10

Syntactic Formalisms

10.1 Introduction

Studies on syntax have been the core of linguistics for most of the 20th century. While the goals of traditional grammars had been mostly to prescribe what the correct usage of a language is, the then-emerging syntactic theories aimed at an impartial description of language structures. These ideas revolutionized the field. Research activity was particularly intense in the years 1940–1970, and the focus on syntax was so great that, for a time, it nearly eclipsed phonetics, morphology, semantics, and other disciplines of linguistics.

Among all modern syntax researchers, Noam Chomsky has had a considerable and indisputable influence. Chomsky's seminal work, *Syntactic Structures* (1957), is still considered by many as a key reading in linguistics. In his book (in Sect. 6.1), Chomsky defined grammars as *essentially a theory of* [a language] that should be (1) adequate: whose correctness should be measurable using corpora; (2) general: extendible to a variety of languages, and, as far as possible, (3) simple. As goals, he assigned grammatical rules to describe syntactic structures:

> "These rules express structural relations among the sentences of the corpus and the indefinite number of sentences generated by the grammar beyond the corpus (predictions)."

More specifically (in Sect. 5.5), Chomsky outlined a formal model of syntax under the form of grammars that was precise enough to be programmable and verifiable.

Chomsky's ideas appealed to the linguistics community because they featured an underlying analogy between human languages and computer – or formal – languages together with a mathematical formalism that was already used for compilers. Chomsky came at a convergence point where advances in computer technology, mathematical logic, and programming languages made his theory possible and acceptable. Chomsky's theories on syntactic structures have originated much research in the domain and an astounding number of followers, notably in the United States.

In addition, his theories spurred a debate that went well beyond linguistic circles reaching psychology and philosophy.

In the meantime, linguists in Europe developed other structural approaches and also tried to derive generic linguistic structures. But instead of using the computer operation as a model or to posit cognition universals, as Chomsky did, some of them tried to study and expose examples from a variety of languages to prove their theories. The most prominent figure of the European school is Lucien Tesnière. Although Tesnière's work (1959, 2nd edn., 1966, both posthumous) is less known it is gaining recognition and it is used with success in implementations of grammars and parsers for English, French, German, and many other languages.

Many computational models of syntactic structures are presently based on the notion of constituent. They are inherited from the American school and are a part of Chomskyan grammars – although Chomsky does not limit grammars to a constituent decomposition. The European school has its origin in an older tradition. It is based on the notion of connections between words where each word of a sentence is linked to another one under a relation of subordination or dependence. For this reason, these syntactic models are also called dependency grammars. This chapter introduces both structural approaches – **constituency** and **dependency** – and associated formalisms.

10.2 Chomsky's Grammar in Syntactic Structures

Chomsky fractionates a grammar into three components. The first level consists of phrase-structure (PS) rules expressing constituency. The second one is made of transformation rules that complement PS rules. **Transformations** enable us to derive automatically new constructions from a given structure: a declarative form into an interrogative or a negative one; an active sentence into a passive one. Transformation rules apply to constituent structures or trees and describe systematic mappings onto new structures.

Initially, PS and transformation rules used a vocabulary made of morphemes, roots, and affixes, as well as complete words. The inflection of a verb with the past participle tense was denoted [*en* + *verb*] where *en* represented the past participle affix, for example, [*en* + *arrive*]. A third **morphophonemic** component handled the final word generation, mapping forms such as [*en* + *arrive*] onto *arrived*.

10.2.1 Constituency: A Formal Definition

Constituency is usually associated with context-free grammars. Formally, such grammars are defined by:

1. A set of designated start symbols, Σ, covering the sentences to parse. This set can be reduced to a single symbol, such as `sentence`, or divided into more symbols: `declarative_sentence`, `interrogative_sentence`.
2. A set of nonterminal symbols enabling the representation of the syntactic categories. This set includes the sentence and phrase categories.

3. A set of terminal symbols representing the vocabulary: words of the lexicon, possibly morphemes.
4. A set of rules, F, where the left-hand-side symbol of the rule is rewritten in the sequence of symbols of the right-hand side.

Chomsky (1957) portrayed PS rules with an example generating *the man hit the ball*. It has a straightforward equivalent in DCG:

```
sentence --> np, vp.
np --> t, n.
vp -- verb, np.
t --> [the].
n --> [man] ; [ball] ; etc.
verb --> [hit] ; [took] ; etc.
```

A set of such PS rules can generate sentences. Chomsky illustrated it using a mechanism that resembles the top-down algorithm of Prolog (Fig. 10.1).

Sentence	0
NP + VP	1
T + N + VP	2
T + N + Verb + NP	3
the + N + Verb + NP	4
the + man + Verb + NP	5
the + man + hit + NP	6
the + man + hit + T + N	7
the + man + hit + the + N	8
the + man + hit + the + ball	9

Fig. 10.1. Generation of sentences.

Generation was the main goal of Chomsky's grammars: to produce all potential sentences – word and morpheme sequences – considered to be syntactically correct or acceptable by native speakers. Chomsky introduced recursion in grammars to give a finite set of rules an infinite capacity of generation.

From the initial goal of generation, computational linguists wrote and used grammars to carry out recognition – or parsing – of syntactically correct sentences. A sentence has then to be matched against the rules to check whether it falls within the generative scope of the grammar. Parsing results in a parse tree – the sequence of grammar rules that were applied. The parsing process can be carried out using:

- a top-down mechanism, which starts from the initial symbol – the sentence – down to the words of the sentence to be parsed
- a bottom-up mechanism, which starts from the words of the sentence to be parsed up to the sentence symbol

Some parsing algorithms run more efficiently with a restricted version of context-free grammars called the **Chomsky normal form** (CNF). Rules in the CNF have either two nonterminal symbols to their right-hand side or one nonempty terminal symbol:

```
lhs --> rhs1, rhs2.
lhs --> [a].
```

Any grammar can be converted into an equivalent CNF grammar using auxiliary symbols and rules as for

```
lhs --> rhs1, rhs2, rhs3.
```

which is equivalent to

```
lhs --> rhs1, lhs_aux.
lhs_aux --> rhs2, rhs3.
```

The equivalence is said to be weak because the resulting grammar generates the same sentences but does not yield exactly the same parse trees.

10.2.2 Transformations

The transformational level consists of the mechanical rearrangement of sentences according to some syntactic relations: active/passive, declarative/interrogative, etc. A transformation operates on a sentence with a given phrase structure and converts it into a new sentence with a new derived phrase structure. Transformations use rules – transformational rules or T-rules – to describe the conversion mechanism as:

```
T1: np1, aux, v, np2 →
    np2, aux, [be], [en], v, [by], np1
```

which associates an active sentence to its passive counterpart. The active part of the rule matches sentences such as

the man will hit the ball

and its passive part enables us to generate the equivalent passive sentence:

the ball will be (en hit) by the boy

where (*en hit*) corresponds to the past participle of verb *to hit*. An additional transformational rule permutes these two elements:

```
T2: affix, v →
    v, affix, #
```

where # marks a word boundary. Once applied, it yields

the ball will be hit en # by the boy

Hit en # is then rewritten into *hit* by morphophonemic rules. Finally, the transformational process yields:

the ball will be hit by the boy

A tree-to-tree mapping as shown in Fig. 10.2 can also reflect transformation rules.

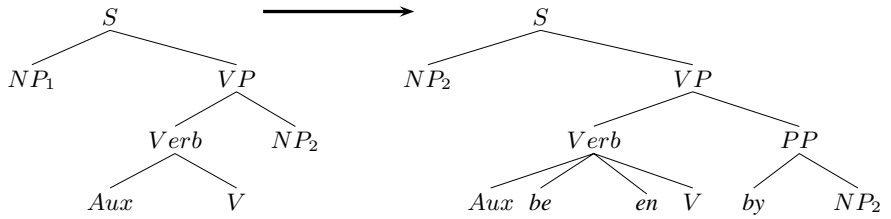

Fig. 10.2. A tree-to-tree mapping representing the active/passive transformational rule.

Other common transformations include (Chomsky 1957):

- Negations. *John comes → John doesn't come*.
- Yes/no questions. *they arrive → do they arrive*; *they have arrived → have they arrived*; *they can arrive → can they arrive*; *they arrived → did they arrive*
- Interrogatives. *John ate an apple → did John eat an apple*; *John ate an apple → what did John eat*; *John ate an apple → who ate an apple*
- Conjunction. (*the scene of the movie was in Chicago; the scene of the play was in Chicago) → the scene of the movie and of the play was in Chicago*.
- Topicalization. that is, moving a constituent in front of a sentence to emphasize it. *the waiter brought the meal to the table → to the table, the waiter brought the meal*; *I don't like this meal → this meal, I don't like.*

In Chomsky's formalism, PS rules are written so that certain generated sentences require a transformation to be correct. Such transformations are said to be obligatory. An example is given by the affix permutation rule (T_2). Other rules are optional, such as the passive/active transformation (T_1). PS rules and obligatory transformations account for the "kernel of a language" and generate "kernel sentences". All other sentences can be unfolded and mapped onto this kernel using one or more transformations.

According to Chomsky, transformations simplify the description of a grammar, and make it more compact. Writing a grammar only requires the phrase structure of kernel sentences, and all others are derived from transformations. Later Chomsky related kernel sentences to a deep structure, while transformed sentences correspond to a surface structure. Transformations would then map the surface structure of a sentence onto its deep structure. The deep structure would consist of a set of obligatory transformations and a core phrase structure on which no transformation could be carried out.

10.2.3 Transformations and Movements

Transformation theory evolved into the concept of movement (Chomsky 1981). A movement is a sentence rearrangement where a constituent is moved to another location. The moved constituent leaves a **trace**: an empty symbol representing its initial location. Passives correspond to a composition of two movements: one that moves the subject noun phrase into the position of a prepositional phrase headed by *by*; and another that moves the object noun phrase into the empty subject position (Table 10.1).

Table 10.1. Movements to obtain the passive of sentence *The man hit the ball*. Traces are represented by —. Original positions of traces are in bold.

Movements	Traces	Passives
First movement	***The man*** *hit . . .*	*. . . — is hit by the man*
Second movement	*. . . hit* ***the ball***	*The ball is hit —*

Paradigms of movement are questions beginning with an interrogative pronoun or determiner: the *wh*-movements. A *wh*-word – *who, which, what, where* – is moved to the beginning of the sentence to form a question. Consider the sentence *John ate an apple in the dining room.* According to questions and to the *wh*-word type in front of the question, a trace is left at a specific location in the original sentence (Table 10.2). Traces correspond to noun phrases.

Table 10.2. Questions beginning with a *wh*-word and their traces (—).

Questions	Traces
Who ate an apple in the dining room?	*— ate an apple in the dining room*
What did John eat in the dining room?	*John ate — in the dining room*
Which apple did John eat in the dining room?	*John ate — in the dining room*
Where did John eat an apple?	*John ate an apple —*

Transformations or movements use a syntactic model of both the original phrase – or sentence – and its transformed counterpart. These models form the left and right members of a T-rule. Applying a transformation to a phrase or conversely unfolding a transformation from it, requires knowing its tree structure. In consequence, transformational rules or movements need a prior PS analysis before being applied.

10.2.4 Gap Threading

Gap threading is a technique to parse *wh*-movements (Pereira 1981, Pereira and Shieber 1987). Gap threading uses PS rules that consider the sentence after the movement has occurred. This requires new rules to account for interrogative pronouns or interrogative determiners moved in front of sentence, as for:

John ate an apple
What did John eat?

with a rule to parse the declaration

```
s --> np, vp.
```

and a new one for the question

```
s --> [what, did], np, vp.
```

One aim of gap threading is to keep changes in rules minimal. Thus the trace – or **gap** – should be handled by rules similar to those of a sentence before the movement. The rule describing a verb phrase with a transitive verb should remain unchanged:

```
vp --> v, np.
```

with the noun phrase symbol being possibly empty in case of a gap.

```
np --> [].
```

However, such a rule is not completely adequate because it would not differentiate a gap: the absence of a noun phrase resulting from a movement, from the pure absence of a constituent. Rules could insert empty lists wrongly in sentences such as

John ate

To handle traces properly, gap threading keeps a list of the moved constituents – or **fillers** – as the parsing mechanism reads them. In our example, fillers are *wh*-terms. When a constituent contains a moved term, it is stored in the filler list. When a constituent contains a gap – a missing noun phrase – a term is reclaimed from the head of the filler list.

Gap threading uses two lists as arguments that are added to each constituent of the DCG rules. These lists act as input and output of gaps in the current constituent, as in:

```
s(In, Out) --> np(In, Out1), vp(Out1, Out).
```

At a given point of the analysis, the first list holds fillers that have been stored before, and the second one returns the remaining fillers once gaps have been filled in the constituent.

In the sentence

What did John eat —?

the verb phrase *eat* — contains a gap. Before processing this phrase, the filler list must have accumulated *what*, which is removed when the verb phrase is completely parsed. Hence, input and output arguments of the `vp` constituent must be:

```
% vp(In, Out)
vp([what], [])
```

or, to be more general,

```
vp([what | T], T)
```

The noun phrase rule handling the gap accepts no word as an input (because it is a gap). Its right-hand side is then an empty list. The real input is received from the filler list. The rule collects the filler from the first argument of `np` and returns the resulting list in the second one:

```
np([what | T], T) --> [].
```

The whole set of rules is finally:

```
s(In, Out) -->
  [what, did],
  np([ what | In], Out1),
  vp(Out1, Out).
s(In, Out) --> np(In, Out1), vp(Out1, Out).

np(X, X) --> ['John'].      % no gap here
np(X, X) --> det, n.        % no gap here
np([what | T], T) --> [].   % the gap

vp(In, Out) --> v, np(In, Out).

v --> [eat]; [ate].

det --> [an].

n --> [apple].
```

When parsing a sentence with a movement, initial and final filler lists are set to empty lists:

```
?- s([], [], [what, did, 'John', eat], []).
Yes
```

as in the initial declaration:

```
?- s([], [], ['John', ate, an, apple], []).
Yes
```

10.2.5 Gap Threading to Parse Relative Clauses

Gap threading can also be used to parse relative clauses. Relative clauses are sentences complementing a noun phrase whose subject or object has been replaced by a relative pronoun. Consider the noun phrase

The meal that the waiter brought

The rule describing such a phrase is

```
np --> ngroup, relative
```

where `ngroup` maps *the meal* and `relative` maps *that the waiter brought*.

The modified sentence corresponding to the relative clause here is

The waiter brought the meal

where the noun phrase *the meal* has been moved from its object position to the front of the relative and has been replaced by the object pronoun *that* (Fig. 10.3). The phrase

The waiter who brought the meal

is similar, but the movement occurs on the subject noun phrase, which is replaced by subject pronoun *who*.

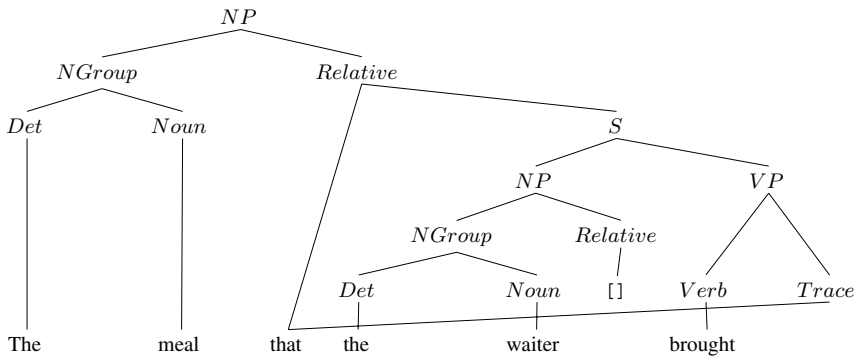

Fig. 10.3. The parse tree of *The meal that the waiter brought* with gap threading.

Let us write a grammar using gap threading to parse such noun phrases. The top rule has two new variables to hold fillers:

```
np(In, Out) -->
  ngroup(In, Out1),
  relative(Out1, Out).
```

The relative clause is a sentence that starts with a pronoun, and this pronoun is stored in the filler input list of the sentence symbol:

```
relative(In, Out) --> [that], s([that | In], Out).
relative(In, Out) --> [who], s([who | In], Out).
```

There might also be no relative clause

```
relative(X, X) --> [].
```

When we encounter a trace, a noun phrase is missing. The head pronoun is then removed from the filler list

```
np([PRO | T], T) --> [].
```

The rest of the grammar is straightforward

```
s(In, Out) --> np(In, Out1), vp(Out1, Out).

vp(In, Out) --> v, np(In, Out).

ngroup(X, X) --> det, n.

det --> [the].
n --> [waiter].
n --> [meal].
v --> [brought].
```

When launching the parse, both filler lists are empty

```
?- np([], [],[the, meal, that, the, waiter, brought],
  []).
Yes

?- np([], [], [the, waiter, who, brought, the, meal],
  []).
Yes
```

In the examples above, we have made no distinction between object and subject pronouns. The program could have been refined to take this difference into account.

10.3 Standardized Phrase Categories for English

The aim of a standard for phrase categories is to define an annotation set that would be common to people working on syntax. Such a standard would facilitate corpus and program sharing, assessment, and communication between computational linguists. Currently, there is no universally accepted standard. Defining an annotation set requires finding a common ground on the structure or the denomination of a specific group of words. It proves to be more difficult than expected. There is a consensus on main categories but details are sometimes controversial.

Most annotation schemes include phrase categories mapping the four main parts of speech, namely nouns, verbs, adverbs, and adjectives. Category names correspond to those of constituent heads:

- **Noun phrases** (NP), phrases headed by a noun.

- **Verb phrases** (VP), phrases headed by a verb together with its objects.
- **Adjective phrase** (AdjP), a phrase headed by an adjective, possibly with modifiers.
- **Adverbial phrase** (AdvP), a phrase headed by an adverb.
- Most annotation sets also feature **prepositional phrases** (PP): noun phrases beginning with a preposition.

The Penn Treebank (Marcus et al. 1993) is a corpus annotated with part-of-speech labels. Parts of it are also fully bracketed with syntactic phrase categories, and it was one of the first corpora widely available with such an annotation. Table 10.3 shows its set of phrase labels.

Table 10.3. The Penn Treebank phrase labels. After Marcus et al. (1993).

	Categories	Description
1.	ADJP	Adjective phrase
2.	ADVP	Adverb phrase
3.	NP	Noun phrase
4.	PP	Prepositional phrase
5.	S	Simple declarative clause
6.	SBAR	Clause introduced by subordinating conjunction or 0 (see below)
7.	SBARQ	Direct question introduced by *wh*-word of *wh*-phrase
8.	SINV	Declarative sentence with subject-aux inversion
9.	SQ	Subconstituent of SBARQ excluding *wh*-word of *wh*-phrase
10.	VP	Verb phrase
11.	WHADVP	*wh*-adverb phrase
12.	WHNP	*wh*-noun phras
13.	WHPP	*wh*-prepositional phrase
14.	X	Constituent of unknown or uncertain category
	Null elements	
1.	*	"Understood" subject of infinitive or imperative
2.	0	Zero variant of *that* in subordinate clauses
3.	T	Trace – marks position where moved *wh*-constituent is interpreted
4.	NIL	Marks position where preposition is interpreted in pied-piping context

As an example, Fig. 10.4 shows the bracketing of the sentence

> *Battle-tested industrial managers here always buck up nervous newcomers with the tale of the first of their countrymen to visit Mexico, a boatload of samurai warriors blown ashore 375 years ago.*

in the Penn Treebank, where `pseudo-attach` denotes an attachment ambiguity for `VP-1`. In effect, *blown ashore* can modify either *boatload* or *samurai warriors*. Both attachments mean roughly the same thing, and there is no way to remove the ambiguity. In this bracketing, *blown ashore* has been attached arbitrarily to *warriors*, and a pseudo-attach has been left to indicate a possible attachment to *boatload*.

```
( (S
   (NP Battle-tested industrial managers
       here)
    always
   (VP buck
       up
       (NP nervous newcomers)
       (PP with
           (NP the tale
           (PP of
               (NP (NP the
                       (ADJP first
                             (PP of
                                 (NP their countrymen)))
                       (S (NP *)
                          to
                          (VP visit
                              (NP Mexico))))
                   ,
                   (NP (NP a boatload
                           (PP of
                               (NP (NP samurai warriors)
                                   (VP-1 blown
                                       ashore
                                       (ADVP (NP 375 years)
                                             ago)))))
                       (VP-1 *pseudo-attach*)))))))))
.)
```

Fig. 10.4. Bracketed text in the Penn Treebank. After Marcus et al. (1993, p. 325).

Bracketing of phrases is done semiautomatically. A first pass uses an automatic parser. The output is then complemented or corrected by hand by human annotators.

10.4 Unification-Based Grammars

10.4.1 Features

In the examples above, there is no distinction between types of noun phrases. They appeared under a unique category: `np`. However, noun phrases are often marked with additional grammatical information, that is, depending on the language, a person, a number, a gender, a case, etc. In German, cases correspond to a specific inflection visible on the surface form of the words (Table 10.4). In English and French, noun phrases are inflected with plural, and in French with gender. We saw in Chap. 5 that such grammatical characteristics are called **features**. Case, gender, or number are

Table 10.4. Inflection imposed to noun group *der kleine Ober* 'the small waiter' by the case feature in German.

Cases	Noun groups
Nominative	*der kleine Ober*
Genitive	*des kleinen Obers*
Dative	*dem kleinen Ober*
Accusative	*den kleinen Ober*

features of the noun that are also shared by the components of the noun phrase to which it belongs.

If we adopt the generative framework, it is necessary to take features into account to have correct phrases. We can get a picture of it with the German cases and a very simple noun phrase rule:

```
np --> det, adj, n.
```

Since we do not distinguish between `np` symbols, the rule will output ungrammatical phrases as:

```
?-np(L, []).
[der, kleinen, Ober];   %wrong
[der, kleinen, Obers];  %wrong
[dem, kleine, Obers]    %wrong
...
```

To avoid such a wrong generation, we need to consider cases and other features and hence to refine our model. In addition, beyond generation features are necessary in many applications such as spelling or grammar checking, style critique, and so on.

A solution could be to define new noun phrase symbols corresponding to cases such as `np_nominative`, `np_genitive`, `np_dative`, `np_accusative`. We need others to consider number, `np_nominative_singular`, `np_nominative_plural`, ..., and it is not over, because of gender: `np_nominative_singular_masc`, `np_nominative_singular_fem`, ... This process leads to a division of main categories, such as noun phrases, nouns, and adjectives, into subcategories to account for grammatical features.

10.4.2 Representing Features in Prolog

Creating a new category for each grammatical feature is clumsy and is sometimes useless in applications. Instead of it, features are better represented as arguments of main grammatical categories. This is straightforward in Prolog using the DCG notation. To account for cases in noun phrases, let us rewrite `np` into:

```
np(case:C)
```

where the C value is a member of list [nom, gen, dat, acc] denoting nominative, genitive, dative, and accusative cases.

We can extend the number of arguments to cover the rest of grammatical information. Prolog functors then represent main categories such as noun phrases, and arguments represent the grammatical details. Arguments are mapped onto feature structures consisting of pairs feature/values as for gender, number, case, person, and type of determiner:

```
np(gend:G, num:N, case:C, pers:P, det:D)
```

Using Prolog's unification, features are easily shared among constituents making up a noun phrase as in the rule:

```
np(gend:G, num:N, case:C, pers:P, det:D) -->
  det(gend:G, num:N, case:C, pers:P, det:D),
  adj(gend:G, num:N, case:C, pers:P, det:D),
  n(gend:G, num:N, case:C, pers:P).
```

Let us exemplify it with a small fragment of the German lexicon:

```
det(gend:masc, num:sg, case:nom, pers:3, det:def) -->
  [der].
det(gend:masc, num:sg, case:gen, pers:3, det:def) -->
  [des].
det(gend:masc, num:sg, case:dat, pers:3, det:def) -->
  [dem].
det(gend:masc, num:sg, case:acc, pers:3, det:def) -->
  [den].

adj(gend:masc, num:sg, case:nom, pers:3, det:def) -->
  [kleine].
adj(gend:masc, num:sg, case:gen, pers:3, det:def) -->
  [kleinen].
adj(gend:masc, num:sg, case:dat, pers:3, det:def) -->
  [kleinen].
adj(gend:masc, num:sg, case:acc, pers:3, det:def) -->
  [kleinen].

n(gend:masc, num:sg, case:nom, pers:3) --> ['Ober'].
n(gend:masc, num:sg, case:gen, pers:3) --> ['Obers'].
n(gend:masc, num:sg, case:dat, pers:3) --> ['Ober'].
n(gend:masc, num:sg, case:acc, pers:3) --> ['Ober'].
```

To consult this lexicon, Prolog needs a new infix operator ":" that we define using the op/3 built-in predicate:

```
:- op(600, xfy, ':').
```

And our grammar generates correct noun phrases only:

```
?- np(_, _, _, _, _, L, []).
    L = [der, kleine, 'Ober'] ;
    L = [des, kleinen, 'Obers'] ;
    L = [dem, kleinen, 'Ober'] ;
    L = [den, kleinen, 'Ober'] ;
 No
```

10.4.3 A Formalism for Features and Rules

In the previous section, we directly wrote features as arguments of Prolog predicates. More frequently, linguists use a notation independent of programming languages, which is referred to as unification-based grammars. This notation is close to Prolog and DCGs, however, and is therefore easy to understand. The noun phrase rule

```
np(gend:G, num:N, case:C, pers:P, det:D) -->
  det(gend:G, num:N, case:C, pers:P, det:D),
  adj(gend:G, num:N, case:C, pers:P, det:D),
  n(gend:G, num:N, case:C, pers:P).
```

is represented as:

$$\underset{\begin{bmatrix} gend:G \\ num:N \\ case:C \\ pers:P \\ det:D \end{bmatrix}}{NP} \rightarrow \underset{\begin{bmatrix} gend:G \\ num:N \\ case:C \\ pers:P \\ det:D \end{bmatrix}}{DET} \underset{\begin{bmatrix} gend:G \\ num:N \\ case:C \\ pers:P \\ det:D \end{bmatrix}}{ADJ} \underset{\begin{bmatrix} gend:G \\ num:N \\ case:C \\ pers:P \end{bmatrix}}{N}$$

Rules of a grammar describing complete sentences are similar to those of DCGs. They consist, for example, of:

$$S \rightarrow \underset{\begin{bmatrix} num:N \\ case:nom \\ pers:P \end{bmatrix}}{NP} \underset{\begin{bmatrix} num:N \\ pers:P \end{bmatrix}}{VP}$$

$$\underset{\begin{bmatrix} num:N \\ pers:P \end{bmatrix}}{VP} \rightarrow \underset{\begin{bmatrix} trans:i \\ num:N \\ pers:P \end{bmatrix}}{V}$$

$$\underset{\begin{bmatrix} num:N \\ pers:P \end{bmatrix}}{VP} \rightarrow \underset{\begin{bmatrix} trans:t \\ num:N \\ pers:P \end{bmatrix}}{V} \underset{[case:acc]}{NP}$$

$$\begin{matrix} NP \\ \begin{bmatrix} gen : G \\ num : N \\ pers : P \\ case : C \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} Pronoun \\ \begin{bmatrix} gen : G \\ num : N \\ pers : P \\ case : C \end{bmatrix} \end{matrix}$$

with lexicon entries such as:

$$\begin{matrix} DET \\ \begin{bmatrix} gend : masc \\ num : sg \\ case : nom \\ det : def \end{bmatrix} \end{matrix} \rightarrow der$$

10.4.4 Features Organization

A feature structure is a set of pairs consisting of a feature name – or attribute – and its value.

$$\begin{bmatrix} feature_1 : value_1 \\ feature_2 : value_2 \\ \vdots \\ feature_n : value_n \end{bmatrix}$$

Unlike arguments in Prolog or DCGs, the feature notation is based solely on the name and not on the position of the argument. Hence, both

$$\begin{bmatrix} gen : fem \\ num : pl \\ case : acc \end{bmatrix} \text{ and } \begin{bmatrix} num : pl \\ case : acc \\ gen : fem \end{bmatrix}$$

denote the same feature structure. Feature structures can be pictured by a graph as shown in Fig. 10.5.

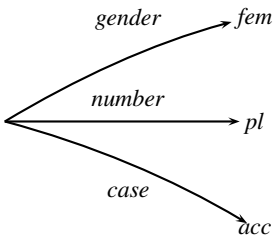

Fig. 10.5. Graph representing a feature structure.

The value of a feature can be an atomic symbol, a variable, or another feature structure to yield a hierarchical organization as in:

$$\begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : \begin{bmatrix} f_5 : v_5 \\ f_6 : v_6 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

whose corresponding graph is shown in Fig. 10.6.

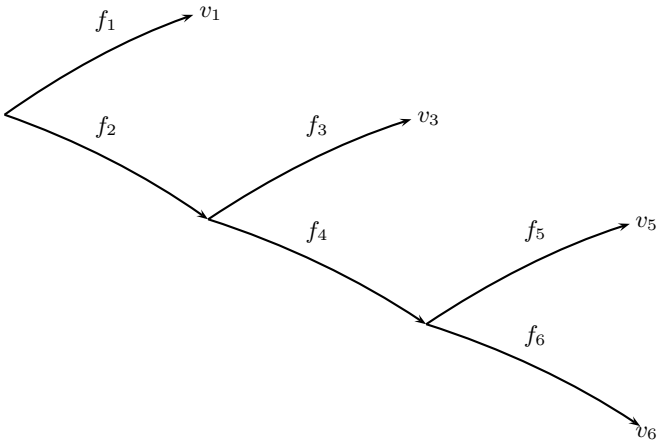

Fig. 10.6. Graph corresponding to embedded feature structures.

Grouping a set of features into a substructure enables the simplification of notations or rules. A feature denoted `agreement` can group gender, number, and person, and can be encoded as a single structure. German nominative and accusative pronouns *er* 'he' and *ihn* 'him' can then be represented as:

$$\begin{matrix} Pronoun \\ \begin{bmatrix} agreement : \begin{bmatrix} gender : masc \\ number : sg \\ pers : 3 \end{bmatrix} \\ case : nom \end{bmatrix} \end{matrix} \rightarrow er$$

$$\begin{matrix} Pronoun \\ \begin{bmatrix} agreement : \begin{bmatrix} gender : masc \\ number : sg \\ pers : 3 \end{bmatrix} \\ case : acc \end{bmatrix} \end{matrix} \rightarrow ihn$$

which enables us to simplify the noun phrase rule in:

$$\begin{matrix} NP \\ \begin{bmatrix} agreement : X \\ case : C \end{bmatrix} \end{matrix} \rightarrow \begin{matrix} Pronoun \\ \begin{bmatrix} agreement : X \\ case : C \end{bmatrix} \end{matrix}$$

We can even push categories into structures and rewrite the previous rule as

$$\begin{bmatrix} cat : np \\ agreement : X \\ case : C \end{bmatrix} \rightarrow \begin{bmatrix} cat : pronoun \\ agreement : X \\ case : C \end{bmatrix}$$

Unlike the case for DCGs, unspecified or nonshared features are simply omitted in unification-based grammars. There is no need for an equivalent of the anonymous variable then.

10.4.5 Features and Unification

Unification of feature structures is similar to term unification of Prolog but is more general. It is a combination of two recursive operations:

- Structures merge the set of all their features, checking that identical features have compatible values.
- Variables unify with values and substructures.

Feature structure unification is usually denoted $\cup$.

Unification results in a merger of features as in

$$\begin{bmatrix} feature_1 : v_1 \\ feature_2 : v_2 \end{bmatrix} \cup \begin{bmatrix} feature_2 : v_2 \\ feature_3 : v_3 \end{bmatrix} = \begin{bmatrix} feature_1 : v_1 \\ feature_2 : v_2 \\ feature_3 : v_3 \end{bmatrix}.$$

Variable unification considers features of same name and applies to values, other variables, or recursive feature structures, just as in Prolog but regardless of their position. Here are a couple of examples:

- $[feature_1 : v_1]$ and $[feature_1 : v_2]$ fail to unify if $v_1 \neq v_2$.
- $$\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix} \cup \begin{bmatrix} f_5 : v_5 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_5 : v_5 \end{bmatrix}$$
- $$\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix} \cup \begin{bmatrix} f_5 : X \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_5 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix}$$

In the last example, both features f_2 and f_5 result of the unification of X and are therefore identical. They are said to be re-entrant. However, the structure presentation does not make it clear because it duplicates the X value as many times as it occurs in the structure: twice here. Different structures could yield the same result, as with the unification of

$$\begin{bmatrix} f_1 : v_1 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix} \text{ and } \begin{bmatrix} f_5 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_2 : X \end{bmatrix}$$

where feature f_2 and f_5 have (accidentally) the same value.

To improve the structure presentation, identical features are denoted with a label. Here $[1]$ indicates that f_2 and f_5 are the same:

$$\begin{bmatrix} f_1 : v_1 \\ f_2 : [1] \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \\ f_5 : [1] \end{bmatrix}$$

and Fig. 10.7 shows the corresponding graph.

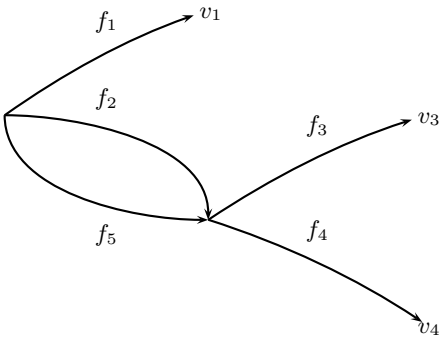

Fig. 10.7. Graph with re-entrant feature structures.

10.4.6 A Unification Algorithm for Feature Structures

Unification of feature structures is close to that of terms in Prolog. However, feature structures provide partial specifications of the entities they represent, while Prolog terms are complete. Feature structure unification is merely a union of compatible characteristics, as in the example

$$[case : nom] \cup [gender : masc] = \begin{bmatrix} case : nom \\ gender : masc \end{bmatrix}$$

where both structures merge into a more specific set. As is, corresponding Prolog terms `struct(case: nom)` and `struct(gender: masc)` would fail to unify.

There are possible workarounds. Given a syntactic category, we could itemize all its possible attributes and map them onto a Prolog term. We would have to assign

each feature to a specific argument rank, for instance, `case` to the first argument, `gender` to the second one, and so on. We would then fill the specified arguments and leave the others empty using the anonymous variable '`_`'. For the example above, this would yield terms

```
struct(case: nom, gender:_)
```

and

```
struct(case: _, gender: masc)
```

that unify properly:

```
?- X = struct(case: nom, gender:_), Y =
struct(case: _, gender: masc), X = Y.

X = struct(case: nom, gender: masc)
Y = struct(case: nom, gender: masc)
```

However, when there are many features and hierarchical structures such a method could be tedious or difficult.

A better idea is to use incomplete lists. Incomplete lists have their tails uninstantiated as `[a, b, c | X]`. Such lists can represent partial structures as `[case: nom | X]` or `[gender: masc | Y]` and be expanded through a Prolog unification. Merging both feature structures is simple. It consists in the unification of `X` with `[gender: masc | Y]`:

```
?- STRUCT = [case: nom | X], X = [gender: masc | Y].
STRUCT = [case: nom, gender: masc | Y]
```

To be more general, we will use the anonymous variable as a tail. Converting a feature structure then consists in representing features as members of a list where closing brackets are replaced by `| _]`. Hence, structures $[case : nom]$ and $[gender : masc]$ are mapped onto `[case: nom | _]` and `[gender: masc | _]`, and their unification yields `[case: nom, gender: masc | _]`. Hierarchical features as:

$$\begin{bmatrix} cat : np \\ agreement : \begin{bmatrix} gender : masc \\ number : sg \\ pers : 3 \end{bmatrix} \\ case : acc \end{bmatrix}$$

are represented by embedded incomplete lists:

```
[cat: np,
agreement: [gender: masc, number: sg, pers: 3 | _],
case: acc | _]
```

Let us now implement the unification algorithm for feature structures due to Boyer (1988). The `unif/2` predicate consists of a fact expressing the end of unification – both structures are the same – and two main rules:

- The first rule considers the case where the heads of both lists represent features of the same name. The rule unifies the feature values and unifies the rest.
- When feature names are different, the second rule uses a double recursion. The first recursion unifies the tail of the first list with the head of the second list. It yields a new list, `Rest3`, which is the unification result minus the head features `F1` and `F2`. The second recursion unifies the rest of the second list with the list made up of the head of the first list and `Rest3`:

```
:- op(600, xfx, ':').

unif(FStr, FStr) :-
  !.
unif([F1:V1 | Rest1], [F1:V2 | Rest2]) :-
  !,
  unif(V1, V2),
  unif(Rest1, Rest2).
unif([F1:V1 | Rest1], [F2:V2 | Rest2]) :-
  F1 \= F2,
  unif(Rest1, [F2:V2 | Rest3]),
  unif(Rest2, [F1:V1 | Rest3]).
```

Consulting `unif/2` and querying Prolog with:

```
?- X = [case: nom | _], Y =[gender: masc | _],
   unif(X, Y).
```

results in:

```
X = [case: nom, gender: masc | _]
Y = [gender: masc, case: nom | _]
```

10.5 Dependency Grammars

10.5.1 Presentation

Dependency grammars form an alternative to constituent-based theories. These grammars describe a sentence's structure in terms of syntactic links – or connections or dependencies – between its words (Tesnière, 1966). Each link reflects a dominance relation (conversely a dependence) between a headword and a dependent word. Examples of simple dependencies tie a determiner to its noun, or a subject noun to its main verb. Dependency links are pictured by arrows flowing from headwords to their dependents (or the reverse).

In noun groups, determiners and adjectives depend on their noun; adverbs depend on their adjective (Fig. 10.8), as in

The very big cat

where the noun *cat* is the head of *the* and *big* and the adjective *big* is the head of *very*. In addition, *cat* is the head – or the root – of the whole phrase.

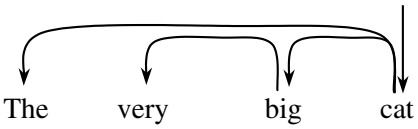

Fig. 10.8. Dependency graph of the noun group *The very big cat*.

Figure 10.9 shows an alternate equivalent representation of the dependencies.

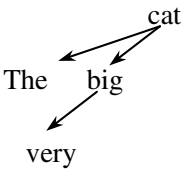

Fig. 10.9. Tree representing dependencies in the noun group *The very big cat*.

According to the classical dependency model, each word is the dependent of exactly one head with the exception of the head of the sentence. Conversely, a head may have several dependents (or modifiers). This means a dependency graph is equivalent to a tree. Figure 10.10 shows a graph representing the structure of a simple sentence where determiners depend on their noun; nouns depend on the main verb, which is the root of the sentence. Tesnière used the word **stemma** – garland or stem in Greek – to name the graphic representation of these links.

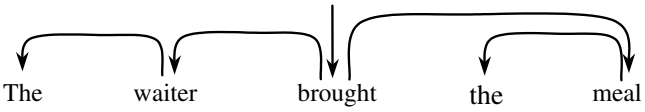

Fig. 10.10. Dependency graph or stemma of the sentence *The waiter brought the meal*.

Although dependency and constituency are often opposed, stemmas embed sorts of constituents that Tesnière called *nœuds*. Deriving a *nœud* from a dependency graph simply consists in taking a word, all its dependents, and dependents of dependents

recursively. It then corresponds to the subtree below a certain word.[1] And in many cases stemmas and phrase-structure trees yield equivalent structures hinting that dependency and constituency are in fact comparable formalisms.

There are a couple of differences, however. One is the importance given to words in dependency grammars. There are no intermediate representation symbols such as phrases of constituent grammars. Syntactic relations involve words only, and nodes in stemmas are purely lexical.

Another difference is that dependency grammars do not need a fixed word order or word contiguity in the *nœuds* to establish links. In that sense dependency theory is probably more suited than constituent grammars to model languages where the word order is flexible. This is the case for Latin, Russian, and German to a lesser extent. Figure 10.11 gives an example with the sentence (Bröker 1998):

Den Mann hat der Junge gesehen
The man$_{/obj}$ *has the boy*$_{/subj}$ *seen* 'The boy has seen the man.'

where positions of noun groups *den Mann* and *der Junge* can be inverted and yield another acceptable sentence: *Der Junge hat den Mann gesehen*. Meaning is preserved because functions of noun groups are marked by cases, nominative and accusative here.

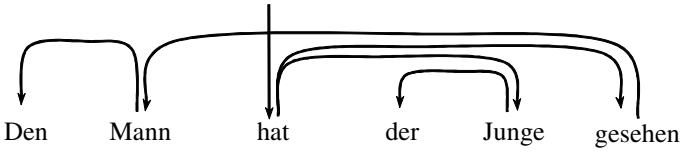

Fig. 10.11. Dependency graph of *Den Mann hat der Junge gesehen*, modified from Bröker (1998).

In the example above, stemmas of both sentences are the same, whereas a phrase-structure formalism requires more rules to take the word order variability into account. Modeling the verb phrase needs two separate rules to describe the position of the accusative noun group

hat <u>den Mann</u> gesehen

and the nominative one

hat <u>der Junge</u> gesehen

$$VP \quad \rightarrow \quad AUX \quad NP \quad V$$
$$\begin{bmatrix} num : N \\ pers : P \end{bmatrix} \quad \begin{bmatrix} num : N \\ pers : P \end{bmatrix} \; [case : acc] \; \begin{bmatrix} tense : pastpart \\ num : N \\ pers : P \end{bmatrix}$$

[1] *Nœud* is the French word for node. It shouldn't be mistaken with a node in a graph, which is a single element. Here a nœud is a whole subtree.

and

$$VP \quad \rightarrow \quad AUX \quad NP \quad V$$
$$\begin{bmatrix} num : N \\ pers : P \end{bmatrix} \quad \begin{bmatrix} num : N \\ pers : P \end{bmatrix} \; [case : nom] \; \begin{bmatrix} tense : pastpart \\ num : N \\ pers : P \end{bmatrix}$$

When word order shows a high degree of freedom, the constituent structure tends to become combinatorial making grammars resorting on it impracticable. For this reason, many linguists, especially in Europe, believe dependency grammar to be a more powerful formalism than constituency. On the contrary, constituency is a property of English that possibly makes dependency less useful in this language.

10.5.2 Properties of a Dependency Graph

After Tesnière, followers extended or modified the definition of dependency grammars. This has led to variations from the original theory. However, some common principles have emerged from the variety of definitions. We expose here features that are the most widely accepted. They result in constraints on dependency graphs. As for constituent grammars, dependency grammars also received formal mathematical definitions.

The first principle is that dependency graphs are acyclic. This means that there is no loop in the graph. Figure 10.12 shows two structures that are not acceptable.

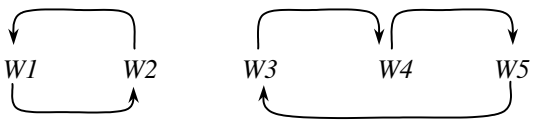

Fig. 10.12. Cyclic dependencies in a graph.

The second principle is that dependency graphs should be connected. This corresponds to the assumption that a sentence has one single head, the root, to which all the other words are transitively connected. Figure 10.13 shows a sentence $w_1 w_2 w_3 w_4 w_5$ with two nonconnected subgraphs.

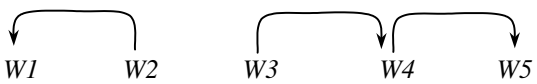

Fig. 10.13. A nonconnected graph spanning sentence $w_1 w_2 w_3 w_4 w_5$.

The third principle is called **projectivity** or **adjacency**. It assumes that all the dependents of a word, direct and indirect, form a contiguous sequence. This means that each pair of words $(Dep, Head)$ in the graph, which is directly or transitively connected, is only separated by direct or indirect dependents of $Head$ or Dep. All

the words in-between are hence dependents of $Head$. In a dependency graph, projectivity results in the absence of crossing arcs.

The projectivity principle is much more controversial that the two first ones. Although less frequent than projective examples, there are many cases of nonprojective sentences. Figures 10.14 and 10.15 show two examples in English and Latin. The sentence *What would you like me to do?* shows a dependency link between *what* and *do*. The projectivity principle would require that *would*, *you*, and *like* are dependent of *do*, which is untrue. The sentence is thus nonprojective.

The Latin verse *Ultima Cumaei venit iam carminis aetas* 'The last era of the Cumean song has now arrived' shows a dependency link between *carminis* and *Cumaei*, but neither *venit* nor *iam* are dependent of *carminis*. We can then better reformulate projectivity as a general principle that suffers exceptions.

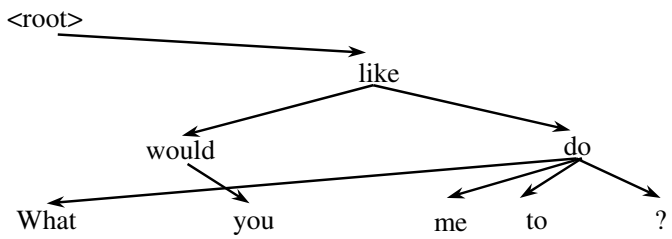

Fig. 10.14. Dependency graph of *What would you like me to do?* After Järvinen and Tapanainen (1997).

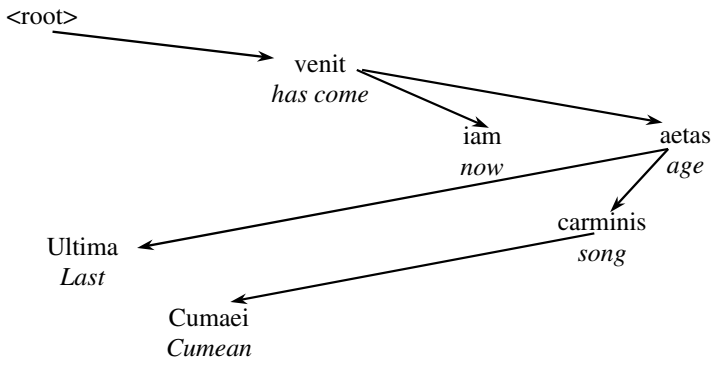

Fig. 10.15. Dependency graph of *Ultima Cumaei venit iam carminis aetas*. 'The last era of the Cumean song has now arrived' (Vergil, *Eclogues* IV.4). After Covington (1990).

10.5.3 Valence

Tesnière and others stressed the importance of verbs in European languages: main verbs have the highest position in the node hierarchy and are the structural centers of sentences. All other types of phrases are organized around them. Hence verbs tend to impose a certain structure to sentences. Connected to this observation, a major claim of dependency grammars is that verbs have specific complement patterns. Verb complements in a broad sense include groups that accompany it: subject, objects, and adjuncts.

Different verbs combine differently with their complements. Certain complements are essential to a verb, like its subject, most of the time. Essential complements cannot be removed from the sentence without making it incorrect or incomplete. Other complements are optional – or circumstantial – like adjuncts that give information on space, time, or manner. Removing them would modify the meaning of the sentence but would still result into something acceptable.

The valence is the number of essential complements of a verb. Using an analogy with chemistry, valence is the attraction power of a verb for potential complements and a specific property of each verb. Just as for chemical elements, the valence is not a strict requirement but rather reflects a sort of most current, stable construction. Common valence values are (Table 10.5):

- 0, for verbs describing weather, *it's raining, snowing*
- 1, corresponding to the subject of intransitive verbs, *he's sleeping, vanishing*
- 2, the subject and object of transitive verbs, *she read this book*.
- 3, the subject and two objects – direct and indirect objects – of ditransitive verbs, *Elke gave a book to Wolfgang, I said it to my sister*.
- 4, the subject, object, source, and destination of certain verbs like *move* or *shift: I moved the car from here to the street* (Heringer 1993).

Table 10.5. Valence values and examples, where $iobject$ denotes the indirect object.

Valences	Examples	Frames
0	*it's raining*	$raining\ []$
1	*he's sleeping*	$sleeping\ [subject : he]$
2	*she read this book*	$read \begin{bmatrix} subject : she \\ object : book \end{bmatrix}$
3	*Elke gave a book to Wolfgang*	$gave \begin{bmatrix} subject : Elke \\ object : book \\ iobject : Wolfgang \end{bmatrix}$
4	*I moved the car from here to the street*	$moved \begin{bmatrix} subject : I \\ object : car \\ source : here \\ destination : street \end{bmatrix}$

From a quantitative definition: the number of slots or arguments attached to a verb and filled with its essential complements, valence is also frequently extended to cover qualitative aspects. It includes the grammatical form and the meaning of these slots. Grammatical properties include possible prepositions and syntactic patterns allowed to each complement of a verb: noun group, gerund, or infinitive. Many dictionaries, especially learners' dictionaries, itemize these patterns, also referred to as **subcategorization frames**. Tables 10.6–10.8 summarize some verb–complement structures.

Table 10.6. Verb–complement structures in English.

Verb	Complement structure	Example
slept	None (Intransitive)	*I slept*
bring	NP	*The waiter brought the meal*
bring	NP + to + NP	*The waiter brought the meal to the patron*
depend	on + NP	*It depends on the waiter*
wait	for + NP + to + VP	*I am waiting for the waiter to bring the meal*
keep	VP(ing)	*He kept working*
know	that + S	*The waiter knows that the patron loves fish*

Table 10.7. Verb–complement structures in French.

Verb	Complement structure	Example
dormir	None (Intransitive)	*J'ai dormi*
apporter	NP (Transitive)	*Le serveur a apporté un plat*
apporter	NP + à + NP	*Le serveur a apporté un plat au client*
dépendre	de + NP	*Ça dépend du serveur*
attendre	que + S(Subjunctive)	*Il a attendu que le serveur apporte le plat*
continuer	de + VP(INF)	*Il a continué de travailler*
savoir	que + S	*Le serveur sait que le client aime le poisson*

In addition, typical complements of a verb often belong to broad semantic categories. The verb *read* generally involves a person as a subject and a written thing as an object. This extension of valence to the semantic domain is called the selectional restrictions of a verb and is exemplified by the frame structure of *gave*:

$$gave \begin{bmatrix} subject : PERSON \\ object : THING \\ iobject : PERSON \end{bmatrix}$$

Chap. 13 gives more details on this aspect.

Table 10.8. Verb–complement structure in German.

Verb	Complement structure	Example
schlafen	None (Intransitive)	*Ich habe geschlafen*
bringen	NP(Accusative)	*Der Ober hat eine Speise gebracht*
bringen	NP(Dative) + NP(Accusative)	*Der Ober hat dem Kunde eine Speise gebracht*
abhängen	von + NP(Dative)	*Es hängt vom Ober ab*
warten	auf + S	*Er wartete auf dem Ober, die Speise zu bringen*
fortsetzen	NP	*Er hat die Arbeit fortgesetzt*
wissen	NP(Final verb)	*Der Ober weiß, das der Kunde Fisch liebt*

10.5.4 Dependencies and Functions

The dependency structure of a sentence – the stemma – generally reflects its traditional syntactic representation and therefore its links can be annotated with function labels. In a simple sentence, the two main functions correspond to subject and object relations that link noun groups to the sentence's main verb (Fig. 10.16).

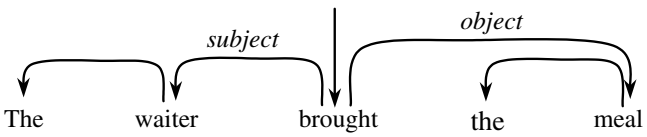

Fig. 10.16. Dependency graph of the sentence *The waiter brought the meal*.

Adjuncts form another class of functions that modify the verb they are related to. They include prepositional phrases whose head is set arbitrarily to the front preposition (Fig. 10.17). In the same way, adjuncts include adverbs that modify a verb (Fig. 10.18).

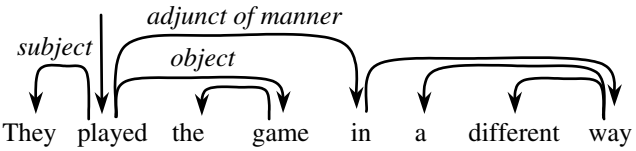

Fig. 10.17. Dependency graph of the sentence *They played the game in a different way*. After Järvinen and Tapanainen (1997).

As for phrase categories in constituent grammars, a fixed set of function labels is necessary to annotate stemmas. Tables 10.9 and 10.10 reproduce the set of dependency functions proposed by Järvinen and Tapanainen (1997). Figures 10.19 and 10.20 show examples of annotations.

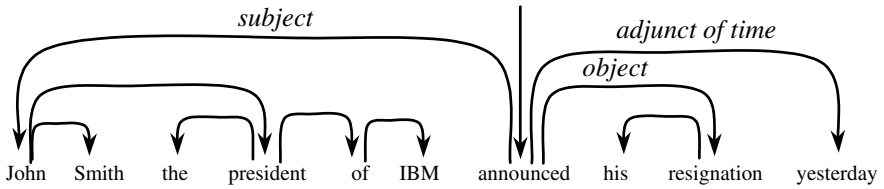

Fig. 10.18. Dependency graph of the sentence *John Smith, the president of IBM, announced his resignation yesterday*. After Collins (1996).

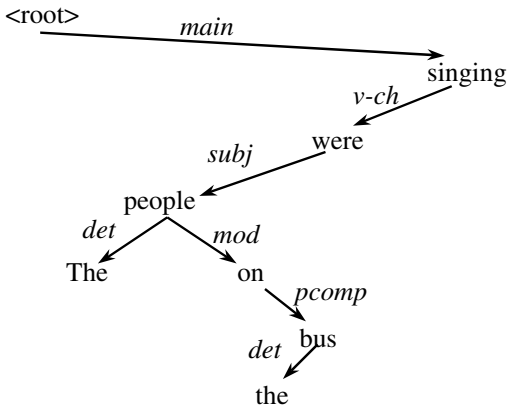

Fig. 10.19. Stemma representing *The people on the bus were singing*. After Järvinen and Tapanainen (1997).

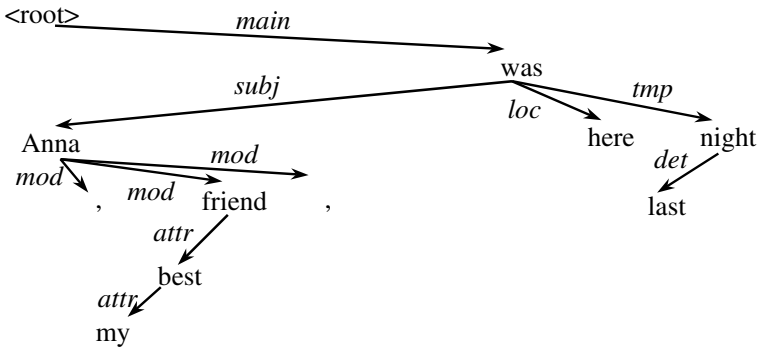

Fig. 10.20. Stemma representing *Anna, my best friend, was here last night*. After Järvinen and Tapanainen (1997).

Table 10.9. Main functions used by Järvinen and Tapanainen (1997) in their functional dependency parser for English. Intranuclear links combine words inside a *nœud* (a consituent). Verb complementation links a verb to its core complements. Determinative functions generally connect determiners to nouns. Modifiers are pre- or postmodifiers of a noun, i.e., dependents of a noun before or after it.

Name	Description	Example
	Main functions	
main	Main element, usually the verb	*He doesn't **know** whether to send a gift*
qtag	Question tag	*Let's play another game, **shall** we?*
	Intranuclear links	
v-ch	Verb chain, connects elements in a complex verb group	*It **may have been being** examined*
pcomp	Prepositional complement, connects a preposition to the noun group after it.	*They played the game **in** a different way*
phr	Verb particle, connects a verb to a particle or preposition.	*He asked me who would look **after** the baby*
	Verb complementation	
subj	Subject	
obj	Object	*I gave him my **address***
comp	Subject complement, the second argument of a copula.	*It has become **marginal***
dat	Indirect object	*Pauline gave it **to** Tom*
oc	Object complement	*His friends call him **Ted***
copred	Copredicative	*We took a swim **naked***
voc	Vocative	*Play it again, **Sam***
	Determinative functions	
qn	Quantifier	*I want **more** money*
det	Determiner	***Other** members will join...*
neg	Negator	*It is **not** coffee that I like, but tea*
	Modifiers	
attr	Attributive nominal	***Knowing** no French, I couldn't express my thanks*
mod	Other postmodifiers	*The baby, Frances **Bean**, was...*
		*The people **on** the bus were singing*
ad	Attributive adverbial	*She is **more** popular*
	Junctives	
cc	Coordination	*Two **or** more cars...*

Table 10.10. Adverbial functions used by Järvinen and Tapanainen (1997). Adverbial functions connect adjuncts to their verb.

Name	Description	Example
	Adverbial functions	
tmp	Time	*It gives me very great pleasure this* ***evening***
dur	Duration	*They stay in Italy all* ***summer*** *through*
frq	Frequency	*I* ***often*** *catch her playing*
qua	Quantity	*It weighed almost a* ***ton***
man	Manner	*They will support him, however* ***grudgingly****...*
loc	Location	*I don't know* ***where*** *to meet him*
sou	Source	*They travelled slowly* ***from*** *Hong Kong*
goa	Goal	*They moved* ***into*** *the kitchen every stick of furniture they possessed*
cnd	Condition	*If I were* ***leaving****, you should know about it*
meta	Clause adverbial	*Will somebody* ***please*** *open the door?*
cla	Clause initial element	***In*** *the view of the authorities, Jones was...*

10.6 Further Reading

Literature on Chomsky's works and generative transformational grammar is uncountable. Most linguistics textbooks in the English-speaking world retain this approach. Recent accounts include Radford (1988), Ruwet (1970), Haegeman and Gueron (1999), Lasnik et al. (2000).

Principles of dependency grammars stem from an old tradition dating back to the ancient Greek and Latin grammar schools. Tesnière (1966) proposes a modern formulation. Heringer (1993) provides a short and excellent summary of his work. Other accounts include Hays (1964), Gaifman (1965), and Mel'cuk (1988). Implementations of dependency theories include the Functional Dependency Grammar (Järvinen and Tapanainen 1997) and Link Grammar (Sleator and Temperley 1993).

Within the work of Tesnière, valence has been a very productive concept although it has not always been explicitly acknowledged. It provides theoretical grounds for verb subcategorization, cases, and selectional restrictions that we find in other parts of this book (Chapter 13). In addition to verbs, valence can apply to adjectives and nouns.

Unification-based grammars were born when Alain Colmerauer designed the *systèmes-Q* (1970) and later the Prolog language with his colleagues. *Systèmes-Q* have been applied in the MÉTÉO system to translate weather reports from English to French (TAUM 1971). MÉTÉO is still in use today. Prolog is derived from them and was also implemented for a project aimed at dialogue and language analysis (Colmerauer et al. 1972). For a review of its history, see Colmerauer and Roussel (1996).

Unification-based grammars have been used in many syntactic theories. The oldest and probably the simplest example is that of Definite Clause Grammars (Colmerauer 1978; Pereira and Warren 1980). Since then there have been many followers.

The most notable include head-driven phrase structure grammars (HPSG, Pollard and Sag 1994) and lexical function grammars (LFG, Kaplan and Bresnan 1982). Unification-based grammars do not depend on a specific syntactic formalism. They are merely a tool that we used with PS rules in this chapter. Dependency grammars can also make use of them. Dependency unification grammar (DUG, Hellwig 1980, 1986) and unification dependency grammar (UDG, Maxwell 1995) are examples. Accounts of unification formalisms in French include Abeillé (1993) and in German, Müller (1999).

Exercises

10.1. Describe step-by-step how the Prolog search mechanism would generate the sentence *the boy hit the ball*, and compare this trace with that of Fig. 10.1.

10.2. Write a Prolog program that converts a DCG grammar into its Chomsky normal form equivalent.

10.3. Write a grammar using the DCG notation to analyze simple sentences: a noun phrase and a verb phrase, where the verb phrase is either a verb or a verb and an object. Write transformation rules that map declarative sentences into their negation.

10.4. Complement PS rules of Exercise 10.3 to parse a possible prepositional phrase within the verb phrase. Write transformation rules that carry out a topicalization of the prepositional phrase.

10.5. Write DCG rules using the gap threading technique to handle sentences and questions of Table 10.2.

10.6. Find a text of 10 to 20 lines in a language you know and bracket the constituents with the phrase labels of Table 10.3.

10.7. Unify $\begin{bmatrix} gen : fem \\ case : acc \end{bmatrix}$ and $\begin{bmatrix} gen : fem \\ num : pl \end{bmatrix}$, $\begin{bmatrix} gen : fem \\ num : pl \\ case : acc \end{bmatrix}$ and $\begin{bmatrix} gen : fem \\ num : sg \end{bmatrix}$, $\begin{bmatrix} gen : masc \\ num : X \\ case : nom \end{bmatrix}$ and $\begin{bmatrix} gen : masc \\ num : pl \\ case : Y \end{bmatrix}$ when possible.

10.8. Unify $\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix}$ and $\begin{bmatrix} f_1 : v_5 \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : v_4 \end{bmatrix} \end{bmatrix}$, $\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix}$ and $\begin{bmatrix} f_1 : Y \\ f_2 : \begin{bmatrix} f_3 : v_3 \\ f_4 : Y \end{bmatrix} \end{bmatrix}$, $\begin{bmatrix} f_1 : v_1 \\ f_2 : X \end{bmatrix}$ and $\begin{bmatrix} f_5 : X \\ f_2 : Y \\ f_1 : Y \end{bmatrix}$.

10.9. Using the unification grammar formalism write rules describing the noun group in a language you know.

10.10. Write a `norm/2` predicate that transforms complete lists into incomplete ones as, for example, `[a, b, [c, d], e]` into `[a, b, [c, d | _], e, | _]`.

10.11. Find a text of approximately ten lines in a language you know and draw the stemmas (dependency links).

10.12. Draw stemmas of sentences in Table 10.9.

10.13. Annotate stemmas of sentences in Table 10.9 with their corresponding functions.