be build. The parse trees for G are trees with the subsequent conditions [HMU01]:

1. each interior node is labeled by a non-terminal in N;
2. each leaf is labeled by either a non-terminal, a terminal, or λ, however, if the leaf is labeled λ, then it must be the only child of its parent;
3. if an interior node is labeled A, and its children are labeled

$$X_1, X_2, \ldots, X_k$$

respectively, from the left, then $A \rightarrow X_1 X_2 \ldots X_k$ is a production in P. Note that the only time one of the X's can be λ is if that is the label of the only child, and $(A \rightarrow \lambda)$ is a production of G.

The Delta Operation

Each sentential form w_i can now be associated with a parse tree t_i. The yield of this tree is defined as the corresponding sentential form, *i. e.* $yield(t_i) = w_i$. A tree can also be described by a set of paths, *i. e.* the result of $path(t_i)$. The in the beginning of the chapter shown parse tree (see Figure 2.1) contains, for example, the

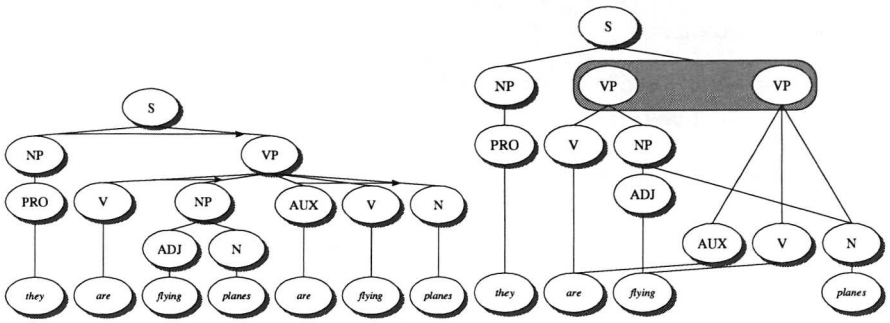


Figure 2.2: (a) An AND/OR-graph and (b) a packed shared forest

path $S_1 \cdot VP_2 \cdot V_1 \cdot thinks_1$. The subscripts indicate the order of the nodes among its siblings in the tree[2].

The *path* and *yield* operations provide a simple relationship between REG and CF languages [Eng02]. For a language L, let the *delta* of L, denoted by $\delta(L)$, be the language of all yields of trees that have all their paths in L:

$$\delta(L) = \{yield(t) | path(t) \subseteq L\}$$

The relationship mentioned above is that the CF languages are exactly the deltas of the REG languages: $\mathcal{L}(\text{CF}) = \{\delta(L) | L \in \mathcal{L}(\text{REG})\}$.

Parse Forests

Definition 2.1 allows multiple production rules to have the same symbol as their LHS. The RHSs of these rules are considered alternatives of each other. Grammars containing alternatives are called non-deterministic. If several of these alternatives lead to a valid parse tree for the input sentence, the grammar is also ambiguous. In these cases the sentence is not described by one parse tree, but by several, *i. e.* a parse forest. Such a forest can be represented by an AND/OR-graph [Nil98, Hal73]. In these graphs conjunctions of nodes are connected by an arc, see Figure 2.2.a. In this example the last three words of the input sentence can be explained by two (not shown in the simple example grammar for the English language) alternative production rules for the verb phrase VP. In one parse tree $flying$ is an adjective to the noun $planes$, in the other structure $flying$ is interpreted as a verb.

A packed shared forest [Tom86], see Figure 2.2.b, aims at structure sharing. Parsing algorithms (to be discussed in more detail in Chapter 4) do easily allow *sharing* of substructures by a well known dynamic programming concept: memoization [Mic68]. The $noun$ node N and most of the terminal nodes are shared this way. This can be

[2]For brevity's sake the order information in paths will be omitted most of the time.

seen as sharing the bottom structures of the parse trees. *Packing* is targeted at sharing the upper structures. A *packed node* contains *sub-nodes* which have common leaf nodes and are labeled with the same non-terminal symbol. In the example structure the two alternatives for the verb phrase are packed together.

Packed shared forests are aimed at parse structures related to CF grammars. This grammar type only allows *local* ambiguity. The combination of ambiguity and the long distance dependencies of CS grammars calls for methods to indicate the global context and scope of a node. One method to achieve this is by naming the disjunctions and to annotate nodes in the same context with this name, see [DE90, Bla97, Bla98].

Disambiguation

In fact ambiguity may appear on many different levels in an application. In *natural language processing* (NLP) ambiguity can be found at these levels:

1. *lexical ambiguity*: a word has more than one meaning;
2. *syntactic or structural ambiguity*: a sentence has two or more parses as in Figure 2.2;
3. *semantic ambiguity*: may directly follow from syntactic ambiguity or from the semantic context.

The NLP community has introduced a vast amount of models and algorithms to *disambiguate* these ambiguities. For an overview see [JM00].

On the syntactic level disambiguation may be done by the use of *probabilistic parsing* [Boo69, Sol69]. In this case each alternative is assigned a probability [Bak79] and the parsing algorithm [Jel69] chooses the most probable parse tree. Another method is to ask for human interaction. Tomita describes a system where manual disambiguation is build into the parser [Tom86]. However, it is also possible to postpone disambiguation to a higher automatic or manual level. In which case the parser will have to deliver the complete ambiguous structure. For example, in [KV94, vdBKMV03] the authors describe disambiguation filters, based on term rewriting, to prune a parse forest.

2.1.3 Regulated Rewriting

By regulating the applicability of a direct derivation step the set of valid sentences of a language can be decreased, while the basic rules keep close to their CF equivalents. In the monograph [DP89] the authors describe 25 regulation mechanism, *e. g.* matrix, programmed and random context grammars. In this section two mechanisms, which are closely related to the one applied by feature grammar systems, are described in detail.

Conditional Grammars

While other mechanisms work by restricting or explicitly stating the order in which productions may be applied the mechanism of *conditional* (C) grammars is based on conditions on the contextual sentential forms.

Definition 2.3. *In a* conditional grammar $G = (N, T, P, S)$ *the productions* P *are of the form* $(w \to z, Q)$ *where*

- Q *is a* REG *language over the alphabet* V,
- N, T *and* S *have their normal meaning.*

The rule $(w \to z, Q)$ *is applicable to* $x = x_1 w x_2$ *yielding* $y = x_1 z x_2$, *i. e.* $x \Rightarrow y$, *iff* $x \in Q$.

□

Example 2.2.

Take for example the following C grammar:

$$G = (\{S, S'\}, \{a\}, \{p_1, p_2, p_3\}, S)$$

with

$$p_1 = (S \to S'S', (S')^* S^+)$$
$$p_2 = (S' \to S, S^*(S')^+)$$
$$p_3 = (S \to a, a^* S^+)$$

[RS97b] shows that this grammar describes a known non-CF language:

$$L(G) = \{a^{2^n} | n \geq 0\}$$

□

In both [DP89] and [RS97b] it is shown that $\mathcal{L}(\text{C}, \text{CF} - \lambda) = \mathcal{L}(\text{CS})$, *i. e.* C grammars with CF rules but no erasing productions are equivalent to CS grammars, and also that $\mathcal{L}(\text{C}, \text{CF}) = \mathcal{L}(\text{RE})$. In [RS97b] this is done by giving rules for transforming a C grammar into a CS grammar, and vice versa.

Tree-controlled Grammars

Another regulated rewriting mechanism uses the parse tree to restrict applications of the derivation step. Those grammars, called *tree-controlled* (TC) grammars, are defined as follows [CM77, DP89]:

Definition 2.4. *A* tree-controlled *grammar is a construct* $G = (N, T, P, S, R)$ *where*

- $G' = (N, T, P, S)$ *is a CF grammar and*
- $R \subseteq V^*$ *is* regular.

$L(G)$ *consists of all words* w *generated by the underlying grammar* G' *such that there is a parse tree of* w *such that each word obtained by concatenating all symbols at any level (except the last one) from left to right is in* R.

□

All nodes of the parse tree with the same distance to the root of the tree are on the same level of the derivation tree.

Example 2.3.

Consider the following grammar:

$$G = (\{S, A, B, C\}, \{a, b, c\}, \{P_1, \ldots, P_7\}, S, R)$$

with

$$\begin{aligned} P_1 &= (S \to ABC) \\ P_2 &= (A \to aA) \\ P_3 &= (A \to a) \\ P_4 &= (B \to bB) \\ P_5 &= (B \to b) \\ P_6 &= (C \to cC) \\ P_7 &= (C \to c) \end{aligned}$$

and

$$R = \{S, ABC, aAbBcC\}.$$

Evidently,

$$L(G) = \{a^n b^n c^n | n \geq 1\}.$$

Which is a known CS language [CM77].

□

Also for this type of mildly CS languages it is shown in [DP89] that $\mathcal{L}(\text{TC, CF} - \lambda) = \mathcal{L}(\text{CS})$.

2.1.4 Grammar Systems

Until now only one grammar at a time is considered. However, grammars can cooperate in so called grammar systems [CVDKP94]. The theory of these systems was inspired by the will to model *multi-agent systems*. A common technique in the field of Artificial Intelligence (AI) is to structure those systems according to the blackboard architecture [Nil98]. In such a system various knowledge sources work together on solving a problem. The current state of the solution resides on the blackboard. A protocol of cooperation encodes the control over the knowledge sources.

In a grammar system the common sentential form is on the blackboard and the component grammars are the knowledge sources. Since its introduction in 1990 various forms of cooperation and control have been studied. Two basic classes are distinguished: *cooperating distributed* (CD) and *parallel communicating* (PC) grammar systems. In this section only CD grammar systems will be introduced, as they form the formal basis for feature grammar systems.

CD Grammar Systems

Definition 2.5. *A* CD grammar system *is a* $(n+2)$*-tuple*

$$\Gamma = (T, G_1, G_2, \ldots, G_n, S),$$

where,

1. *for* $1 \leq i \leq n$, *each* $G_i = (N_i, T_i, P_i)$ *is a (usual)* CF grammar, *called a* component, *with*
   (a) *the set* N_i *of* non-terminals,
   (b) *the set* T_i *of* terminals,
   (c) $V_i = N_i \cup T_i$,
   (d) *the set* P_i *of* CF rules, *and*
   (e) *without* axiom,
2. T *is a subset of* $\bigcup_{i=1}^{n} T_i$,
3. $V = \bigcup_{i=1}^{n} V_i$, *and finally*
4. $S \in \bigcup_{i=1}^{n} N_i = N$.

□

The components correspond to the knowledge sources solving the problem on the blackboard, where every rule represents a piece of knowledge which results in a possible change of the blackboard. The axiom, or start symbol, S corresponds with

the initial state of the problem on the blackboard. The alphabet T corresponds to knowledge pieces which are accepted as solutions or parts of solutions.

A derivation step in a grammar system is now defined as follows:

Definition 2.6. *Let* Γ *be a* CD grammar system *as in Definition 2.5. Let* $x, y \in V_i^*$. *Then* $x \Rightarrow_{G_i}^k y$ *is applicable iff there are words* $x_1, x_2, \ldots, x_{k+1}$ *such that:*

1. $x = x_1, y = x_{k+1}$,
2. $x_j \Rightarrow_{G_i} x_{j+1}$, *i. e.* $x_j = x_j' A_j x_j'', x_{j+1} = x_j' w_j x_j'', (A_j \to w_j) \in P_i, 1 \leq j \leq k$.

Moreover, this leads to the following other derivation modes:

- $x \Rightarrow_{G_i}^{\leq k}$ *iff* $x \Rightarrow_{G_i}^{k'} y$ *for some* $k' \leq k$,
- $x \Rightarrow_{G_i}^{\geq k}$ *iff* $x \Rightarrow_{G_i}^{k'} y$ *for some* $k' \geq k$,
- $x \Rightarrow_{G_i}^{*}$ *iff* $x \Rightarrow_{G_i}^{k} y$ *for some* k, *and*
- $x \Rightarrow_{G_i}^{t}$ *iff* $x \Rightarrow_{G_i}^{*} y$ *and there is no* $z \neq y$ *with* $y \Rightarrow_{G_i}^{*} z$.

□

Any derivation $x \Rightarrow_{G_i}^k y$ corresponds to k direct derivation steps in succession in the component grammar G_i. In a $\leq k$-derivation mode the component can perform at most k changes. The $\geq k$-mode requires the component to be competent enough to perform at least k steps. A component may work on the problem as long as it wants when the derivation is in $*$-mode. Finally, the t-mode corresponds to the case where the component should work on the problem as long as it can.

The language induced by a CD grammar system Γ is now defined as follows:

Definition 2.7. *Let*

$$f \in \{*, t, 1, 2, \ldots, \leq 1, \leq 2, \ldots, \geq 1, \geq 2, \ldots\},$$

and let Γ *be a* CD grammar system. *Then the language* $L_f(\Gamma)$ *generated by* Γ *is defined as the set of all words* $z \in T^*$ *for which there is a derivation*

$$S = w_0 \Rightarrow_{G_{i_1}}^f w_1 \Rightarrow_{G_{i_2}}^f w_2 \Rightarrow_{G_{i_3}}^f \cdots \Rightarrow_{G_{i_r}}^f w_r = z.$$

□

Finally, the choice of the "terminal" set of a CD grammar system may be restricted. A CD grammar system as specified in Definition 2.5 accepts in

$style(arb)$ iff T is an arbitrary subset of $\bigcup_{i=1}^n T_i$,

$style(ex)$ iff $T = \bigcup_{i=1}^{n} T_i$,

$style(all)$ iff $T = \bigcap_{i=1}^{n} T_i$,

$style(one)$ iff $T = T_i$ for some $i, 1 \leq i \leq n$.

Now (CD_nCF, f, A) denotes a class of CD grammar systems with at most n components working in the f-mode of derivation and accepting in style A, where CF denotes that CF component grammars are used. $(CD_\infty CF, f, A)$ indicates a CD grammar system with an arbitrary number of components.

Internal Control

Just like with normal grammars regulation mechanisms can be added to restrict the application of derivation steps. This may take the form of either external or internal control. With external control a supervisor is added or a sequence of components is fixed in advance, *e. g.* by paths in a directed graph. Internal control makes use of conditions on the current state of the problem, *i. e.* the sentential form. These conditions can be used to either start or stop a component. As internal control is used by feature grammar systems only this form of control is further investigated.

Definition 2.8. *A* dynamically controlled *CD grammar system* Γ *is a grammar system as in Definition 2.5 with* $G_i = (N_i, T_i, P_i, \pi_i, \rho_i)$ *where*

- π_i *is a* start condition, *and*
- ρ_i *is a* stop condition *for component* G_i.

Then the language $L(\Gamma)$ *generated by* Γ *is defined as the set of all words* $z \in T^*$ *for which there is a derivation*

$$S = w_0 \Rightarrow_{G_{i_1}}^{*} w_1 \Rightarrow_{G_{i_2}}^{*} w_2 \cdots \Rightarrow_{G_{i_r}}^{*} w_r = z$$

such that, for $1 \leq j \leq r$,

$$\pi_{i_j}(w_{j-1}) = true \text{ and } \rho_{i_j}(w_j) = true$$

and for $f \in \{t, 1, 2, \ldots, \leq 1, \leq 2, \ldots, \geq 1, \geq 2, \ldots\}$, *the language* $L_f(\Gamma)$ *generated by* Γ *in the* f*-mode as the set of all words* $z \in T^*$ *such that there is a derivation*

$$S = w_0 \Rightarrow_{G_{i_1}}^{f} w_1 \Rightarrow_{G_{i_2}}^{f} w_2 \cdots \Rightarrow_{P_{i_r}}^{f} w_r = z$$

such that, for $1 \leq j \leq r$, $\pi_{i_j}(w_{j-1}) = true$.

□

Notice that when the derivation is not in $*$-mode the stop condition $\rho_{i_j}(w_j) = true$ is replaced by the stop condition which is naturally given by the f-mode.

Some special types of conditions have been studied for CD grammar systems. Condition σ may be of these types:

$type(a)$ iff $\sigma(w) = true$ for all $w \in V^*$,

$type(rc)$ iff there are two subsets R and Q of V and $\sigma(w) = true$ iff w contains all letters of R and w contains no letter of Q,

$type(K)$ iff there are two words x and x' over V and $\sigma(w) = true$ iff x is a subword of w and x' is not a subword of w,

$type(K')$ iff there are two finite subsets R and Q of V^* and $\sigma(w) = true$ iff all words of R are subwords of w and no word of Q is a subword of w,

$type(C)$ iff there is a regular set R over V and $\sigma(w) = true$ iff $w \in R$.

Notice that $type(C)$ corresponds with the conditions known from C grammars.

In the notation for grammar systems the f is now replaced by (X, Y), where X indicates the start condition type and Y the same for the stop condition, when the grammar uses $*$-mode. In the other derivation modes (X, f) is used, for example $(CD_8CF, (rc, t), arb)$.

Many more variants of CD grammar systems have been studied in the literature, including the use of bidirectional components [FH99], and the monograph [CVDKP94] discusses a lot of them. Some variants are combined to form the basis for the concept of feature grammar systems, and will be described during the formal definition of feature grammar systems in the next section.

2.2 Feature Grammar Systems

Lets, while gradually working toward a formal definition of feature grammar systems[3] using the building blocks from the previous section, return to the annotation example from Chapter 1. As stated in the introduction of this chapter the annotation items can be seen as words in a sentence, see Figure 2.3. A possible parse tree for this sentence is also shown in the same figure.

This parse tree is the result of a derivation process driven by the this grammar:

[3] In NLP the term feature grammars is sometimes used to indicate a type of grammar formalisms, *e. g.* HPSG [PS94] and LFG [Bre82], where next to a *c*-structure, *i. e.* a parse tree, also a f-structure is constructed. The f-structure contains (semantic) features which are propagated up in the tree using unification rules. Unification leads in the end to one combined f-structure for the root of the *c*-structure. Unfortunately this type of grammars was only encountered by the author when the concept of feature grammars, where the term feature refers to multimedia features, was already introduced to the multimedia and database research communities.

Example 2.4.

$$
\begin{aligned}
\mathit{Image} &\rightarrow \mathit{Location}\ \mathit{Color}\ \mathit{Class} \\
\mathit{Location} &\rightarrow \mathit{url} \\
\\
\mathit{Color} &\rightarrow \mathit{Number}\ \mathit{Prevalent}\ \mathit{Saturation} \\
\mathit{Number} &\rightarrow \mathit{int} \\
\mathit{Prevalent} &\rightarrow \mathit{flt} \\
\mathit{Saturation} &\rightarrow \mathit{flt} \\
\\
\mathit{Class} &\rightarrow \mathit{Graphic} \\
\mathit{Class} &\rightarrow \mathit{Photo}\ \mathit{Skin}\ \mathit{Faces} \\
\mathit{Graphic} &\rightarrow \mathit{bit} \\
\mathit{Photo} &\rightarrow \mathit{bit} \\
\mathit{Skin} &\rightarrow \mathit{bitmap} \\
\mathit{Faces} &\rightarrow \mathit{int} \\
\\
\mathit{url} &\rightarrow \mathit{http://\ldots} \\
\mathit{int} &\rightarrow \mathit{1} \\
\mathit{int} &\rightarrow \mathit{29053} \\
\mathit{flt} &\rightarrow \mathit{0.3} \\
\mathit{flt} &\rightarrow \mathit{0.19} \\
\mathit{bit} &\rightarrow \mathit{true} \\
\mathit{bitmap} &\rightarrow \mathit{00\ldots}
\end{aligned}
$$

□

In the upcoming sections this CF grammar will be extended until the full fledged power of a feature grammar system is reached.

2.2.1 Detectors

The dependency description evolves around the annotation extraction algorithms. Before introducing how feature grammar systems capture both output/input and contextual dependencies, annotation extraction algorithms are formally introduced into the grammar.

In a feature grammar these algorithms are bound to specific non-terminals, called (feature) detectors. In the example grammar the set of detectors is { *Color*, *Graphic*,

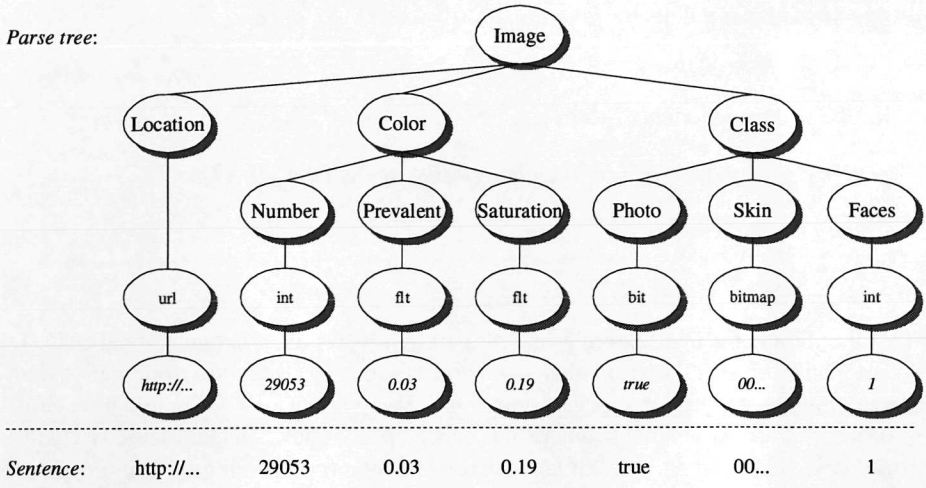


Figure 2.3: Annotation sentence

Photo, *Skin*, *Faces* }. Each of these detector algorithms can be seen as a function, which transforms its input sentence into an output sentence. The *Faces* detector, for example, transforms the sentence *00* ... into the sentence *1*. The output sentence of each of these detectors is conveniently described by a set of CF grammar productions, *i. e.* each detector corresponds to a component in a CD grammar system.

The following definition describes the addition of feature detectors to CD grammar systems.

Definition 2.9. *A* basic feature grammar system *is a* $(n+6)$*-tuple*

$$\Gamma = (D, N, T, P_N, G_1, G_2, \ldots, G_n, G_S, S),$$

where,

1. $V = (D \cup N \cup T)$ *is the shared alphabet consisting of*
   (a) *the set* D *of* detectors, *containing at least* S_S,
   (b) *the set* N *of* non-terminals *and*
   (c) *the set* T *of* terminals,
2. P_N *contains* productions *of the form* $(N \rightarrow V^*)$,
3. *for each* $d_i \in D$, *i. e.* $n = |D|$, *there is a* $G_i = (V, P_i = P_{d_i} \cup P_N, f_i)$ *with*
   (a) *the set* P_{d_i} *consisting of* CF rules *of the form* $(d_i \rightarrow V^+)$,

(b) a partial detector *function* $f_i : T^* \rightarrow (D \cup T)^+$, *and*

(c) $\lambda \notin L(G_i)$,

4. *the start component* $G_S = (V, P_S = \{(S_S \rightarrow D \cup N)\} \cup P_N, f_S)$ *where*

   (a) f_S *is a function producing an initial sentence* $w_S \in (D \cup T)^+$,

5. *and finally* $S = S_S$.

□

All components in a feature grammar system always use the same alphabet V. In more practical terms: there is only one symbol table as all feature grammar system components share the same set of symbols. This variant of CD grammar systems is denoted as $(CD'_\infty CF, f)$, where the acceptation style is deleted as it is always $style(ex)$. This makes it possible to share semantically meaningful non-terminals. These non-terminals, which are not detectors, are used to group terminals. These semantic structures can be shared by different detectors and are put in the shared set of production rules P_N.

Each grammar component corresponds with one detector symbol. The detector function restricts the language accepted by a feature grammar system Γ in the following formal way:

Definition 2.10. *Let* Γ *be a feature grammar system as specified in Definition 2.9. Then the language* $L(\Gamma)$ *generated by* Γ *is defined as the set of all words* $w \in T^+$ *for which there is a derivation*

$$S \overset{*}{\Rightarrow}_\Gamma^* w_l d_i w_r \Rightarrow_{G_i}^* w_l f_i(w_l d_i w_r) w_r \overset{*}{\Rightarrow}_\Gamma^* w$$

Where f_i *is the* partial mapping function *associated with* detector d_i.

□

The moment a detector d_i is encountered the remainder of the derivation should replace this symbol by the output of the associated detector function $f_i(w_l d_i w_r)$. In other words f_i prescribes the sentence generated by this part of the grammar, and forms the stop condition of the component. The grammar component G_i mimics the unknown internal mapping of f_i and thus validates that the output of this detector conforms to the grammar[4]. This generated sentence functions as a stop condition of $type(C)$, *i. e.* a feature grammar system is a $(CD'_\infty CF, (a, C))$ grammar system.

Notice that the empty sentence (λ) is not allowed as output of a detector function, *i. e.* $\lambda \notin L(G_i)$. An empty sentence thus indicates the partiality of the detector function: the mapping is unknown for the input sentence.

[4] Notice that the grammar components are CF while the detector function itself may be more powerful, *i. e.* produce a sentence in a CS language like $a^n b^n c^n$. The grammar component will only be able to validate the CF envelope, *i. e.* $a^* b^* c^*$.

Definition 2.9 also introduces a dummy start symbol S_S. This special symbol is introduced to make it possible to have a "real" start symbol which is either a non-terminal or a detector. In the case of a detector the G_S component will stop directly after the application of the $(S_S \rightarrow d_i)$ rule, as it does not contain any rules for d_i, and control will be transfered to G_i. In the case of a non-terminal P_N helps to parse the initial sentence produced by f_S. How f_S produces this initial sentence is an implementation issue and will be discussed in Chapter 4.

The feature grammar system for Example 2.4 is constructed from these building blocks (the dots (...) indicates some omitted production rules from the example):

Example 2.5.

$$
\begin{aligned}
\Gamma = (& \\
D &= \{S_S, Color, Graphic, Photo, Skin, Faces\}, \\
N &= \{Image, Location, Number, Prevalent, Saturation, Class, \\
&\quad url, int, flt, bit, bitmap\}, \\
T &= \{http://\ldots, 1, 29053, 0.03, 0.19, true, 00\ldots\}, \\
P_N &= \{(Image \rightarrow Location\ Color\ Class), \ldots, (bitmap \rightarrow 00\ldots)\}, \\
G_{Color} &= (V, P_{Color} = \{(Color \rightarrow Number\ Prevalent\ Saturation)\} \\
&\quad \cup P_N, f_{Color}), \\
G_{Graphic} &= (V, P_{Graphic} = \{(Graphic \rightarrow bit)\} \cup P_N, f_{Graphic}), \\
G_{Photo} &= (V, P_{Photo} = \{(Photo \rightarrow bit)\} \cup P_N, f_{Photo}), \\
G_{Skin} &= (V, P_{Skin} = \{(Skin \rightarrow bitmap)\} \cup P_N, f_{Skin}), \\
G_{Faces} &= (V, P_{Faces} = \{(Faces \rightarrow int)\} \cup P_N, f_{Faces}), \\
G_S &= (V, P_S = \{(S_S \rightarrow Image)\} \cup P_N, f_S), \\
S &= S_S \\
&)
\end{aligned}
$$

□

The derivation process of the example sentence using this feature grammar looks as follows:

Example 2.6.

$$
\begin{aligned}
w_1 &= S_S \\
&\Rightarrow^*_{G_S} \\
w_2 &= f_S(w_1)\ Color\ Photo\ Skin\ Faces \\
&= http://\ldots\ Color\ Photo\ Skin\ Faces \\
&\Rightarrow^*_{G_{Color}}
\end{aligned}
$$

$$
\begin{aligned}
w_3 &= \mathit{http://\ldots}\ f_{Color}(w_2)\ \mathit{Photo\ Skin\ Faces} \\
&= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19\ Photo\ Skin\ Faces} \\
&\Rightarrow^*_{G_{Photo}} \\
w_4 &= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19}\ f_{Photo}(w_3)\ \mathit{Skin\ Faces} \\
&= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19\ true\ Skin\ Faces} \\
&\Rightarrow^*_{G_{Skin}} \\
w_5 &= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19\ true}\ f_{Skin}(w_4)\ \mathit{Faces} \\
&= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19\ true\ 00\ldots Faces} \\
&\Rightarrow^*_{G_{Faces}} \\
w_6 &= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19\ true\ 00\ldots}\ f_{Faces}(w_5) \\
&= \mathit{http://\ldots}\ \mathit{29053\ 0.03\ 0.19\ true\ 00\ldots 1}
\end{aligned}
$$

□

The S_S start symbol allows for the non-terminal $Image$ to be the "real" start symbol. The initial sentence $\mathit{http://\ldots}$ is produced by f_S and triggers the mappings of the other detectors.

2.2.2 Atoms

Enumerating the complete terminal domain as illustrated in the example grammar is far from practical. But as this grammar is CF, and CF grammars are closed under substitution, a CF language L_a can be chosen to further describe each symbol a in Σ.

Example 2.7.

$$
\begin{aligned}
\mathit{url} &\rightarrow \{\text{\^{} http://([\^{} :/]*)(:[0-9]*)?/?(.*)\$}\} \\
\mathit{int} &\rightarrow \{\text{\^{} -?[0-9]+\$}\} \\
\mathit{flt} &\rightarrow \{\text{\^{} -?[0-9]+\textbackslash.[0-9]+([Ee][-+]?[0-9]+)?\$}\} \\
\mathit{bit} &\rightarrow \{\text{\^{} (0|1|(true)|(false))\$}\} \\
\mathit{bitmap} &\rightarrow \{\text{\^{} (0|1)*\$}\}
\end{aligned}
$$

□

In this case for url the regular expression (remember $\mathcal{L}(\text{REG}) \subset \mathcal{L}(\text{CF})$) corresponds to L_{url}, and so on. The non-terminals which are the root of such a substitution language are called *atoms* (in parsing literature they are sometimes called *pre-terminals* [Tom86]). The yield of a specific (partial) parse tree rooted by an atom is called an *instantiation* of the atom, *i. e. 29053* is an instantiation of the atom domain int or in short: $\mathit{int}(29053)$. The complete set of terminals described by the regular expressions are called the *lexicon* of the grammar.

In practice this phase is called *lexical analysis* [ASU86], in which the stream of characters or bytes is translated into pre-terminals. This abstraction process helps to

get the actual grammar rules closer to a *semantic grammar* [BB75]. In a semantic grammar the rules and symbols are designed to correspond directly to entities and relations from the domain being discussed. This makes the results associated with the grammar, *i. e.* the parse trees, better suited for (human) interaction in other system components of the DMW application.

2.2.3 Dependencies

Do feature grammar systems as defined until now capture the dependency types as identified in Chapter 1? Contextual dependencies, like a $Photo$ which contains $Skin$ colors, may also contain one or more $Faces$, are clearly described in grammars. But how about output/input dependencies?

In the first chapter it was stated that due to the explicit handling of context dependencies detectors stay generic. However, this property is limited again by the output/input dependency. The definition of the output/input dependency of a detector should be powerful enough to be precise, without being to restrictive on the context. This context knowledge influences both the input and output specification of a detector component, which will be investigated in the next two sections.

Detector Input

In Definitions 2.9 and 2.10 detector functions depend on the whole sentential form as input, but in reality the mapping function only uses a part of this information. For example: f_{Faces} only needs the bitmap of skin pixels to find the number of faces, all the other information, like it is a photo, is irrelevant for its mapping. This form of mild context-sensitivity can be captured by adding a regulated rewriting mechanism as a start condition to a detector component.

In Section 2.1.3 C grammars were introduced. Using these conditions on the sentential form a detector component can only become active when its input is available. Take once more the $Faces$ detector: it needs the skin *bitmap* derived from the image. The REG language $R_{Faces} = (.^* \cdot 00 \ldots \cdot .^*)$, using the in Section 2.1.1 defined regular expression syntax, indicates that the sentential form should always contain a specific bitmap. However, as this condition works directly on the sentential form it does not give much control over the semantic context, *e. g.* the non-terminal level, where the desired input data resides in. TC grammars could give more grip on this context as they operate on the parse tree. But both mechanisms of the C and TC grammars restrict the context of the desired input in a horizontal fashion, *i. e.* C grammars limit the left and right contexts in the sentential form, while TC grammars limit the level sentences of the parse tree. To be able to create these conditions the developer of a detector needs to have complete knowledge about the context of the detector in a specific feature grammar system, which is just what the system should circumvent.

An easier and more context-free way is to look at the parse tree vertically, *i. e.* use the paths in the parse tree. This new regulated rewriting mechanism, *i. e.* leading to

path-controlled (PC) grammars, can be defined as follows:

Definition 2.11. *In a* path-controlled grammar $G = (N, T, P, S)$ *the productions* P *are of the form* $(w \rightarrow z, R)$ *where* R *is a* REG *language over the alphabet* V. N, T *and* S *have their normal meaning. The rule* $(w \rightarrow z, R)$ *is applicable to* $x = x_l w x_r$ *yielding* $y = x_l z x_r$, *i. e.* $x \Rightarrow y$, *iff the parse tree* t *associated to* x *satisfies* $R \subseteq path(t)$.

□

Notice that PC grammars are at least as powerful as C grammars, *i. e.* a C grammar can always be translated into a PC grammar.

A special variant of this type of grammars are *left path-controlled* (lPC) grammars. In their case only paths referring to the left context of w, x_l, are allowed.

Definition 2.12. *In a* left path-controlled grammar $G = (N, T, P, S)$ *the productions* P *are of the form* $(w \rightarrow z, R)$ *where* R *is a* REG *language over the alphabet* V. N, T *and* S *have their normal meaning. The rule* $(w \rightarrow z, R)$ *is applicable to* $x = x_l w x_r$ *yielding* $y = x_l z x_r$, *i. e.* $x \Rightarrow y$, *iff the parse tree* t *associated to* x *satisfies* $\delta(R \subseteq path(t)) \subseteq x_l$.

□

The *delta* operation (δ, see Section 2.1.2) creates a new sentence from the selection of paths from t which is created by the intersection between $path(t)$ and R_i. This new sentence may only contain terminals from x_l.

Adding this specific version of the path control mechanism to Definition 2.9 gives the formal definition of a lPC feature grammar system:

Definition 2.13. *A* left path-controlled feature grammar system *is a feature grammar system* Γ *as in Definition 2.9 with* $G_i = (V, P_i, R_i, f_i)$, *where* R_i *is a* REG *language over* V. *The start* (R_i) *and stop* (f_i) *conditions of the component* G_i *restrict a derivation in the following way:*

$$w_j = w_l d_i w_r \Rightarrow^*_{G_i} w_l f_i(\delta(R_i \cap path(t_j)))w_r = w_{j+1}$$

Such that $\delta(R_i \subseteq path(t_j)) \subseteq w_l$. *Where* t_j *is the parse tree associated with the sentential form* w_j, *i. e.* $yield(t_j) = w_j$.

□

The new sentence created by the *delta* operation contains exactly the information the detector f_i needs to perform its mapping, *i. e.* the context knowledge is optimally limited.

Using the atoms as terminal level, and adding this additional rewriting mechanism the example feature grammar looks formally as follows:

Example 2.8.

$$
\begin{aligned}
\Gamma = (&\\
&D = \{S_S, Color, Graphic, Photo, Skin, Faces\},\\
&N = \{Image, Location, Number, Prevalent, Saturation, Class\},\\
&T = \{url, int, flt, bit, bitmap\},\\
&P_N = \{(Image \rightarrow Location\ Color\ Class), \ldots,\\
&\qquad (Class \rightarrow Photo\ Skin\ Faces)\},\\
&G_{Color} = (V, P_{Color} = \{(Color \rightarrow Number\ Prevalent\ Saturation)\}\\
&\qquad \cup P_N, (.^* \cdot Location \cdot url), f_{Color}),\\
&G_{Graphic} = (V, P_{Graphic} = \{(Graphic \rightarrow bit)\} \cup P_N, (.^* \cdot Number \cdot int)+\\
&\qquad (.^* \cdot Prevalent \cdot flt) + (.^* \cdot Saturation \cdot flt), f_{Graphic}),\\
&G_{Photo} = (V, P_{Photo} = \{(Photo \rightarrow bit)\} \cup P_N, (.^* \cdot Color \cdot .^*), f_{Photo}),\\
&G_{Skin} = (V, P_{Skin} = \{(Skin \rightarrow bitmap)\} \cup P_N, (.^* \cdot Location \cdot url), f_{Skin}),\\
&G_{Faces} = (V, P_{Faces} = \{(Faces \rightarrow int)\} \cup P_N, (.^* \cdot bitmap), f_{Faces}),\\
&G_S = (V, P_S = \{(S_S \rightarrow Image)\} \cup P_N, \emptyset, f_S),\\
&S = S_S\\
&)
\end{aligned}
$$

□

The derivation process of the example sentence using this *lPC* feature grammar looks as follows (see the parse tree in Figure 2.4 for the binding of the regular path expressions):

Example 2.9.

$$
\begin{aligned}
w_1 &= S_S\\
&\Rightarrow^*_{G_S}\\
w_2 &= f_S(\lambda)\ Color\ Photo\ Skin\ Faces\\
&= url(\text{http://}\ldots)\ Color\ Photo\ Skin\ Faces\\
&\Rightarrow^*_{G_{Color}}\\
w_3 &= url(\text{http://}\ldots)\ f_{Color}(url(\text{http://}\ldots))\ Photo\ Skin\ Faces\\
&= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\ Photo\ Skin\ Faces\\
&\Rightarrow^*_{G_{Photo}}\\
w_4 &= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\\
&\qquad f_{Photo}(int(29053)\ flt(0.03)\ flt(0.19))\ Skin\ Faces\\
&= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\ bit(\text{true})\ Skin\ Faces\\
&\Rightarrow^*_{G_{Skin}}
\end{aligned}
$$

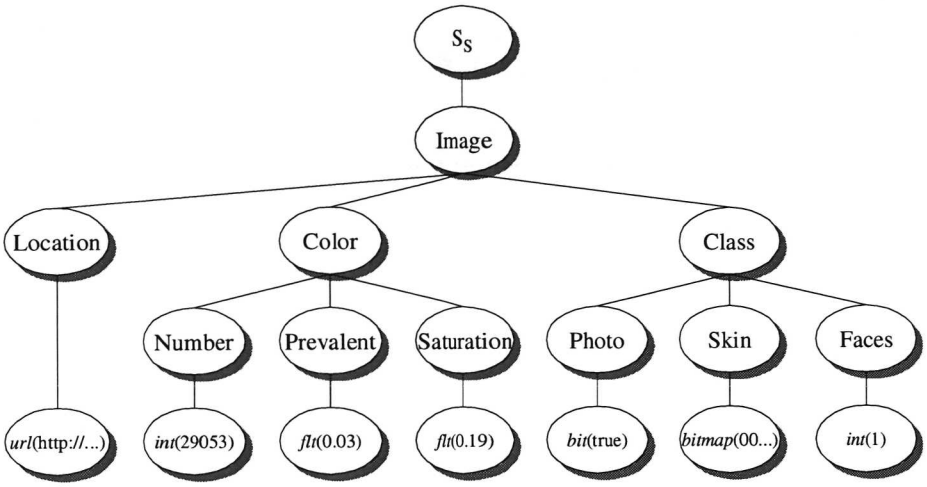


Figure 2.4: A parse tree constructed by a lPC feature grammar system

$$
\begin{aligned}
w_5 &= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\ bit(\text{true}) \\
&\quad f_{Skin}(url(\text{http://}\ldots))\ Faces \\
&= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\ bit(\text{true})\ bitmap(00\ldots)\ Faces \\
&\Rightarrow^*_{G_{Faces}} \\
w_6 &= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\ bit(\text{true})\ bitmap(00\ldots) \\
&\quad f_{Faces}(bitmap(00\ldots)) \\
&= url(\text{http://}\ldots)\ int(29053)\ flt(0.03)\ flt(0.19)\ bit(\text{true})\ bitmap(00\ldots)\ int(1)
\end{aligned}
$$

□

The derivation form in this example is unspecified: control may be transfered to any detector for which the input sentence is valid. For example, after derivation of w_2 control can be transfered to both G_{Color} and G_{Skin}. The example derivation favors the leftmost symbol, *i. e.* resembles a *leftmost* derivation (see Section 2.1.2). The leftmost derivation of detector symbols is always applicable in a lPC grammar, as this rewrite mechanism enforces that the input sentence of the leftmost symbol is always available (when the sentence is valid). However, notice that normal non-terminals take precedence over detectors, *e. g.* the $Class$ non-terminal is resolved before control is transfered to the $Color$ detector. So the leftmost derivation takes place on the control level of the grammar system.

Using the lPC rewriting mechanism for a feature grammar system has a practical advantage: it implements a *deadlock* prevention mechanism. Deadlock situations occur when the regular expressions of two or more detectors depend on each others,

not yet available, parse trees. The start conditions of these detectors can thus never be satisfied. A common deadlock prevention strategy is *linear ordering* [Sta92]. lPC implements this strategy in a straightforward fashion: detectors only depend on the existence of preceding terminals, and can thus naturally be ordered from left to right.

The addition of left path-control to a feature grammar system turns it from a $(CD'_{\infty}CF, (a, C))$ into a $(CD'_{\infty}CF, (lPC, C))$ system.

Detector Output

Just like the detector function in Definition 2.9 depends on a sentential form for its input its result should also exactly match a sentential form, *i. e.* the stop condition of the component. In the example grammar this works fine as there are no nested detectors like:

Example 2.10.

$$Color \rightarrow Number\ isBitmap\ Prevalent\ Saturation$$

with

$$G_{isBitmap} = (V, P_{isBitmap} = \{(isBitmap \rightarrow bit)\} \cup P_N, (.^* \cdot Number \cdot .^*), f_{isBitmap})$$

□

This new $isBitmap$ detector takes the $Number$ of colors and maps it to bit(true) when there are exactly two colors, otherwise it returns bit(false). Integrating this call into the derivation of Example 2.9:

Example 2.11.

$$\begin{aligned} w_{3a} &= url(\text{http://...})\ f_{Color}(url(\text{http://...}))\ Photo\ Skin\ Faces \\ &= url(\text{http://...})\ int(29053)\ isBitmap\ flt(0.03)\ flt(0.19)\ Photo\ Skin\ Faces \\ &\Rightarrow^*_{G_{isBitmap}} \\ w_{3b} &= url(\text{http://...})\ int(29053)\ f_{isBitmap}(int(29053))\ flt(0.03)\ flt(0.19)\ Photo\ Skin \\ &\quad Faces \\ &= url(\text{http://...})\ int(29053)\ bit(\text{false})\ flt(0.03)\ flt(0.19)\ Photo\ Skin\ Faces \end{aligned}$$

□

This example shows that the $Color$ detector now needs to know the exact position of the $isBitmap$ detector in its output sentence. Definition 2.9 captures this by stating that f_i is a mapping from V^* into $(D \cup T)^+$. How to lift the burden of this context knowledge from the detector function? Optimally the range domain becomes a member of T^+. As stated in Section 2.2.1 the output sentence, z, can be seen as the only member of language described by a REG language. By enlarging $z \in T^+$ to

a language containing all possible nestings of detectors, the detector implementation can become less context-sensitive.

$$\begin{aligned} f_i(w) &= a_1 \dots a_m = z_{d_i} \\ A &= (d_1 + \dots + d_n) \\ f_D(z_{d_i}) &= (A^* \cdot a_1 \cdot A^* \dots A^* \cdot a_q \cdot A^* \dots A^* \cdot a_m \cdot A^*) \end{aligned}$$

The function f_D takes the output of f_i and turns it into a REG language consisting of all words interleaved with arbitrary sequences of detectors, represented by the REG language A.

Definition 2.14. *Let a* conditional feature grammar system Γ *be a feature grammar system as in Definition 2.9 where f_i is a mapping from $w \in T^+$ into $z \in T^+$. The stop condition f_i of the component G_i restricts the derivation in the following way:*

$$w_j = w_l d_i w_r \Rightarrow^*_{G_i} w_l z w_r = w_{j+1}$$

where

$$z \in f_D(f_i(w_j))$$

The function $f_D : T^+ \rightarrow \mathcal{L}(REG)$ maps the output sentence, z_{d_i}, of f_i into a REG language where each terminal, a_q a word in z_{d_i}, is enveloped by an arbitrary sequence of detectors.

□

Notice that this adapted stop condition is also used for the output of the special "dummy" detector f_S.

Using this latest addition to the concept of feature grammar systems the (extended) example derivation can be (partially) rewritten:

Example 2.12.

$$\begin{aligned} w_2 &= url(\text{http://}\dots)\ Color\ Photo\ Skin\ Faces \\ &\Rightarrow^*_{G_{Color}} \\ w_{3a} &= url(\text{http://}\dots)\ int(29053)\ isBitmap\ flt(0.03)\ flt(0.19)\ Photo\ Skin\ Faces \\ &\Rightarrow^*_{G_{isBitmap}} \\ w_{3b} &= url(\text{http://}\dots)\ int(29053)\ bit(\text{false})\ flt(0.03)\ flt(0.19)\ Photo\ Skin\ Faces \end{aligned}$$

where

$$w_{3a} \in f_D(f_{Color}(w_2)) = (A^* \cdot int(29053) \cdot A^* \cdot flt(0.03) \cdot A^* \cdot flt(0.19) \cdot A^*)$$
$$w_{3b} \in f_D(f_{isBitmap}(w_{3a})) = (A^* \cdot bit(\text{false}) \cdot A^*)$$

and

$$A = (S_S + Color + Graphic + Photo + Skin + Faces + isBitmap)$$

□

The derivation of the symbol *Color* from w_2 into w_{3a} is a member of the language generated by $f_D(f_{Color}(w_2)$ and thus satisfies the stop condition of the G_{Color} component. So the implementation of the *Color* detector function is now just as context-sensitive as needed, as it contains a precise specification of the input and produces a context-free sentence.

2.2.4 Ambiguous Feature Grammar Systems

Just like any other grammar feature grammar components may be ambiguous. In fact support for ambiguity is identified in Section 1.2.1 as a main requirement of the target application domain of Acoi. Take for example the classification of the *Image* into either a *Photo* or a *Graphic*. It is possible that both classes are valid, *i. e.* the f_{Photo} and $f_{Graphic}$ detectors both return a valid output. This results in two parse trees combined in one forest as discussed in Section 3.

In such a forest the possibility exists that detector input can be bound to nodes from different parse trees, *i. e.* the binding is ambiguous. To illustrate this the example grammar is extended with an object recognizer, *e. g.* to find a vehicle.

Example 2.13.

$$Image \rightarrow Location\ Color\ Class\ Object$$
$$Object \rightarrow Vehicle$$

with

$$G_{Vehicle} = (V, P_{Vehicle} = \{(Vehicle \rightarrow bit)\} \cup P_N,$$
$$(.^* \cdot Location \cdot url + .^* \cdot Class \cdot .^*), f_{Vehicle})$$

□

As shown in this extension the $Vehicle$ detector constructs an input sentence containing the location of the image and the detected class. It uses the class to select between different image segmentation algorithms for photos and graphics. When the class detectors are confident enough, *i. e.* only one is valid, the input sentence of the $Vehicle$

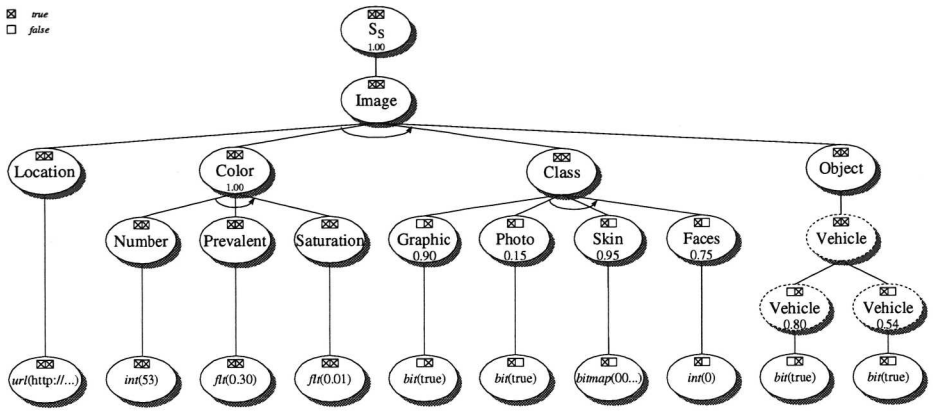
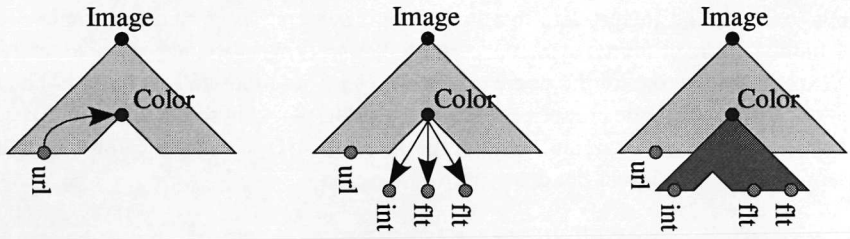


Figure 2.5: A parse forest containing quasi-nodes plus context and confidence annotations

detector can only be bound to one class. However, in the case of ambiguity two bindings are possible. The $Vehicle$ detector cannot prefer one class over the other. Instead it needs to be executed twice: once for each class.

To reflect this in the parse forest the detector node is split in two levels: (1) a top *quasi-node*, the *quasi-root*, and (2) one or more bottom quasi-nodes, the *quasi-foots*. Quasi-nodes belong to quasi-trees which are inspired by D-Theory [MHF83]. [VS92] states: "vp_1 [the quasi-root] and vp_2 [the quasi-foot] can both refer to the same node, to avoid confusion, henceforth we will call them **quasi-nodes**." In the feature grammar system case the quasi-root represents the detector symbol within its rule context. Each quasi-foot represents one possible execution of the detector. By gluing together the quasi-root with one of the quasi-foots one specific instantiation of the detector call is chosen. An example parse forest containing quasi-nodes, indicated with dashed outlines, for the $Vehicle$ detector is shown in Figure 2.5. The figure also shows that when there is only one bottom node the quasi-nodes can be glued together into a normal tree node.

The parse forest in this figure contains additional information to cope with the long distance dependencies of a feature grammar system: nodes are explicitly labeled with a context. As Section 3 already said, CF grammars have only local ambiguity which makes it easy to locate the separate parse trees. However, due to the more advanced dependencies of detector nodes additional information is needed. In the NLP community named disjunctions (NDs) are used [Bla97]. However, NDs expect each disjunction to be of the same arity. Which means that when the third disjunction of a *controller* ND is valid the third disjunction in the *controlled* disjunction should also be valid. In the simple example of Figure 2.5 this is true: the $Class$ disjunction controls the $Vehicle$ disjunction, and within their parse trees the indices of the disjunctions



a) Binding the input sentence

b) Executing the detector

c) Validating the output sentence

Figure 2.6: Execution of the $Color$ detector

match. However, the input sentence of a detector may be constructed from terminal nodes obtained from several disjunctions, *i. e.* there will not be one controlling disjunction. So instead of naming disjunctions each node is provided with a context. This context consists of a list of binary flags, where each flag represents a parse tree and indicates if the node belongs to it (*true*) or not (*false*). In the example forest there are two trees. Most nodes belong to both trees, except for the different $Class$ alternatives and their dependent $Vehicle$ quasi-foots. Limiting the binding of detector parameters to the scope of the current quasi-root is now easily implemented by a binary operation (see Section 4.3.3).

Next to contexts the parse forest in Figure 2.5 also contains detector confidences. Although both $Class$ detectors are valid, they are not both as confident. The $Graphic$ detector is 90% sure that the image is a graphic, while the $Photo$ detector has only a confidence of 15%. Providing this information may help the user or librarian in disambiguating the parse forest. The confidence values are in this case seen as node annotations, but may also be just modeled as ordinary leaf nodes.

Notice that the parse forest shown in Figure 2.5 describes several sentences, which may partially overlap. This is another reason why a packed shared parse forest as introduced in Section 3 is not usable in the case of ambiguous feature grammar systems. A packed shared forest describes always one sentence.

2.2.5 Mildly Context-sensitive Feature Grammar Systems

Combining the two forms of regulated rewriting as defined in Definitions 2.13 and 2.14 with Definition 2.9 the definition of *mildly context-sensitive feature grammar systems* is obtained. This form of feature grammar systems is used throughout the rest of this thesis.

Figure 2.6 illustrates the working of the $Color$ detector. The url instantiation belongs to its input sentence, as specified by $R_{Color} = (.^* \cdot Location \cdot url)$. The f_{Color} detector uses this information to load the image, analyze it, and produce the

output sentence: an integer and two floats. This sentence is parsed and matches the rules in the feature grammar system, so the stop condition is satisfied. The detector has only limited knowledge of the context it operates in: its input and its output. The CF grammar rules within the components can be used to place these detectors in different contexts. However, the start and stop conditions will enforce that these contexts match the (limited) knowledge of the detector.

2.3 Discussion

In this chapter the formal theory of feature grammar systems has been described. To achieve mild context-sensitivity this theory is based on a combination of CD grammar systems and a regulated rewriting mechanism. The major extensions to these existing theories include the addition of detector functions to produce (part of) the sentence just-in-time and the PC and lPC regulation mechanisms. A detector function directly determines, when the start condition is satisfied, the stop condition of the grammar component. This enables a very tight integration between user-defined functions and the grammar component, while staying formally sound. The lPC mechanism allows mildly context-sensitive specification of the start condition, while also implementing a deadlock prevention strategy.

Two kinds of dependencies were identified in Chapter 1 as key components of a declarative dependency description language. A feature grammar system captures them both. The RHSs of the CF production rules within each grammar component describe the context dependencies. While the regular expressions of the start conditions of the same components capture output/input dependencies.

Using a feature grammar system a parse tree/forest can be build which represents all (ambiguous) contextual knowledge about the extracted annotations, and may thus form a good basis for incremental maintenance.

Next to Acoi there are some systems which are also focus at controlling the flow of annotation extraction, or could be used for that purpose. MOODS [GYA97] is based on the extension of an object oriented schema with semantic objects. The novel aspect of a semantic object is its processing graph. In a processing graph extraction algorithms are linked together. The developer defines a split point in the processing graph. The first part is executed in a data-driven way when a new object instance is inserted into the database. The last part is executed in a demand-driven fashion during query processing. Additionally, the object can be extended with inference rules, which can be checked during query processing. There is no specific support in MOODS for context dependencies, unless they are hard-coded as an output/input dependency.

In the Mirror system [dVEK98, dV99] extraction algorithms are encapsulated in daemon (CORBA) objects. The daemons issue a *get_work* query to the database, which functions as a data pool, extract the meta-data (*i. e.* annotations) for the returned objects and then issue a *finish_work* query to commit their results. All knowledge about dependencies is embedded in the *get_work* queries, *i. e.* there is no global

declarative specification. Context dependencies thus need to be tightly bound, *i. e.* hardcoded, and may become distributed over several daemon objects.

Linda tuple spaces [Gel95] can form an alternative for the daemon architecture. The daemon requests its input data once from the tuple space, instead of polling for it from time to time, blocks and becomes active when the tuple space notifies the availability of the data. However, although this offers a different implementation model it does not resolve the need for hardcoding the context dependencies.

Dataflow languages are also focused on output/input dependencies. For example, Microsoft DirectShow [Mic99] offers a way to tie algorithms together using filter graphs. This builds a pipeline of algorithms through which multimedia streams flow. The target of such systems is to produce mainly one final result, *e. g.* a video converted from color to black and white and from the MPEG format to AVI format. In a feature grammar system the result is a parse forest which contains all the annotation information, which may be considered intermediate data in the pipeline.

The ToolBus system [BK94, BK96, dJK] provides a coordination architecture for heterogeneous, distributed software systems. The coordination mechanism, *i. e.* the toolbus, is steered by T scripts, which are based on process algebra [BK86]. A T script contains one or more processes, while each process may control one or more external tools. Tools communicate, *i. e.* send messages or share data, with each other using the bus. To overcome different data formats the tools are encapsulated by adapters which transform the native formatted data into the general form used by the bus, *i. e.* ATerms [vdBdJKO00]. The ToolBus thus addresses separation of coordination (T scripts), representation (ATerms) and computation (the tools). This closely resembles feature grammar systems: where grammar components provide coordination, the parse tree provides representation and detectors computation. The main difference being the, in the case of feature grammar systems, by default constructed parse tree. This tree contains (semantic) contextual knowledge both for data and processes. This knowledge is, as shown in the introductory chapter, indispensable for the application domain considered in this thesis. However, ATerms are generic data types and can be used to describe trees or forests [Vis97]. But in the implementation the dependencies would become implicit in a, more generic, T script and possibly even hidden inside ATerm messages. To enable reasoning about these dependencies, *e. g.* to allow the FDS to steer incremental maintenance, this knowledge would have to made explicit again. So once more a declarative dependency description, *i. e.* in the form of a feature grammar system, would be needed. However, this makes the ToolBus a possible, feature grammar driven, candidate to implement parts of the Acoi system, *e. g.* the FDE.