Appeared in *Dependency and Valeny. An International Handbook of Contemporary Research.* Edited by V. Agel, L.M. Eichinger, H.-W. Eroms, P. Hellwig, H.-J. Heringer, H. Lobin. Volume II. pp. 1081-1108. Mouton: 2006

79. Parsing with Dependency Grammars

1. Introduction

This article deals with the analysis of natural languages by computer using dependency grammars as resources. In order to reveal the advantages and disadvantages of the dependency approach, we will proceed from a list of criteria for parsers in general and then discuss the particular choices which are on hand when dependency grammars are used.

1.1. Prerequisites of a parser

From a formal point of view a language is a (possibly infinite) set of character strings. The character strings are structured, that is, elementary strings reoccur in various combinations. It is the task of a grammar to define all of the well-formed strings of a language and to describe their structure. A parser, in turn, is a computer program that accepts or rejects character strings relative to a given grammar and assigns a structural description to each accepted string.

This definition implies the following prerequisites of a parser:

- a grammar formalism,
- a grammar,
- an algorithm.

As a first step, a formal notation for drawing up grammars must be created. This step may take universal principles into consideration. Next, concrete grammars of particular languages must be written in the chosen formalism. Finally, an algorithm must be invented which determines, whether a given input is covered by the given grammar, and if so, which structural description applies to it. In the tradition of Artificial Intelligence, these three tasks are known as "knowledge representation", "knowledge", and "knowledge processing". Different requirements characterize each of these tasks, e.g. expressiveness for the grammar formalism, adequacy for the grammar, efficiency for the algorithm.

1.2. Criteria for classifying parsers

There are numerous technical reports about particular parsers and each conference adds a few more to the list. A substantial amount of proposals adheres to the dependency approach. It can not be the task of this article to survey this huge variety of existing implementations. What we want to achieve is a basic understanding of the essentials and a guidance for evaluating existing parsers as well as for putting together new ones. For that purpose we try to identify the issues that arise in any parser development and record the alternative solutions that exist for each identified task. Our check list comprises the following tasks and choices.

Check list for parsers:

(1) Connection between grammar and parser
- interpreting parser
- procedural parser
- compiled parser

(2) The type of structure to be assigned
- constituency structure
- dependency structure

(3) Grammar specification format
- rule-based specification
- lexicon-based specification
- parameter-valued categories

(4) Recognition strategy
- top-down
- bottom-up
- depth first
- breadth first

(5) Processing the input
- one-pass from left to right or from right to left
- left-associative
- several passes
- proceeding non-continuously

(6) Handling of alternatives
- backtracking
- parallel processing
- well-formed substring table (chart)
- rule filtering
- looking ahead
- probabilistic methods

In the following sections the tasks and choices listed here are briefly described, the impact that the dependency framework has on the solutions is explained and the advantages and disadvantages of this approach are discussed.

The following requirements serve as criteria for the evaluation:

- The parser should be as efficient as possible both in space and time.
- The capacity of the parser must cover the phenomena of natural language.
- Drawing up linguistic resources for the parser should be easy.

2. Survey of parsing tasks and choices

2.1. Connection between grammar and parser

There are three ways of connecting the linguistic data with the parsing procedure: interpreting parsers, procedural parsers, compiled parsers.

(i) Interpreting parser

Grammar and parsing procedure are separate. The grammar is input to the program and interpreted by the parsing routine. The algorithm is based exclusively on the syntax of the grammar formalism, not on the contents of the individual grammar. (Aho/Sethi/Ullman 1986, 3 f.)

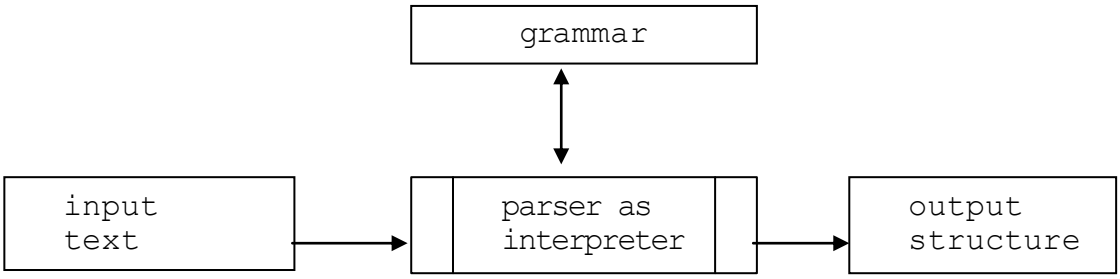

Fig. 79.1: Interpreting parser

The program of an interpreting parser reads the rules and the lexicon of the attached grammar before deciding what output to derive from the input text. The grammatical data is "declarative", that is, it is formulated independently from the aspect of analysis or synthesis. In turn, the same parser can be used for many grammars of different languages. The parser's behavior can be tuned by changing the grammatical data rather than plunging into the code of the program. These advantages of an interpreter are slightly diminished by the fact that the excessive reading of external data makes the program slow.

(ii) Procedural parser

Grammatical data and parsing procedure are no separate categories. The grammar is integrated in the algorithm. The grammar is "procedural", that is, it is formulated as a set of instructions for the analysis of the input language. A new procedure must be programmed for each additional language.

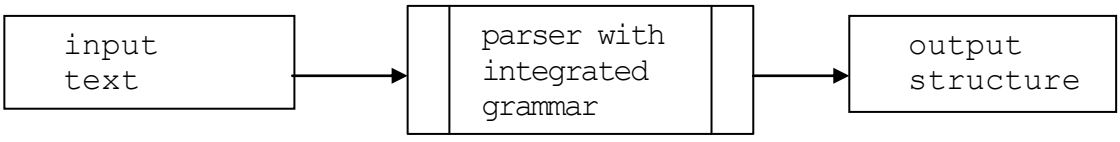

Fig. 79.2: Procedural parser

Procedural parsing played an important role in the history of computational linguistics. Most prominent are Augmented Transition Networks (ATN; Woods 1970). They are programs that start at the beginning of a sentence and try to foresee all alternatives that may occur until the program reaches the end of the sentence. Since at most stages in a sentence there are many possibilities for continuation, a network that

is to cover a whole language is very complex and difficult to maintain. However, ATN parsers can be very fast and have unlimited descriptive power.

Word Expert parsers are the variants of procedural parsers that are closer to the dependency view. The idea is that the individual words in a sentence interact with each other. Every word may behave differently, by its own virtue and under the influence of other words. Consequently each word meaning is implemented as a small program, a so-called word expert. Each expert "actively pursues its intended meaning in the context of other word experts and real-world knowledge" (Small 1987, 161). A prerequisite of this approach is a modular architecture of the parsing system.

It has been argued that procedural parsers are better cognitive models of natural language understanding than the declarative ones. Cohen, Poldrack and Eichenbaum (1997) hold that declarative and procedural knowledge differs in their neuro-anatomic substrates, in their operating characteristics, and in the nature of the representations they use. On the other hand, it is a widely-held opinion that the distinction is not an intrinsic property of linguistic knowledge as such. If it is reasonable to assume that the same linguistic knowledge is employed by humans for listening and for speaking, then the grammar of a language should be kept separate from the computer programs that simulate the two usages, namely one for parsing and another one for generating utterances.

(iii) Compiled parser

The grammar is drawn up in a declarative form and exists independently from the parser. Before execution, the grammar is transformed into a procedural form by a specific program (a parser generator; Aho, Sethi, Ullman 1986, 257 f.). In the final parser, the grammar is part of the program code.

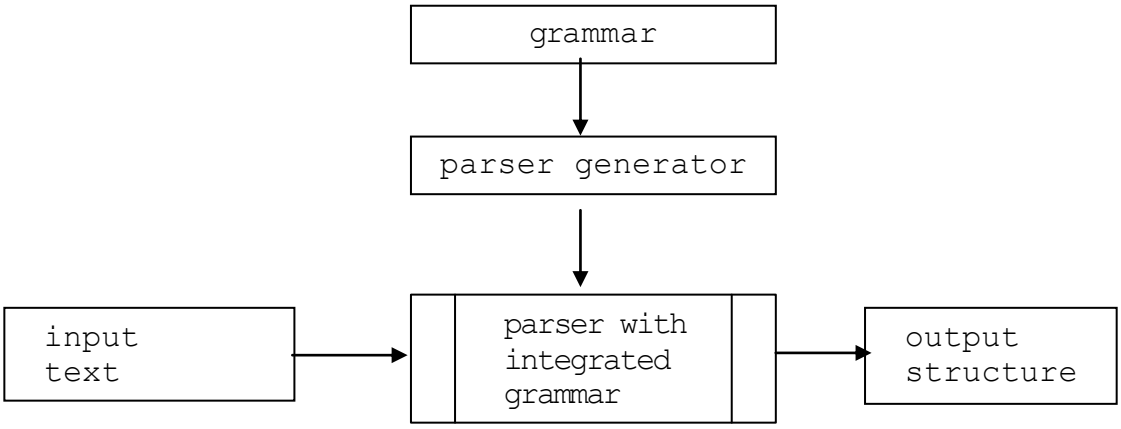

Fig. 79.3: Compiled parser

Compiled parsers embrace the advantages of declarative grammar writing, while they avoid the drawbacks in efficiency at runtime.

2.2. The type of structure to be assigned

It is common in linguistics to represent syntactic structure by trees. A tree is a connected directed acyclic graph. All nodes are connected. There is exactly one node with no incoming edge (the "root"). Every other node has exactly one incoming edge. There may be any number of outgoing edges from any node

but none must lead to a loop, i.e. to an incoming edge of the same node. A tree is projective if no edges cross each other.

The linguistic phenomena that are modeled by trees can be quite different. First of all, trees are used for reflecting constituency structure and dependency structure.

(i) Constituency structure

The nodes in a constituent tree reflect smaller or larger segments of the input string. The edges represent the composition of larger phrases from elementary ones. Each level in a phrase structure tree denotes another phase of segmentation of the original sentence or phrase. The topmost node covers the whole expression; subordinated nodes cover smaller parts of the same expression and so on.

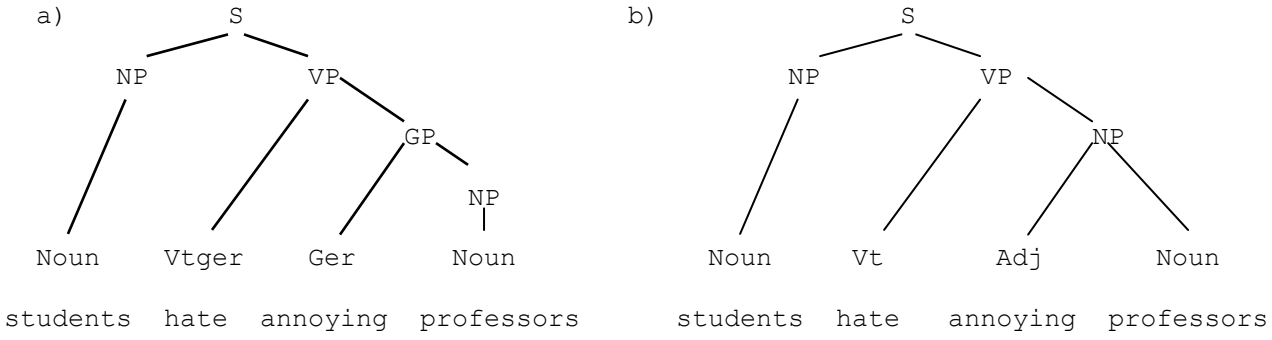

Fig. 79.4: Constituency trees

(ii) Dependency structure

All units that correspond to the nodes in a dependency tree are elementary segments of the input string. Each arc denotes the syntagmatic relationship between an elementary segment and one of its complements or adjuncts (Tesnière 1959). All nodes belong to the same unique and exhaustive segmentation. Larger segments are present implicitly as a combination of elementary segments in the tree.

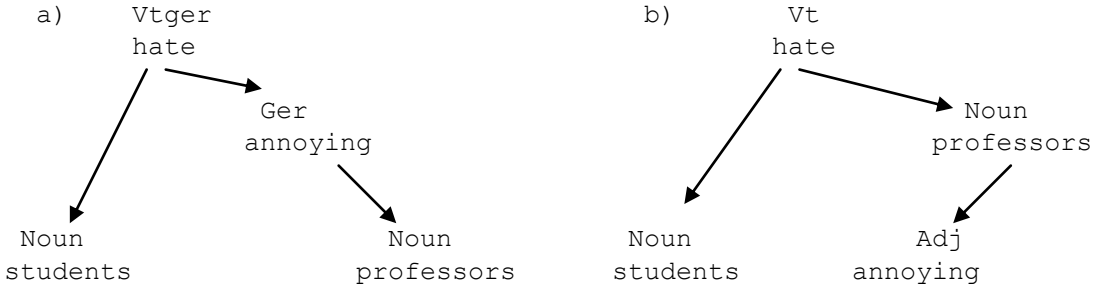

Fig. 79.5: Dependency trees

The constituency model is focused on the concatenation of strings; the dependency model is focused on the functional complementation of syntactic units. Usually this difference has an impact on the way the grammar is conceived and on the applicable parsing algorithms. However, both approaches are compatible if the notion of “head" is introduced in the constituency framework. In a head-marked

constituency grammar, one of the immediate constituents of each constituent must be the head. It can be stipulated in addition that the head constituent must be an individual word (cf. Jackendoff 1977).

The following criteria for heads can be found in Zwicky (1985): Often there is one word in the construction which has a distribution parallel to the construction as a whole. It is semantically the generic term of the whole construction. Typically, this word bears the morpho-syntactic marking indicating the relationship of the construction to other constructions. It requires the existence of other members in the construction and determines their particular inflectional form. For example *hate* is a concept that necessarily involves *someone who hates* and something *hated.* The latter can be an object or an event. Hence, a subject and a direct object or a gerund construction with known morpho-syntactic properties can be predicted with almost a hundred percent certainty when the verb *hate* turns up. The lexical filling of the subject (e.g. *students*) and the object phrase (e.g. *professors*) are by no means arbitrary either. It seems that ultimately it is lexical semantics that determines syntax.

The main cause of inefficiency of parsers is the generation of useless intermediate results. Individual words as heads of phrases are a remedy against such bad performance because they maximally constrain the syntagmatic relationships.

What are the advantages of dependency representations as compared to constituency-based head-marked grammars? Most important is the fact that grammatical functions can be integrated more easily in the dependency framework. The hierarchy of nodes in a dependency structure is designed in a way that all substitutions of a node preserve the grammatical function. For example, all the surface realizations of the subjects in the sentences of (1) are a result of the expansion of the same complement node depending on the verb *surprise.*

(1) a. *His reactions surprised me.*
b. *That he should make such mistakes surprised me.*
c. *To see him here surprised me.*
d. *What surprised you?*

The interrogative pronoun *what* in (1) d. carries the same functional label *subject* as the subject portions in (1) a. - c. that are the answers to sentence d. The functional label yields the relevant portions for answering the question, although the subjects in (1) are instances of quite different constituents: a noun phrase, a *that*-clause, and an infinitive construction.

Last but not least, the paths along which features must be propagated for unification are longer in constituent trees than in dependency trees. For example, in Figure 79.4 four edges must be passed in order to check the agreement between *students* and *hate* while just one edge suffices in Figure 79.5. This is a considerable gain in efficiency.

Still, both constituency and dependency trees might be too limited to model all phenomena of natural languages, especially those that relate to linearity rather than to hierarchy. Discontinuous constituents, coordination, ellipses are cases in point. In spite of these deficiencies, trees are likely to survive in computational linguistics because of their technical convenience. The advantage of trees from a computational perspective is the fact that there is exactly one path from any node to any other node and, hence, there is always an efficient way to access any information distributed in the tree. The further development of dependency grammar is probably moving into the direction of enrichments of the trees by new types of attributes decorating the nodes (cf. HSK 25.1, 44, chapter 6).

2.3 The Grammar specification format

It is a different matter to decide what kind of formal structure should reflect the grammatical relationships and to find a way to define and build the structural representation. There are two principal

approaches: the rule-based specification of grammars which is accompanied by a sentence-oriented view on syntax and the lexicon-based specification which implies that words ultimately determine syntactic constructions. Rule-based and lexicon-based specifications of grammars are to a certain extent independent of the constituency-dependency distinction. The matrix in Figure 79.6 displays the four possible combinations (Hellwig 1978, 70 f.).

	rule-based	lexicon-based
constituency	G1	G3
dependency	G2	G4

Fig. 79.6: Grammar types with rule-based or lexicon-based specification

Let as briefly compare the four formalisms.

(i) Rule-based specification

The grammar consists of a set of so-called production rules. All of the well-formed character strings of a language are generated by rewriting symbols according to the rules. The essence of this approach is the assumption of an abstract structure which provides the honeycombs for the words and phrases in the concrete case. Syntactic relationships are a matter of the rules, while the lexicon provides simple bricks.

G1 is the type of Generative Phrase Structure Grammar introduced in Chomsky (1957, 26-27). Chomsky's example is

(2) *the man hit the ball*

Figure 79.7 and 79.8 show the G1-rules and the associated phrase marker.

(i) *Sentence* → NP + VP

(ii) NP → T + N

(iii) VP → *Verb* + NP

(iv) T → *the*

(v) N → *man, ball,* etc.

(vi) Verb → *hit, took,* etc

Fig. 79.7: Phrase structure rules

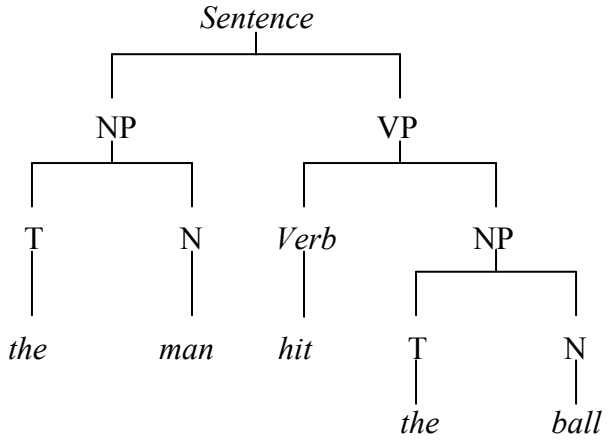

Fig. 79.8: Constituency tree generated with phrase structure rules

G2 is the type of early dependency grammars as they have been suggested by Hays (1964) and Gaifman (1965). Figure 79.9 displays the rules needed for sentence (2).

(i)	V (N * N)	V = {*hit, took*, etc.}
(ii)	N (T *)	N = {*man, ball*, etc.}
(iii)	T (*)	T = {*the*}

Fig. 79.9: Rules of a dependency grammar

The symbol in front of the brackets is the head; the symbols within the brackets represent the dependents. The asterisk marks the linear position of the head among its dependents. The categories in the rules are parts-of-speech. Parts-of-speech are assigned to the words in the lexicon. There are no categories for larger constituents. Categories that can be the head in a sentence and, hence, can serve as starting points in the generation process must be specially marked. We use an underscore for this purpose. Figure 79.10 shows the tree that results from the applications of the rules.

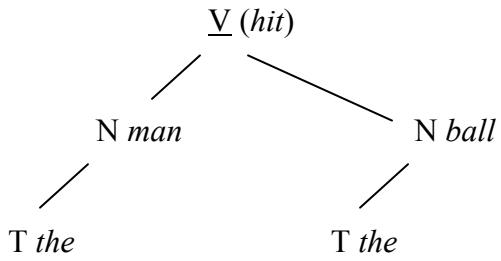

Fig. 79.10: Dependency tree generated with dependency rules

(ii) Lexicon-based specification

G3 is a lexicon-based constituency grammar. A case in point is Categorial Grammar (cf. Bar-Hillel/Gaifman/Shamir 1964). The grammar is simply a set of words associated with categories as displayed in Figure 79.11.

the	N / N
man	N
ball	N
hit	(N \ S) / N
took	(N \ S) / N

Fig. 79.11: The lexicon of a categorical grammar

If a lexical category includes a slash and if the neighboring constituent has the same category as the category specified below the slash, then a subtree can be built with the symbol above the slash as the new joint category. The process is repeated. Brackets are removed when they coincide with the boundaries of constituents. In this way, the constituency tree shown in Figure 79.12 is built on top of the sequence of words in (2).

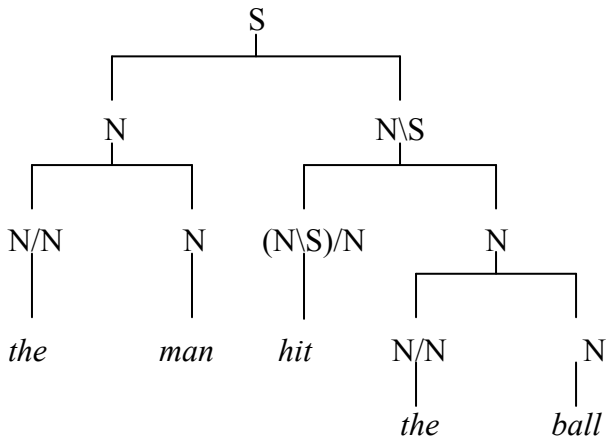

Fig. 79.12: Constituency tree built with a categorical grammar

G4 is the type of dependency grammar that solely relies on valency descriptions in the lexicon. The lexicon that can cope with example (2) is introduced in Figure 79.13.

the	T
man	(T) N
ball	(T) N
hit	(N) V (N)
took	(N) V (N)

Fig. 79.13: The lexicon of a valency grammar

Symbols in brackets represent dependents; the symbol outside the brackets is the head. The control is exerted by the lexicon. If the category of a neighboring constituent is the same as an expected dependent in a valency description then a dependency subtree can be formed. The tree resulting from our example is shown in Figure 79.14.

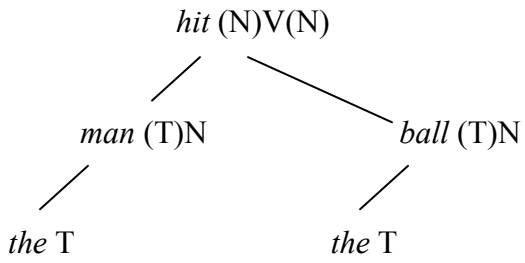

Fig. 79.14: Dependency tree built with a valency grammar

The isomorphic trees of the two constituency and the two dependency approaches (compare Figure 79.8 with Figure 79.12 and Figure 79.10 with Figure 79.14) prove that rule-based grammars and lexicon-based grammars can under certain provisions be strongly equivalent. While the lexical categories of the lexicon-based grammars are transparent with regard to the combinatorial potentials of the lexical item, the rule-based variants use categories in the lexicon which are syntagmatically opaque. However, rule-based grammars can not do without the lexical valency information either. In order to avoid wrong lexical selection, Chomsky (1965, 94) introduced the so-called "strict subclassification", that is, each lexical item is provided with information about the constituency structure into which it fits. A similar subclassification is necessary for rule-based dependency grammars of the Hays-Gaifman type. The subclassification of lexical items in a rule-based grammar looks very much like the entries of the corresponding lexicalized grammar. It seems that one does not loose very much if one does without rules altogether.

(iii) Parameter-valued categories

A few words about the format of categories should conclude this section on the specification format of grammars. While rule-based systems certainly profit from parameter-valued categories, the lexicalized versions of grammars do not work at all without them.

A word as well as a phrase can be classified according to a host of features. It is advantageous to use complex categories that are composed of "bundles" of subcategories. The essence of parameter-valued categories lies in the distinction between feature type and concrete features. The feature type, often called "attribute", is treated as a parameter that may assume "values", i.e. the concrete features. Since the parameter is always made explicit, generalizations can be expressed by means of attributes alone, e.g. the agreement of phrases in number or gender can be stipulated without mentioning singular and plural, masculine, feminine and neuter. The mechanism of instantiation and calculating agreement is often referred to as "unification". In contrast to conventional pattern matching, there is no pre-established assignment of one item as pattern and the other one as instance. The instantiation of an attribute that is to be shared by two constructs can go in both directions and the directions may differ for different attributes in the same category. Agreement with respect to many properties may be propagated in this way across long distances up and down in trees. In principle, each attribute can reflect another dimension. Many forms of ambiguity can be represented compactly and elegantly and processed efficiently. Huge trees can grow from lexical seeds formulated with complex categories. The whole grammar adopts the character of an equation rather than that of a generative device (Kratzer/Pause/von Stechow 1974, Hellwig 1980, Shieber 1986).

There are many variants of this kind of grammar writing. Some authors base their "typed-feature structures" in formal logic, others rely on the concept of "constraints" in recent mathematical frameworks. Constraint programming (CP) is actually an emergent software technology for declarative description and effective solving of large, particularly combinatorial, problems (Barták 1999, Koller/Niehren 2002). Enrichments of syntactic trees with non-recursive features have the formal properties of attribute grammars in the sense of Knuth (1968). It depends on the attributes whether an

attribute grammar is context-free or beyond. Attributes may be introduced that cover phenomena of natural languages that are context-sensitive in nature.

2.4. Recognition strategy

We now turn to the algorithms that enable the computer to reveal the structure inherent in a character string. Let us, for the time being, concentrate on constituency grammars (cf. Figure 79.7 and Figure 79.8).

The root of a constituency tree is often known in advance. It is the category of the constituent under consideration, in most cases a sentence (*S*). The exterior nodes of the tree ($t_1, \ldots t_n$) are also known. They are identical with the elementary units of the input string and are identified and classified by means of the lexicon (usually in terms of parts-of-speech). What is unknown is the interior coherence of the tree. Figure 79.15 illustrates the point of departure. The area of the unknown in the tree is dotted.

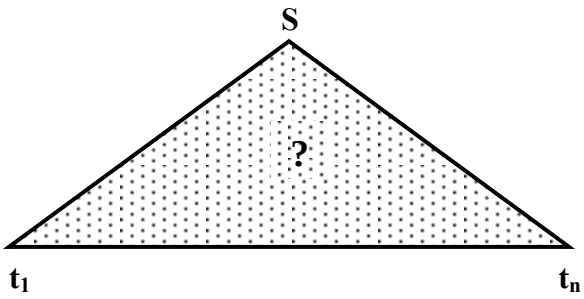

Fig.79.15: The search tree

The interior of the tree has to be filled with the aid of the grammar. The following strategies differ in how they proceed from the known to the unknown.

(i) Top-down analysis

The root of the search tree is the starting point. Proceeding from top to bottom, nodes are added according to the grammar rules, until the sequence of terminal nodes is reached. For example, after the rule $S \rightarrow NP + VP$ has been applied, we have the situation of Figure 79.16.

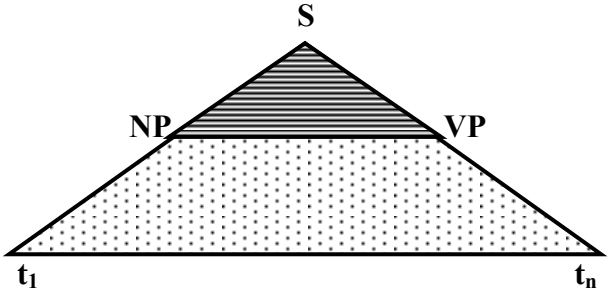

Fig. 79.16: Top-down analysis

The top-down strategy makes use of the production rules in the direction from left to right. Connecting a symbol in the tree occurring on the left side of the rule with the symbols on the right side of the rule is

called "expansion". The procedure is also said to be "expectation driven" since the new symbols are hypotheses of what units will be found in the input string.

(ii) Bottom-up analysis

Here, the lexical elements are the point of departure. In accordance with the grammar, new nodes are linked to old ones, thus proceeding from bottom to top until the root of the tree has been reached. After a rule $NP \rightarrow T + N$ has been applied somewhere in the sentence, the situation is like Figure 79.17.

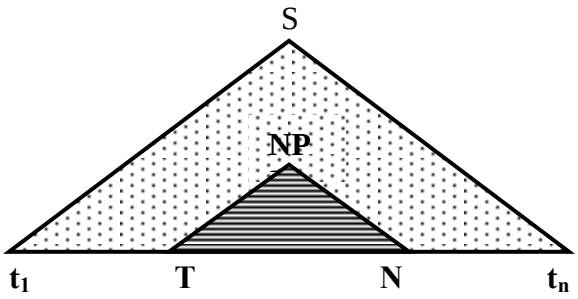

Fig. 79.17: Bottom-up analysis

The bottom-up strategy makes use of the production rules in the direction from right to left. The symbols in the tree corresponding with the right hand side of a rule are connected with the symbol on the left side of the rule. This method is called "reduction". The procedure is also said to be "data driven" since only those categories that are present in the input lead to the application of a rule.

(iii) Depth first

The left-most (or the right-most) symbol created in the previous step is always processed first, until a terminal symbol has been reached (or, in the case of bottom-up analysis, the root of the tree has been reached). This strategy is reasonable in combination with a top-down analysis since the terminal nodes must be inspected at the earliest possible stage in order to verify or disprove the derivation. In the following tree, the left context of the constituent A is already verified by the derivation of the terminals t_1 till t_m. Hence, one can be sure that the expansion of A is worth the effort.

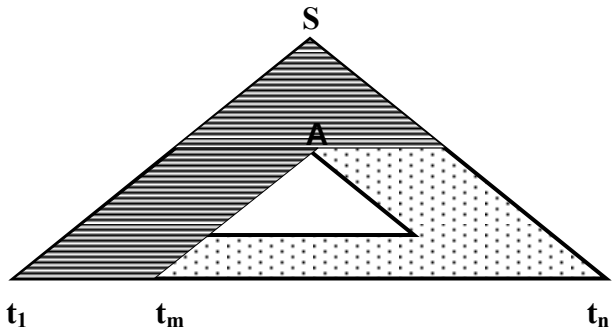

Fig. 79.18: Depth first analysis

(iv) Breadth first

Here the symbols are processed first that are on the same level of expansion or reduction. As a consequence, the tree is filled in its entire width. This is a useful organization for a bottom-up analysis since all of the immediate constituents must have reached the same level and be complete with respect to

their own constituents before they can be linked to a larger constituent. A stage in such a breadth-first analysis is illustrated by the tree in Figure 79.19:

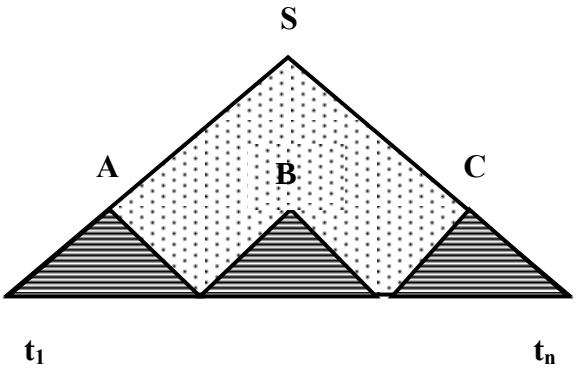

Fig. 79.19: Breadth first analysis

Both top-down analysis and bottom-up analysis have advantages and disadvantages. The advantages of top-down parsers result from the fact, that these parsers rely, in a sense, on expectations what comes next. The advantages of bottom-up parsers result from the fact that they are data-driven, i.e. they will not expand a rule that will have no chance with respect to the actual input. The disadvantage is that a top-down parser may generate expansions that will never be met by the terminal elements while a bottom-up parser may construct constituents from terminal elements which are obsolete in the broader context. Ideally both principles should be combined, which is the case in some advanced parsers like the top-down chart parser with divided productions (Earley 1970, Aho/Ullman 1972, 320 f.) or the table-controlled shift-reduce parser (Aho/Ullman 1977, 198-248; Tomita 1986).

The recognition strategies mentioned above can be carried over to dependency trees, although linguistically, the distinction means something different here. In the dependency framework, the fundamental recognition strategy is slot filling. Top-down analysis means proceeding from the governing to the dependent element while bottom-up analysis proceeds from the dependent to the governing element. Whilst the first alternative can be said to be expectation-driven, both procedures are more or less data-driven, since all of the nodes in a dependency tree represent lexical elements. If you take the dependency approach, the nodes in the interior of the search tree in Figure 79.15 are also known in advance: there must be exactly one node for each word. What is unknown, are the links between the nodes. Searching for links can start at the top, at the bottom or anywhere in the middle of the tree and proceed in various directions.

2.5 Processing the input

Another issue is how the parser proceeds through the input. The following possibilities exist, which can partly be combined.

(i) One-pass from left to right or from right to left

In this case the transition of the parser from one position in the input to the next is the primary principle of control. If the parser moves from left to right, sentence boundaries may be crossed readily and entire texts can be parsed on-line.

(ii) Left-associative

In addition to processing the input from left to right, it may be stipulated that the next constituent is accepted only if it is compatible with the analysis of the complete context on the left of it. A parser that checks this property is called left-associative. An incremental parser is one that connects new input with the left context on a word-by-word basis. It is not easy to fulfill the left-association requirement, especially in a dependency framework, but it is something worth striving for. A left-associative parser mirrors the incremental understanding of a sentence by humans, which is obviously very effective.

(iii) Several passes

Several passes through the input may occur, because other control principles guide the process. Cascaded applications of programs may create a series of intermediate representations. Each application replaces or adds something which is then the input to the next application. Principles, like "easy-first parsing" may be applied. Parsing with finite-state-transducers is an example (Abney 1996; Roche 1997).

(iv) Proceeding non-continuously

Processing the input may not be continuous at all but may start from several points in the input and proceed from there to the left and to the right. This kind of "island-parsing" is well-matched to the valency principle of dependency grammars: Start parsing with the main verb, then look for subjects and objects and so forth. Such a strategy could also be useful for partial parsing and information extracting tasks. Other parsers try to identify the edges of a constituent first and then fill the inner space, e.g. parsers that are built on top of a part-of-speech tagger (Ait-Mokhtar/Chanod 1997). There is plenty room for inventions.

2.6 Handling of alternatives

Alternatives occur while filling the search tree, because there is not (yet) enough context or because the input is ambiguous. If alternative transitions to the next state are possible in the course of parsing the following actions can be taken.

(i) Backtracking

The alternative grammar rules are processed one after the other. The path of one particular alternative is pursued as far as possible. Then the parser is set back to the last road's fork and the path of the next alternative is tracked. This procedure is continued until all alternatives have been checked. The disadvantage of this method is the loss of all information gathered in course of paths that have been given up eventually. Since a branching of rules at a higher level does not imply that everything is different on the lower level, the same work might be done over and over again.

(ii) Parallel-processing

All alternatives are pursued simultaneously. As a consequence, the program's control has to cope with several concurrent results or paths. If the hardware does not support real parallel processing, the steps within the parallel paths are in fact processed consecutively. Without further provisions this method is inefficient, because it always arrives at the worst case of checking all possible combinations.

(iii) Well-formed substring table (chart)

The parser stores all intermediate results in a data structure called “well-formed substring table" or “chart". Each entry in the table contains an identification of an input segment, e.g. the positions of the segment's start and end, and the information so far created by the parser for this particular segment, e.g. its grammatical category or even a complete parse tree. Every intermediate result can be re-used in any new combination. The parser checks for any new segment if it combines with any of the old segments in the chart. At the end of the execution the table should contain at least one segment that covers the whole input. The structural description associated with this segment is the parsing result. This organization guarantees that no work is done more than once.

(iv) Rule filtering

Another strategy is to reduce the number of alternatives when they arise. The choice of a rule of the grammar can be made dependent on various heuristics. For example, a table could be consulted, which contains all the derivable terminals given a particular non-terminal category. Additional attributes in the grammar, including semantic features, may narrow down the applicable rules.

(v) Looking ahead

Looking ahead is another kind of avoiding wrong decisions (Aho/Ullman 1977, 198-248; Hellwig 1989, 389 f.). The choice of a rule depends on the next *k* categories in the input beyond the segment to be covered by the rule. A function “first of follow” is necessary that yields all the first elements of all the constituents that can follow a given constituent according to the rules of the grammar. The idea is to delay the application of a rule until the next element occurs in the input that certainly cannot belong to the current construction. For example, the *subject-NP* in a sentence is constructed only after the first element of the *VP* has shown up.

If there is only one possibility left, after taking into account the next *k* categories, the grammar has the LL(k) or LR(k) property and the parser can work deterministically. Obviously, natural languages as a whole do not have the LL(k) or LR(k) property, since ambiguity is their inherent feature. Nevertheless, it should be possible to process large portions of text deterministically if enough context is taken into account.

(vi) Probabilistic methods

If there is a conflict, it is a reasonable action to try the most frequent rule first. The problem is the acquisition of frequency data for rule applications. Obviously, the frequencies of rules must be gathered from corpora automatically. Since this is impossible for high-level intertwining rules, the common procedure is to simplify the grammar. In the extreme case, the syntactic relationships are reduced to *n*-grams, i.e. the sequences of *n* words or *n* categories. The frequency of *n*-grams can be counted in a corpus. Probabilistic parsing, then, is the task to compute the probability of a sentence by optimizing the probabilities of the contained *n*-grams. This method has certain merits in the framework of speech recognition. It has been noted however that statistics of lexical dependencies would be much more adequate than the simple *n*-gram model (Charniak 2001; Jurafsky, Martin 2000, 447 f.).

3 Dependency parsing with a head-marked phrase structure grammar

Context-free phrase-structure grammars have been studied extensively since half a century and efficient parsing algorithms are available for them. Could the existing parsing technology for phrase structures be utilized for dependency parsing as well? The goal of this chapter is to settle this question. One way of understanding a theory is by observing how it functions in practice. Therefore the following presentation takes the form of examples.

Rules:		Lexicon:	
(R-1)	S → NP + **VP**	Det	= {*the, their*}
(R-2)	NP → **Noun**	Adj	= {*loving, hating, annoying, visiting*}
(R-3)	NP → Det + **Noun**	Noun	= {*relatives, students, professors*}
(R-4)	NP → Adj + **Noun**	Noun	= {*love, hate*}
(R-5)	NP → **Pron**	Pron	= {*they*}
(R-6)	VP → **Vt** + NP	Vt	= {*love, hate, annoy, visit*}
(R-7)	VP → **Vtger +** GP	Vtger	= {*love, hate*}
(R-8)	GP → **Ger** + NP	Ger	= {*loving, hating, annoying, visiting*}

Fig. 79.20: A grammar fragment of English

The grammar fragment in Figure 79.20 covers ambiguous sentences like the one in Figure 79.4.

(3) *students hate annoying professors*

The parser must reconstruct the derivation of the sentence from the starting symbol *S* on the basis of the rules in Figure 79.20. Possible recognition strategies are top-down, bottom-up, depth-first, and breadth first. Alternatives can be handled by backtracking, parallel processing, a well-formed substring table or looking-ahead.

3.1. Top-down recognition

A top-down depth-first phrase-structure parser with backtracking can be implemented as follows (Aho/Ullman 1972-73, 289 f.). The core is a working space for replacements according to the rules of the grammar. At the beginning the working space just contains the starting symbol. A position variable points to the first word in the input. There is an empty stack for "snapshots" and a key for recording successfully applied rules called "parse key".

Two procedures make up the parser: expansion and recognition.

Top-down phrase-structure parser with backtracking

Expansion: Loop through the rules. If the leftmost symbol in the working space is the same as the symbol on the left hand side of the rule then replace the symbol in the working space by the symbols on the right hand side of the rule. Add the number of the applied rule to the parse key. If a subsequent rule in the grammar starts with the same symbol then put a snapshot of the current state on a stack. The snapshot consists of the current derivation in the working space, the current word position, the parse key, and the next rule to try in case of a dead-end. Repeat the expansion as long as there is a rule for expanding the new leftmost symbol. If there is no rule any more then try Recognition.

Recognition: If the leftmost symbol in the working space is the same as the lexical category of the word at the current position then remove the symbol from the working space, move the position to the next word, and try Expansion for the new leftmost symbol. If the symbol in the working space cannot be recognized, a former rule application might have been incorrect. Therefore, restore the state recorded in the most recent snapshot, that is, store the old derivation in the working space, set the current position to the old position, and restore the old parse key. Remove the last snapshot from the stack and then try the next rule for Expansion. If the end of the input is reached and the working space is empty then output the parse key. If the working space is not empty then a former decision might have been wrong. In this case backtracking is necessary. The same is true if more than one result is possible due to ambiguous input. Restore previous states as long as there is still a snapshot on the stack.

Consulting the lexicon for sentence (3) yields the following, partly ambiguous, categories:

(4) *Noun + Vt/Vtger + Adj/Ger/Noun + Noun*

The stages of the parser while processing input (4) are illustrated in Figure 79.21.

P	Category at P	Derivation	Parse key	Explanation
1	Noun	S	-	start
1	Noun	NP+VP	1	snapshot 1; expansion R-1
1	Noun	Noun + VP	1,2	expansion R-2
2	Vt/Vtger	VP	1,2,*	"Noun" recognized
2	Vt/Vtger	Vt + NP	1,2,*,6	snapshot 2; expansion R-6
3	Adj/Ger/Noun	NP	1,2,*,6,*	"Vt" recognized
3	Adj/Ger/Noun	Noun	1,2,*,6,*,2	snapshot 3; expansion R-2; recognition fails
3	Adj/Ger/Noun	NP	1,2,*,6,*	backtracking, restore snapshot 3
3	Adj/Ger/Noun	Det Noun	1,2,*,6,*,3	snapshot 3; expansion R-3; recognition fails
3	Adj/Ger/Noun	NP	1,2,*,6,*	backtracking, restore snaphot 3
3	Adj/Ger/Noun	Adj Noun	1,2,*,6,*,4	expansion R-4
4	Noun	Noun	1,2,*,6,*,4,*	"Adj" recognized
5	Noun	-	1,2,*,6,*,4,*,*	"Noun" recognized; first result found

Fig. 79.21: Stages of a top-down depth-first phrase-structure parser with backtracking

The last row of Figure 79.21 is in fact not the end of the parsing process. In order to find the second reading of sentence (3) the parser has to backtrack to snapshot 2 und try another *VP*-expansion.

The described algorithm consumes the input categories from left to right until the workspace is empty. It is a pure recognition algorithm. In order to output a structural description of the input sentence, we chose to maintain a key consisting of the numbers of the successfully applied rules. A parse tree can be generated by expansion of the start symbol according to these rules. Figure 79.22 illustrates the generation of a structural description for the key [*1,2,*,6,*,4,*,*]*. An asterisk in the key denotes lexical substitution.

(S)	
(S (NP) (VP))	1
(S (NP (Noun *students*))(VP))	1,2,*
(S (NP (Noun *students*))(VP (Vt *hate*) (NP)))	1,2,*,6,*
(S (NP (Noun *students*))(VP (Vt *hate*) (NP (Adj *annoying*) (Noun *professors*))))	1,2,*,6,*,4,*,*

Fig. 79.22: Generation of a phrase structure description

Schematic backtracking is the least efficient strategy. The same result can be reached more effectively by other top-down parsers, e.g. the Early chart parser, Kuno's Predictive Analyzer or Augmented Transition Networks (ATN), to name only a few (cf. Hellwig 1989; Lombardo/Lesmo1996).

3.2 Bottom-up recognition

The opposite of the top-down depth-first strategy is the bottom-up breadth-first strategy. A simple algorithm for the latter is the following (Dietrich/Klein 1974, 70 f.).

Bottom-up breadth-first Parser

Reduction: Start with a record containing the sequence of categories assigned to the words in the input. This is the current record. Loop through all rules. The rightmost symbol on the right hand side of the rule is called the "handle". Loop through all symbols in the current record. If the handle is equal to a symbol in the record, then check if the rest of the symbols in the rule matches with symbols in the record. If so, then create a new record with the matching symbols replaced by the symbol on the left side of the rule. Any new record that is identical to an existing one is discarded.

As long as new records have been created in the course of reductions, make one after the other the current record and try the Reduction procedure on them. Parsing is successful if at the end at least one of the records contains the start symbol alone.

Figure 79.23 displays the records created for the same input and using the same grammar as in the top-down example above.

No.	Reduction:	Rule:	Applied to:
1	Noun + Vt/Vtger + Adj/Ger/Noun + Noun	-	-
2	NP(Noun) + Vt/Vtger + Adj/Ger/Noun + Noun	2	1
3	Noun + Vt/Vtger + Adj/Ger/Noun + NP(Noun)	2	1
4	Noun + Vt/Vtger + NP(Adj + Noun)	4	1
5	NP(Noun) + Vt/Vtger + Adj/Ger/Noun + NP(Noun)	2	2
*	NP(Noun) + Vt/Vtger + Adj/Ger/Noun + NP(Noun)	2	2
6	NP(Noun) + Vt/Vtger + NP(Adj + Noun)	4	2
*	NP(Noun) + Vt/Vtger + Adj/Ger/Noun + NP(Noun)	2	3
7	Noun + Vt/Vtger + GP(Ger + NP(Noun))	8	3
*	NP(Noun) + Vt/Vtger + NP(Adj + Noun)	2	4
8	Noun + VP(Vt + NP(Adj + Noun)	6	4
9	NP(Noun) + Vt/Vtger + GP(Ger + NP(Noun))	8	5
10	NP(Noun) + VP(Vt + NP(Adj + Noun))	6	6
*	NP(Noun) + Vt/Vtger + GP(Ger + NP(Noun))	2	7
*	NP(Noun) + VP(Vt + NP(Adj + Noun)	2	8
11	NP(Noun) + VP(Vtger + GP(Ger + NP(Noun)))	7	9
12	S(NP(Noun) + VP(Vt + NP(Adj + Noun)))	1	10
13	S(NP(Noun) + VP(Vtger + GP(Ger + NP(Noun))))	1	12

Fig. 79.23: Records of a bottom-up breadth-first parser

Immediate constituents are put into brackets rather than being replaced in Figure 79.23. The bracketed representation can serve as a complete structural description at the end of the procedure. All expressions in brackets are supposed to be invisible to the algorithm, though.

The records marked by an asterisk in Figure 79.23 are discarded because they already exist. Creating, checking and discarding useless records is time consuming. Hence this type of a parser can not be recommended in practice. The same result can be produced more efficiently by other bottom-up parsers, e.g. the Cocke-Kasami-Younger chart parser or the Tomita table-controlled shift-reduce parser with look-ahead (cf. Hellwig 1989).

3.3 Creating dependency output

It is possible to create dependency output with constituency parsers. The precondition is a head-marked phrase structure grammar (Hays 1966, 79). In Figure 79.20, the constituents which we regarded as heads have been printed in bold. In principle, a dependency structure emerges if the node of each head-marked immediate constituent in a constituency tree is moved up and replaces the dominating node while all other relationships are kept intact.

Another option is the direct construction of dependency output on the basis of the parse key. For this purpose we reformulate the grammar of Figure 79.20 resulting in Figure 79.24.

Rules:		Lexicon:	
(R-1)	S → (NP) **VP**	Det	= {*the, their*}
(R-2)	NP → **Noun**	Adj	= {*loving, hating, annoying, visiting*}
(R-3)	NP → (Det) **Noun**	N	= {*relatives, students, professors*}
(R-4)	NP → (Adj) **Noun**	Pron	= {*they*}
(R-5)	NP → **Pron**	Vt	= {*love, hate, annoy, visit*}
(R-6)	VP → **Vt** (NP)	Vtger	= {*love, hate*}
(R-7)	VP → **Vtger** (GP)	Ger	= {*loving, hating, annoying, visiting*}
(R-8)	GP → **Ger** (NP)		

Fig. 79.24: A phrase structure grammar indicating dependencies

This grammar consists of normal production rules with terminals and non-terminals. The arrow denotes constituency. The symbols on the right hand side reflect the concatenation of constituents. Brackets, however, have a special meaning. They denote subordination in the sense of dependency. The phrase structure parser ignores these brackets. When the output is created on the basis of the parse key, the bracketing is observed.

Let us use the parse key [*1,2,*,6,*,4,*,**] generated in Figure 79.21. Figure 79.25 illustrates the replacements according to our reformulated grammar yielding a dependency description of sentence (3).

Replacement:	Parse key:
S	-
(NP) VP	1
(Noun) VP	2
(Noun *students*) VP	*
(Noun *students*) Vt (NP)	6
(Noun *students*) Vt *hate* (NP)	*
(Noun *students*) Vt *hate* ((Adje) Noun)	4
(Noun *students*) Vt *hate* ((Adje *annoying*) Noun)	*
(Noun *students*) Vt *hate* ((Adje *annoying*) Noun *professors*)	*

Fig. 79.25 Generation of a dependency structure

Obviously a dependency structure can be produced by applying a mapping function to the output of phrase structure parsers. Can all that is known about efficiency and complexity of phrase structure parsers be applied to the task of dependency parsing then (Fraser 1989, 297)? The answer is likely to be yes for rule-based dependency grammars. They seem to be equivalent to head-marked phrase structure grammars. Further experiments with this type of grammar and how it fits with the known algorithms are desirable.

4. Parsing with a lexicalized dependency grammar

True dependency analysis is akin to a word-oriented view on syntax rather than to a sentence-oriented one. That is why a lexicalized grammar with complement slots seems more appropriate for specifying dependencies than production rules. This chapter presents a bottom-up chart-based parser for this type of grammars. The example adheres to the notation of Dependency Unification Grammar (DUG; Hellwig 2003), but there are several similar approaches, for example Covington's dependency parser for variable-word-order languages (Covington 1990) and McCords Slot Grammar (McCord 1990).

4.1 Resources

From the perspective of DUG, dependency is a relationship between heads and complements. Heads are usually single words; complements may be structured, consisting of heads and complements in turn. Suppose the following sentence has to be parsed (compare HSK 25.1, Figure 44.1):

(5) *the robot picks up a red block*

Fig. 79.26 illustrates the structure that is imposed by a DUG on the string of words in (5).

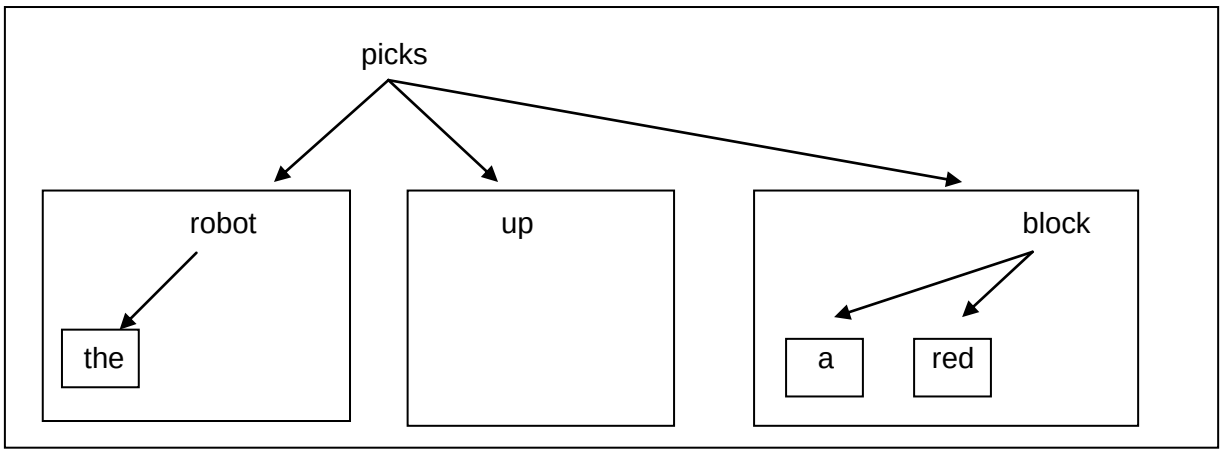

Fig. 79.26: Dependency structure as a relationship between heads and complements

A dependency structure implies a constituency structure. Any head together with its dependents forms a constituent. The constituents of sentence (5) are depicted in Figure 79.26 by square boxes. In the DUG notation, what is surrounded by matching brackets is a constituent.

The parser starts with collecting the available information about the particular words in the input. Figure 79.27 displays the grammatical categories provided in the morpho-syntactic lexicon:

No.	Description:
1	(string[the] lexeme[definite'] category[determiner] number[singular, plural] determined[+])
2	(string[robot] lexeme[robot] category[noun] noun_type[count] number[singular] person[it] vowel[-])
3	(string[picks] lexeme[pick] category[verb] form[finite] tense[present] voice[active] person[he,she,it])
4	(string[up] lexeme[up] category[particle])
5	(string[a] lexeme[indefinite'] category[determiner] number[singular] determined[+] vowel[-])
6	(string[red] lexeme[red] category[adjective] use[attributive, predicative] vowel[-])
7	(string[block] lexeme[block] category[noun] noun_type[count] number[singular] person[it] vowel[-])

Fig. 79.27: Morpho-syntactic categorization of the words in the example

Since each word is looked up in isolation it may be morpho-syntactically ambiguous. That is why there are disjuncts of features in the specifications, e.g. *number[singular, plural]*. Usually these disjuncts dissolve in the course of the parsing process.

In addition to the paradigmatic categories associated with each word in Figure 79.27, the parser needs information about the possible syntagmatic relationships between the words. A way to picture a syntagmatic relationship is the assumption of a head that opens up a slot for a dependent. The words in Figure 79.26 are such heads, the square boxes can be seen as slots, and each arrow represents a relationship between one head and one slot. The task of the parser is simply to insert fillers into slots if they fit. This process has to be repeated recursively until all words and word groups have found a slot and all mandatory slots have found a filler.

If the parser is to follow this strategy, all relevant properties of heads and dependents of any syntagmatic relation must be specified. This can be done in form of so-called templates.

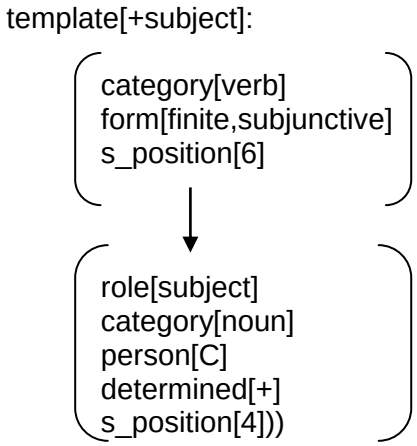

Fig. 79.28: A template for a dependency relation

The template in Figure 79.28 can be paraphrased as follows: "If there is a finite or subjunctive verb in a virtual sixth position in the sentence then there may be a phrase which depends on the verb and is

headed by a noun in the virtual fourth position of the sentence. The noun phrase is functioning as the subject, the phrase must be congruent with the verb with respect to person and it must be determined, for example by inclusion of a determiner." Templates are partial specifications of trees. The set of trees that conform to this partial specification form a syntactic equivalence class (Sikkel 1997, 69)

a) Templates for subject, particle of phrasal verbs and direct objects

```
(template[+subject] category[verb] form[finite,subjunctive] s_type[statement] s_position[6]
        (< slot[regular] role[subject] category[noun] person[C] determined[+] s_position[4]))

(template[+phrasal] category[verb] s_position[6]
        (> slot[regular, select] role[phrasal_part] category[particle] s_position[14,19]))

(template[+dir_object] category[verb] voice[active] s_position[6]
        (> slot[regular] role[dir_object] category[noun] determined[+] s_position[17]))
```

b) Template for determiners of singular count nouns

```
(template[%dete_count_singular] category[noun] noun_type[count] number[singular]
n_position[10]
        (< slot[adjunct] role[determination] category[determiner] number[singular] determined[C]
        n_position[2]))
```

c) Template for the Determiner *a/an*

```
(template[%dete_count_singular_a] category[noun] noun_type[count] number[singular]
n_position[10]
        (< slot[adjunct] role[determination] category[determiner] number[singular] determined[C]
        vowel[C] n_position[2]))
```

d) Template for adjectives

```
(template[%attr_adj] category[noun]  n_position[10]
        (< slot[adjunct, multiple] role[attribute] category[adjective] use[attributive] vowel[C]
        n_position[6]))
```

Fig. 79.29: A sample of templates

Figure 79.29 comprises all the templates needed for parsing sentence (5). The name of each template is included in the head term. A tag at the beginning of the slot term denotes the left or right adjoining of the filler. Various types of slots may be distinguished. The rest of the attributes describe properties of the head and the dependent constituent. Not all of the attributes are context-free. For example, the position of various phrases in the sentence (s_position) and the position of the words in a noun phrase (n_position) can not be handled by a context-free automation, but they can be easily checked by a computer.

The templates must be associated with the lexeme and reading attribute of each word. In this way the combination capabilities of individual words are described. The data structure for this purpose is called "syntactic frame". Figure 79.30 contains the syntactic frames needed for the example.

```
(lexeme[pick] reading[lift]
        (complement[+phrasal] lexeme[up])
        (complement[+subject])
        (complement[+dir_object]))

(lexeme[definit']
        (adjunct[%dete_count_singular, %dete_count_plural, %dete_non_count]))

(lexeme[indefinit']
        (adjunct[%dete_count_singular_a))

(lexeme[red]
        (adjunct[%attr_adj]))
```

Fig. 79.30: Syntactic frames assigned to the lexemes and readings of the words

DUG distinguishes between complements and adjuncts. A complement specified in a syntactic frame indicates that a certain dependent is expected in the environment of the given word. An adjunct template in a frame means that the given word can itself be made dependent of another word as an adjunct.

So far, the parser is equipped with a grammar consisting of a morpho-syntactic lexicon, a set of templates, and a lexicon of syntactic frames. Next, we introduce a well-formed substring table or chart for keeping track of the parser's actions and for storing intermediate results. Figure 79.31 illustrates the state of the chart after the lexicon has been consulted.

No.	Input segment:	L	R	H	F	Template used:
1	the	1	1	-	-	-
2	robot	2	2	-	-	-
3	picks	3	3	-	-	-
4	up	4	4	-	-	-
5	a	5	5	-	-	-
6	red	6	6	-	-	-
7	block	7	7	-	-	-

Fig. 79.31: The chart after the words in the example have been read

Each row in the chart refers to a smaller or larger segment of the input. At the beginning, the chart contains as many rows as there are readings of words in the input. The columns contain information associated with each segment. Let us assume that the numbers in the first column refer to the linguistic descriptions of the segments. At the beginning these are the ones extracted from the morpho-syntactic lexicon and displayed in Figure 79.27. Two rows are provided for storing the left (*L*) and the right (*R*) boundary of the segments in terms of a word count. Two more rows show the original head (*H*) and the original filler (*F*) from which the segment in question has been composed. The template used for joining the slot and the filler is displayed in another row.

4.2. Bottom-up slot-filling

The task of fitting fillers into slots until the final dependency structure emerges can be organized in many ways. We chose a program architecture that has the potential for parallel processing. It consists of four processes that can be executed in parallel and are connected via message queues: the Scanner, the Predictor, the Selector, and the Completer. The algorithms for these processes are roughly the following.

Bottom-up slot-filling chart parser

Scanner: Read the input text until the end of file. Divide the input text into elementary segments according to the strings stored in the morpho-syntactic lexicon (multi-word lexical entries are possible). Save the description for each reading in the lexicon associated with each particular segment and register it in the chart. Send the identification of each new item to the Predictor.

Predictor: As long as the receiving queue is not empty, read an item (and remove it from the queue). Extract the lexeme from the corresponding description and retrieve all syntactic frames available for the lexeme in question. Inspect all the templates that are mentioned in each frame. Compare the constraints in the template with the information collected by the Scanner. If the lexical attributes and the attributes in the template contradict each other then discard the template, otherwise merge the attributes of both sources. If the procedure is successful, it results in a lexically instantiated template: Slots are subordinated to the individual lexeme in the case of complements. A virtual head is superordinated to the individual lexeme in the case of an adjunct. The device may result in a disjunction of descriptions due to competing syntactic frames and competing templates. Store the augmented description, update the registry in the chart and send the identification of each item to the Selector.

Selector: As long as the receiving queue is not empty, read an item (and remove it from the queue). Search for entries in the chart whose segments are promising candidates for mutual combination with the given segment. As long as continuous combination is requested, a candidate segment is promising if the position of its rightmost word (*R*) is equal to the leftmost word (*L*) of the given segment minus one. Send the identifications of the resulting pairs of chart entries to the Completer. Note that the receiving queue of the Selector is fed by both the Predictor and the Completer process. The former introduces lexical segments; the latter submits composed segments for further combination. In this way the slot-filling activity is recursive.

Completer: As long as the receiving queue is not empty read a pair of chart entries (and remove it from the queue). Try to combine the descriptions of both segments according to the following slot-filling device. Inspect the descriptions associated with both chart entries for slots. If a slot is found then check whether the description of the other chart entry meets the specified filler requirements. If there are unfilled obligatory slots in the filler description then discard this candidate right away. Apply the unification method appropriate to each attribute while comparing slot and filler and while checking the agreement of filler and head. If filler fits into a slot and if the agreement constraints (symbol *C* in the templates) are satisfied then form a new description in which the filler description replaces the slot. Remove all slots from the head that are alternatives to the slot just filled. Save the new description and register it in the chart. The value of *L* in the new row is the *L*-value of the left segment, the value of *R* in the new row is the *R*-value of the right segment. Check whether the segment spans the whole input. If this is the case then output the description. Otherwise pass the chart entry of the new segment to the Selector.

No.	Input segment:	L	R	H	F	Template used:
8	the robot	1	2	2	1	% dete_count_singular
9	the robot picks	1	3	3	8	+subject
10	the robot picks up	1	4	9	4	+phrasal
11	red block	6	7	6	7	%attr_adj
12	a red block	5	7	11	5	%dete_count_singular_a
13	the robot picks up a red block	1	7	10	12	+dir_object

Fig. 79.32: The chart at the end of the slot-filling process

The chart that results from processing the example is shown in Figure 79.32. The descriptions connected with the chart entries are displayed in Figure 79.33. The filler portion of each description is underlined. Entry 13 in the chart is the one that indicates a complete coverage of the example sentence. The corresponding structure 13 in Figure 79.34 is the desired result of the parser.

No.	Description:
8	(string[robot] lexeme[robot] category[noun] noun_type[count] number[singular] person[it] vowel[-] determined[+,U] n_position[2,10] (< role[determination] string[the] lexeme[definite'] category[determiner] number[singular] determined[+,C] n_position[2]))
9	(string[picks] lexeme[pick] category[verb] form[finite] tense[present] voice[active] person[he,she,it] s_type[statement] s_position[4,6] (< string[robot] role[subject] lexeme[robot] category[noun] noun_type[count] number[singular] person[it] vowel[-] determined[+,U] n_position[2,10] s_position[4] (< role[determination] string[the] lexeme[definite'] category[determiner] number[singular] determined[+,C] n_position[2])))
10	(string[picks] lexeme[pick] category[verb] form[finite] tense[present] voice[active] person[he,she,it] s_type[statement] s_position[4,6,14] (< string[robot] role[subject] lexeme[robot] category[noun] noun_type[count] number[singular] person[it] vowel[-] determined[+,U] n_position[2,10] s_position[4] (< role[determination] string[the] lexeme[definite'] category[determiner] number[singular] determined[+,C] n_position[2])) (> string[up] role[phrasal_part] lexeme[up] category[particle] s_position[14]))
11	(string[block] lexeme[block] category[noun] noun_type[count] number[singular] person[it] vowel[-] n_position[6,10] (< role[attribute] string[red] lexeme[red] category[adjective] use[attributive] vowel[-,C] n_position[6]))
12	(string[block] lexeme[block] category[noun] noun_type[count] number[singular] person[it] vowel[-] determined[+,U] vowel[-,U] n_position[2,6,10] (> role[determination] string[a] lexeme[indefinite'] category[determiner] number[singular] determined[+,C] vowel[-,C] n_position[2]) (< role[attribute] string[red] lexeme[red] category[adjective] use[attributive] vowel[-,C] n_position[6]))

13	(string[picks] lexeme[pick] category[verb] form[finite] tense[present] voice[active] person[he,she,it] s_type[statement] s_position[4,6,14,17] (< string[robot] role[subject] lexeme[robot] category[noun] noun_type[count] number[singular] person[it] vowel[-] determined[+,U] n_position[2,10] s_position[4] (< role[determination] string[the] lexeme[definite'] category[determiner] number[singular] determined[+,C] n_position[2])) (> string[up] role[phrasal_part] lexeme[up] category[particle] s_position[14]) (> role[dir_object] string[block] lexeme[block] category[noun] noun_type[count] number[singular] person[it] vowel[-] determined[+,U] vowel[-,U] n_position[2,6,10] s_position[17] (< role[determination] string[a] lexeme[indefinite'] category[determiner] number[singular] determined[+,C] vowel[-,C] n_position[2]) (< role[attribute] string[red] lexeme[red] category[adjective] use[attributive] vowel[-,C] n_position[6])))

Fig. 79.33: Dependency descriptions associated with the chart entries

4.3 Enrichments

At first sight a slot-and-filler parser has many advantages. As opposed to most parsers, the number of rules/slots that must be checked is independent of the size of the grammar. It depends solely on the number of templates assigned to the encountered words. The parser is data-driven and expectation-driven at the same time, or more precisely, even its expectations are data-driven. It is easy to draw up lingware in the DUG framework. Complement and adjunct descriptions are modular, so few side effects are to be expected from the addition of more descriptions to the system. How about the coverage?

There are properties of natural languages that cannot be represented by simple dependency trees. Flexible word order, scope, discontinuity, coordination, ellipsis, nucleus, raising, scrambling and other syntactic phenomena require an enrichment of the trees by additional attributes (see Hellwig 2003). The result is an attribute grammar (Knuth 1968, Van Wijngaarden 1969, Pagan 1981) whose generative capacity is gradually extending into the context-sensitive area. The algorithm for each attribute must be built in the parser. The implementation may be similar to "agents" or "word experts". Neuhaus/Bröker (1997) point out that the complexity of parsing non-projective dependency structures is in principle *NP*-complete. However, the implementation of attributes in form of agents opens up the possibility of introducing all kinds of heuristics which are likely to result in polynomial costs.

Word Expert Parsing (Small/Rieger 1982; Small 1987) deserves to be mentioned here once more. Word Experts are kind of a procedural variant of dependency parsing. Parsing is conceived as a distributed process of interacting words. Each word is connected with an expert process. Processes communicate with each other in some organized way. The most advanced system of this sort for German is ParseTalk (Bröker/Hahn/Schacht 1994).

A weakness of lexicalized dependecy parsers is the lack of a sentence prefix, i.e. a completely recognized string from the beginning of the sentence up to the word that is to fill a slot. Fillers often occur in a sentence before a slot is available, which impedes an incremental slot-filling. The design of incremental parsers for dependency grammars is still a challenge (cf. Lombardo 1992; Daum 2004; Nivre 2004).

5. Surface dependency parsing

One must admit that the kind of parsers we described in the previous chapters is not the focus of attention nowadays. Speech recognition and text retrieval are the areas that are most important today in commercial applications. Programs in these environments must be robust and fast. Preferably the analysis is shallow and planar. Speech recognition should proceed word by word without interruption. Text retrieval searches for information in large amounts of running text. Many researchers try to achieve these goals with statistical methods. In this chapter we will check how the dependency approach accommodates to this kind of surface-oriented language processing.

5.1. Finite-state recognizers and transducers

The first choice for a fast linear recognizer of character strings is a finite-state automaton (FSA). The basic idea of such an automaton is the simultaneous advancement within two symbol sequences: the character string in the input and a pattern stored in the computer. The pattern is often represented as a network. The arcs of the network denote input units, the nodes of the network represent states of the parser in the course of inspecting the input. The arcs define under what conditions the parser is allowed to move from one state to the next. (Aho/Sethi/Ullman 1986, 183 f.). The network of states is kind of a flowchart of the program.

The patterns for an FSA can also be represented declaratively in form of so-called regular expressions. A regular language is the type of language generated by a regular grammar (Chomsky's hierarchy Type 3). Regular expressions are a way to formulate such a grammar. There are algorithms to transform regular expressions into the transition table of an FSA. This is an instance of a compiled parser illustrated in Figure 79.12.

Parsing with an FSA is favorable, because any non-deterministic FSA can be transformed into a transition network that can be processed deterministically (Aho/Sethi/Ullman 1986, 117 f., 132 f.). In a non-deterministic environment, costly backtracking devices are needed in order to recover from a dead-end, if a wrong decision has been made. In contrast, at any state of a deterministic procedure the next step is totally clear, given the next symbol in the input. A parser of this type performs recognition in linear time while others need cubic time at best.

(1) NP → **Noun**
NP → Det + **Noun**
NP → Adj + **Noun**
NP → **Pron**

(2) {Det? Adj* N | Pron}

(3)

1	Det	2
1	Adj	2
1	N	3/e
1	Pron	3/e
2	Adj	2
2	N	3/e

Fig. 79.34: A deterministic recognizer for noun phrases

Figure 79.34 (1) displays the four *NP*-rules of the grammar fragment in Figure 79.20. These four rules are re-phrased as a single regular expression in (2). This regular expression is transformed into a deterministic state-transition table in (3). Each row in the table indicates a starting state, a condition, and a target state. The algorithm for a deterministic recognizer is very simple.

Deterministic Recognizer

Transition: Check the transition table for a row with the actual state as the starting state, the category of the word in the actual position as the condition and a target state then make the target state the actual state, increase the current position by one and repeat Transition, until the end of the input is reached. If the end of the input is reached and automaton is in an ending state (denoted by "/e" in Figure 79.34), then accept the input. Reject the input otherwise.

Of course natural languages are not regular. They include embedding, long-distance agreement and ambiguity which cannot be coped with by FSAs. Nevertheless, applications like speech recognition and text retrieval are an incentive to approximate a deterministic behavior of the parsers. At present, many researches are pinning their hopes on finite-state transducers (FST) to reach this goal.

An FST is a finite-state transition network with transitions labeled by pairs of symbols *(u:l)*. In contrast to a finite-state recognizer, an FST produces output while scanning the input. We ignore bi-directional mapping for the moment, so *u* in the transition label denotes characters in the input and *l* denotes characters in the output. *l* includes a symbol for the empty string. The number of characters or strings on both sides in the pair need not to be the same. A finite-state transducer maps between two regular languages. A transduction is valid only, if the string on the input side takes the finite-state transducer to a final state.

In the simplest case, a transducer is used to output a structural description or some other result if the parser is a finite-state recognizer. However, transducers can be arranged in "cascades". The subsequent transducer takes the output of the previous one as input. The output of cascaded transducers may be equivalent to the expansion or reduction of production rules. Hence, cascaded FST are at least as power-full as context-free grammars (Laporte 1996, Abney 1996). This is not exciting, because any classical parser processes each of its intermediate derivations in an FSA fashion.

FSTs are attractive, because they focus on the word stream rather than on the hierarchical structure. Parsing viewed as string transformation is appealing when scanning large amounts of text is the task. A parser that in the first place inserts labels into the input stream fits in well with the current SGML/XML markup of documents. Often the document has been pre-processed by a part-of-speech tagger and the goal is now to annotate larger segments and to classify them by syntactic functions.

The properties of FSTs are well-matched with the present tendency towards lexicalized phrase structure grammars. Most FSTs nowadays use regular expressions that in one way or the other contain explicit words. An example is Roche's transducer (Roche 1997).

(6) *John thinks that Peter kept the book.*

(7) (S (N *John* N) *thinks that* (S (N *Peter* N) *kept* (N *the book* N) S) S)

(7) is the expected output of Roche's transducer for sentence (6). The point of departure is a syntactic dictionary. The entries needed for example (6) are displayed in Figure 79.35.

N *thinks that* S	S
N *kept* N	S
John	N
Peter	N
the book	N

Fig. 79.35: Fragment of a syntactic dictionary

Each entry in the dictionary is turned into a separate transducer, which contains a regular expression for the input, made up from the left column of the dictionary, and a pattern for the output. The corresponding transducers are listed in Figure 79.36.

T_{thinks_that}	[S *a thinks that* b S]	→	(S[N *a* N] <V *thinks* V> *that* [S b S] S)
T_{kept_N}	[S *a kept b* S]	→	(S [N *a* N] <V *kept* V> [N b B] S)
T_{John}	[N *John* N]	→	(N *John* N)
T_{Peter}	[N *John* N]	→	(N *John* N)
$T_{the\ book}$	[N *the book* N]	→	(N *the book* N)

Fig. 79.36: A list of transducers

Parsing simply consists of applying all the transducers on the input string and then checking if the output is different from the input. If it is, then the output is turned into the input and the set of transducers is applied to the string again. We don't go into details here. It is easy to see how example (6) is turned into the tagged sentence (7) with this method.

Lexicalized cascaded transducers are an interesting variant of the G3-type of grammars, which we associated with lexicon-based constituency grammars in section 2.3 above. Regular expressions might have advantages in describing the units of lexicalized constituency grammars, as compared to the hierarchically structured categories of categorial grammar.

A huge amount of material of the type needed for lexicalized FSTs has been collected for French by Maurice Gross and his group at LADL (now domiciled at the University of Marne-la-Valeé). His lexicon-grammar consists of so-called sentence forms drawn from very fine-grained classifications of verbs, including the frozen parts in idioms or support verb constructions (Gross 1984).

At present a lot of efforts are spend to refine the methods of FST-parsing (Laporte 1996; Ellworthy 1999; Oflazer 2003.) The FST technique gives much freedom to heuristics. One principle is doing the easiest tasks first, e.g. marking apparent beginnings and endings of syntactic units first. Parsing proceeds by growing islands of certainty. Containment of ambiguity is possible. A former analysis can be reassigned when more information is available. Pattern precision should be kept high, though. Aït-Mokhtar/Chanod (1997) hold that the ordering of transducers is in itself a genuine linguistic task.

5.2. Link grammar parsing

It is easy to see that the lexicalized transducers mentioned above exploit dependency relations. Why don't we base surface parsing on a dependency grammar right away? Figure 79.37 illustrates a view on dependency that is focused on the stream of words.

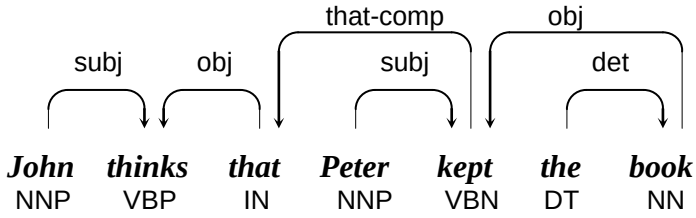

Fig. 79.37: Dependency relations projected onto the word stream

Dependencies in Figure 70.36 are classified by grammatical functions. Part-of-speech tags have been added, as it is usual in many applications. The tags in the example stem from the Penn Treebank Project (http://www.cis.upenn.edu/~treebank/home.html). The arcs in Figure 79.37 lead from the dependent to the dominating word, rather than the other way around as in dependency trees. This is favorable for the following reason. If dependency relations are encoded bottom-up then each word has to be provided with exactly one link. The stream of words and the syntactic tagging can be made completely planar, as it is shown in Figure 79.38.

```
John   </id=1 pos=NNP role=subj head=2>
thinks  </id=2 pos=VBP role=root head=0>
that    </id=3 pos=IN role=obj head=2>
Peter   </id=4 pos=NNP role=subj head=5>
kept    </id=5 pos=VBN role=that-comp head=3>
the     </id=6 pos=DT role=det head=7>
book    </id=7 pos=NN role=obj head=5>
```

Fig. 79.38: Tagging words with dependency information

Dependency links between words in a linear sequence are a characteristic feature of Dick Hudson's Word Grammar. According to Hudson, the advantage of dependency analysis is its total flatness. It is a fact that surface dependency analysis can always be translated into a phrase structure. But exactly this fact is the proof that non-terminal phrase structure categories are superfluous (Hudson 2003, 520).

An advanced implementation of surface dependency parsing is the freely available Link Grammar Parser developed by Daniel Sleator, Davy Temperley and John Lafferty (Sleator/Temperley 1993). Figure 79.39 is the output of their online parser for sentence (8).

(8) *can he see me today*

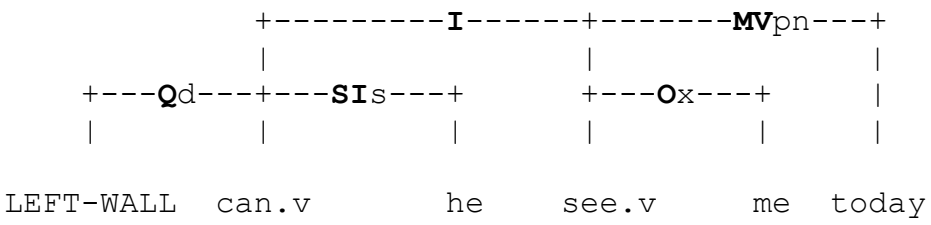

Fig. 79.39: An output of the link grammar parser

The parser creates labeled links connecting pairs of words. The inventory of links reflects a wide variety of syntactic constructions , including many rare and idiomatic ones. In the example, *SI* connects subject nouns to finite verbs in subject-verb inversions; *O* connects verbs to their objects; *I* connects infinite verb forms to modal verbs; *MV* connects verbs to adverbs that follow; *Q* connects the wall to the auxiliary in simple yes-no questions. The "wall" is an artificial construct marking the beginning of the sentence. It is treated like a word. The labels can be varied by lower-case subscripts. Labels without subscripts match with labels with subscripts; labels with subscripts match with each other. *s* stands for singular, *pn* for pronoun, *d* for decision question, *and x* for pronouns as objects.

The capability of a word to form a link with another word is stated in the lexicon by means of connectors. Connectors specify the complementary word on the right or on the left of the word in question. A right connector consists of a link label followed by "+", a left connector consists of the link followed by "-". A left connector on one word connects with a right connector of the same type on another word. The two connectors together form a link. Figure 79.40 displays the fragment of the lexicon entries needed for parsing example (8). Curly brackets denote optionality.

```
can.v: SI+ & {Q-} & I+

he: SIs-

see.v: I- & O+ & {MV+}

me: Ox-

today: MVpn-
```

Fig. 79.40: Example of lexicon entries

The task of the parser is to check the expectations expressed by the connectors in the lexicon and to form a link if the connectors of two words fit in the actual constellation.

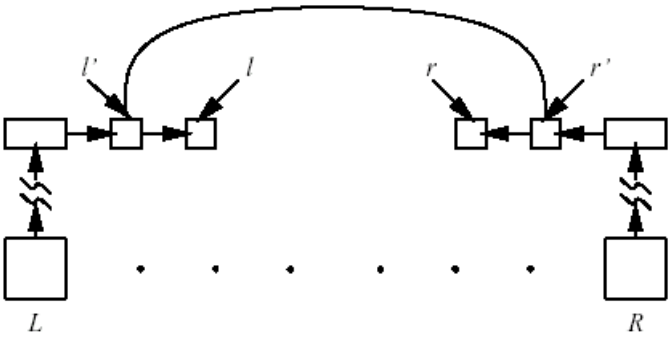

Fig. 79.41: A constellation of the link grammar parser

Figure 79.41 (taken from Sleator/Temperley 1993, 9) shows the situation after a link has been proposed between a connector *l'* on word *L* and a connector *r'* on word *R*. The square boxes above the words *L* and *R* represent the data structure associated with the word. There may be many alternative linking requirements stored with each word in the lexicon due to different usages. One of these alternatives is represented by the rectangular box above. Each alternative consists of a list of left and right connectors represented by the small squares in the diagram. It is obvious that the algorithm for extending the partial solution into the region between *L* and *R* is tricky. The program has to cope with the sequence of words, the disjunctions of connectors and some overall stipulations like the fact that links must not cross each other and that at the end all the words of the sentence must be connected by some link. According to the authors (Sleator/Temperley 1993, 10), the complexity of the parser is $O(N^3)$.

It is noteworthy that Link Grammar does without categories for constituents and even parts-of-speech. (The existing part-of-speech markers *.n, .v, .a* on words have no theoretical relevance.) The only parameter is the complementary role of words in direct syntagmatic relationships. The result is a very restrictive classification, which might even face difficulties with free word order and a rich morphology. Nevertheless, the connectors of Link Grammar don't yet meet completely the idiosyncratic behavior of words. Words can not only be classified according to individual syntactic relationships, but also according to the semantic nature of the complementary words in the relationship. We already mentioned the work of Maurice Gross. In his opinion every word behaves differently. Scrutinizing his valency descriptions of verbs he eventually took the view that complements of words should be described using words as well (Gross 1984, 279). Class symbols were eliminated from his Lexicon-Grammar and replaced by specific nouns. The entries in the lexicon are now of the type displayed in Figure 79.42.

$(\text{person})_0$ eat $(\text{food})_1$

$(\text{person})_0$ give $(\text{object})_1$ to $(\text{person})_2$

$(\text{person})_0$ kick the bucket

Fig. 79.42: Example entries in the lexicon-grammar

Subscripts 0, 1, 2 in Figure 79.42 denote subject, direct object and indirect object respectively. The parser accepts only such lexical material in the indicated slots that is related to the specific nouns in an associated ontology.

5.3. Lexicalized probabilistic parsing

Natural language processing originated in different scientific communities among which were linguistics, computer science and electrical engineering. As a consequence, two paradigms evolved: symbolic and stochastic. The symbolic paradigm took off with Chomsky's generative grammar and encompasses the work of many linguists and computer scientists on grammars and parsing algorithms. Beginning with Terry Winograd's SHRDLU system (Winograd 1972), the symbolic paradigm was strengthened by the natural language understanding field in Artificial Intelligence. Recently the pendulum swung in the direction of stochastic approaches, due to the growing importance of speech recognition and fast text retrieval.

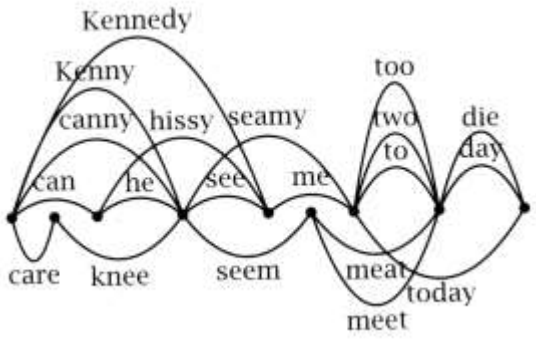

Fig. 79.43: Word candidates of a speech recognizer

Speech processing is faced with a vicious circle. The recognition of words requires context, context requires the recognition of the words it is composed of. As a result there is always a huge number of competing intermediate parses for each spoken input. Figure 79.43 (taken from Manning/Schütze, 408) represents a small subset of the word hypotheses generated if the sentence (8) is entered. In this situation, speech recognition engineers resorted to statistics.

Obviously simple word counts would not do. The identification of a word depends on the identification of neighboring words. What is needed is conditional probability. In general, conditional probability is the probability of an event occurring, given that another event also occurs. It is expressed as *P(A | B)*, which reads as probability *P* of event *A* on condition of event *B*. *P(A | B)* can be calculated by a count of the co-occurrences of *A* and *B* divided by the occurrences of *B* alone.

The literature on probabilistic parsing is characterized by all kinds of proposals for sophisticated conditioning. The simplest model is based on the collocation of words. The probability of a chain of words (e.g. a sentence) is the probability of the words in the chain, each occurring in its correct location. This probability is represented in (9).

(9) $P(w_1, w_2, ..., w_{n-1}, w_n)$

According to Markov's assumption, the probability of the next word in a particular sequence can be approximated by taking only the *n* previous words in the sequence into account. The bigram model approximates the probability of a word by the conditional probability of the preceding word $P(w_n \mid w_{n-1})$, the trigram model computes $P(w_n \mid w_{n-2}, w_{n-1})$ and so forth. The probability of the entire sequence is then calculated as the product of the *n*-gram probabilities of the words in it. (10) is an example of how the probability of sentence (8) in a situation like Figure 79.43 is computed on the basis of bigrams, (11) shows the same computation on the basis of trigrams. <s> denotes the start of a sentence.

(10) *P(can he see me today) = P(can | <s>) P(he | can) P(see | he) P(me | see) P(today | me)*

(11) *P(can he see me today) = P(can | <s> <s>) P(he | can <s>) P(see | can he) P(me | he see) P(today | see me)*

The probabilities of *n*-grams of words must be derived from corpora. As can be imagined, the accuracy of *n*-gram models increases if *n* is increased. Unfortunately, increasing the length of the *n*-gram results in low probabilities, since longer sequences of words re-occur very rarely even in large corpora. We are faced with the sparse data problem of probabilistic language processing. Furthermore, the "simple-minded *n*-gram model" (Lafferty/Sleator/Temperley 1992) belies our linguistic intuition according to which significant probabilities must not to be conditioned on concatenated words but on syntactically related words which may be in varying distances from each other.

One of the attempts to condition probabilities on grammatical relationships is Probabilistic Context-Free Grammar (PCFG).

Rules:			Lexicon:		
(R-1)	S → NP + VP	[0.75]	Det	= {*the, their*}	[1.00]
(R-2)	NP → Noun	[0.15]	Adj	= {*loving, hating, annoying, visiting*}	[0.50]
(R-3)	NP → Det + Noun	[0.40]	Ger	= {*loving, hating, annoying, visiting*}	[0.50]
(R-4)	NP → Adj + Noun	[0.20]	Pron	= {*they*}	[1.00]
(R-5)	NP → Pron	[0.10]	Noun	= {*relatives, students, professors*}	[1.00]
(R-6)	VP → Vt + NP	[0.50]	Noun	= {*love, hate*}	[0.30]
(R-7)	VP → Vtger + GP	[0.05]	Vt	= {*annoy, visit*}	[1.00]
(R-8)	GP → Ger + NP	[0.90]	Vt	= {*love, hate*}	[0.40]
			Vtger	= {*love, hate*}	[0.30]

Fig.79.44: A grammar fragment of English augmented with probabilities

A PCFG is a phrase structure grammar like the one we introduced in Figure 79.20. In addition, a probability is assigned to each rule and each terminal substitution. The probabilities of all rules that expand the same category should add up to 1. The probabilities of all parts-of-speech that can be assigned to the same word should add up to 1 as well. The sum of the invented values in Figure 79.44 is not 1, though, because this is just a fragment of the English grammar.

The probability of a particular parsing result is the product of the probabilities of all rules and lexical substitutions used in it. In section 3.1. we introduced the notion of parse key. The adjective reading of the sentence *students hate annoying professors* received the key [*1,2,*,6,*,4,*,**], the gerund reading is represented by the key [*1,2,*,7,*,8,*,2,**]. The asterisk denotes terminal substitution. On the basis of Figure 79.44, the probabilities of the two readings of the sentence can be computed as follows:

(12) P([*1,2,*,6,*,4,*,**]) = 0.75 * 0.15 * 1.00 * 0.50 * 0.40 * 0.20 * 0.50 * 1.0 = 0.00225

P([*1,2,*,7,*,8,*,2,**]) = 0.75 * 0.15 * 1.00 * 0.05 * 0.30 * 0.90 * 0.30 * 0.15 * 1.00 = 0.0000683

It is questionable whether such calculations are useful, though (Jurafsky/Martin 2000, 456 f.). One problem is the inherent assumption that the expansion of any one non-terminal is independent of any other non terminal. It is known, however, that the probability of expanding an *NP* into a pronoun might differ considerably if the *NP* is the subject or an object in a sentence. Even more questionable is the status of the global frequency of rules in a PCFG. Figure 79.45 (taken from Manning/Schütze 1999, 48) shows the huge differences in probability, if VP-rules in a corpus are conditioned on individual verbs. Given these differences, the global frequency of a rule seems senseless.

	Verb			
Local tree	*come*	*take*	*think*	*want*
VP → V	9.5%	2.6%	4.6%	5.7%
VP → V NP	1.1%	32.1%	0.2%	13.9%
VP → V PP	34.5%	3.1%	7.1%	0.3%
VP → V SBAR	6.6%	0.3%	73.0%	0.2%
VP → V S	2.2%	1.3%	4.8%	70.8%
VP → V NP S	0.1%	5.7%	0.0%	0.3%
VP → V PRT NP	0.3%	5.8%	0.0%	0.0%
VP → V PRT PP	6.1%	1.5%	0.2%	0.0%

Fig. 79.45: The frequency of VP-rules conditioned on individual verbs

Meanwhile it is a widespread opinion that grammars must be lexicalized to allow the construction of useful probabilistic models (Charniak 1997; Collins 1999). The first step to lexicalize a CFG is to distinguish the head constituent from the rest of the constituents in each rule (see chapter 3 above on head-marked PSGs). As a next step, Collins (1999, 79 f.) suggests to annotate each constituent with its head word. A non-terminal constituent receives its head word from its head child. Figure 79.46 shows a phrase structure tree augmented in this way.

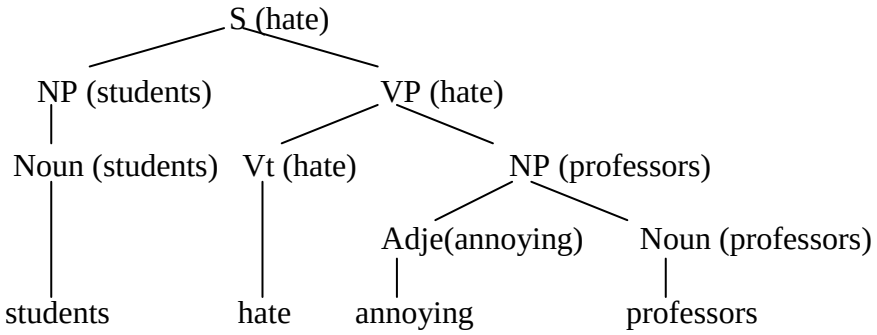

Fig. 79.46: A lexicalized tree following Collins (1999)

A simple way to condition the probability of phrase structure rules on lexical heads is to make a copy of each rule for each possible head word (Schabes 1990) and then calculate the probability of sentences like (8). A similar effect is achieved if lexicalized finite-state transducers of the type shown in Figure 79.36 are provided with probabilities. The overall probability of a parsed sentence can then be calculated on the basis of the probability of the involved transducers.

To summarize, the probabilistic paradigm of language processing ended up with conditioning probabilities on dependencies. This resulted in a growing interest in dependency grammars (Magerman (1995), Eisner 1996; Collins 1996). According to Chaniak (1997), the best conditioning for a word *w* is its head word. The head word propagation in Figure 79.46 suggested by Collins (1999) is exactly the

mechanism we employed in section 3.3. in order to create a dependency output from a phrase structure grammar.

The planar representation of dependencies, illustrated in Figure 79.37 above, lends itself to new Markov models. Rather than conditioning the statistical model on *n*-grams of concatenated words, such a model relies on tuples of co-occurring words linked up by a dependency. A trigram model of this kind, based on Link Grammar, is presented in Lafferty/Sleator/Temperley (1992).

It seems reasonable also to explore the stochastic usefulness of genuine lexicalized dependency grammars, introduced in chapter 4. What they offer is a number of valuable parameters for conditioning probabilities: lexical items as heads associated with slots for mandatory complements, lexical items being suited for filling such slots, lexical items functioning as optional adjuncts of specific heads. All these parameters differ for each semantic reading of the same surface word. This should help to eliminate fraud statistical values that ignore the polysemy of words.

Reliable estimates of the parameters of probabilistic models are always a fundamental issue. In principle, there are two ways to acquire probabilities of syntactic relationships: generative or empirical. The generative approach requires a grammar, drawn up manually. The total probability mass one is assigned to the total set of strings the grammar is able to generate. Of course, probabilities of this kind can only be approximated, because the number of strings in a natural language is infinite. If the rules are lexicalized, it should be possible, though, to approximate the probability of each word co-occurring with other words in a rule according to the grammar. Note that a probability is assigned to any well-formed phrase or sentence, even if it has never been uttered.

In contrast, the empirical approach conceives probability as the frequency of usage. Probabilities are assigned to syntactic constructions according to their occurrence in past utterances. The optimal tool for this kind of estimates would be a parser with full coverage. A representative corpus is analyzed with this parser. Then, the syntactic constructions in the parser output are counted. At present, mainly corpora that have been manually annotated with parse trees are checked in this way. A well-known example is the Pen Tree Bank (Marcus/Santorini/Marcinkiewicz 1993) and the Prague Dependency Tree Bank (Hajič 1998).

Although it is not difficult to write grammar rules and to improve them if necessary, people seem reluctant to do this kind of job. A lot of language processing nowadays can be characterized as linguistically void. Like Baron von Münchhausen using his own boot straps to pull him self out of a swamp, bootstrapping from corpora is the catchword of many researchers today. The goal is not only to retrieve the frequencies of constructions from corpora, but to detect the constructions themselves by statistical means. "Of current interest in corpus-oriented computational linguistics are techniques for bootstrapping broad coverage parsers from text corpora" Abney (1996, 337) . The steps are, according to Abney, "inducing a tagger from word distributions, a low-level 'chunk' parser from tagged corpus, and lexical dependencies from a chunked corpus."

The assumption behind the bootstrapping approach is that any structure is reflected by a deviation from the arbitrary distribution of elements. There are some doubts about the prospects of this method for detecting grammars. First, there is the sparse data problem. A language is a device to create an infinite number of different utterances. Finding even an identical sequence of four words in a corpus is a rare event. The bare frequency of a couple of words in a syntactic relationship, among all the other relationships and correlations in a corpus, is too low to discover the rule. Second, the bootstrapping attempt is inadequate as a model for human language learning. When humans learn a language, there is always another input in addition to the utterances they hear. The child understands the situation she is in to a certain extent, while she interprets the utterances. She compares previous situations when comparing the utterances she hears with previous utterances. The relevant oppositions that lead to the generalization if a syntactic pattern are not available in a normal corpus.

As a compromise, one could try to combine the generative and the empirical approaches. For example, a carefully designed dependency grammar associated with manually encoded subcategorization frames for a large amount of word meanings should provide the slots. The sets of words that fit into the slots are then derived from a corpus. Prototypical examples of fillers, like the ones in the Lexicon Grammar of Gross (see Figure 79.42), may be used as "seeds" that are extended by similar words in the corpus, possibly with the help of a lexical ontology. Schulte im Walde (2003) made experiments on the automatic induction of German semantic verb classes in the same vein. Eventually the subcategorization and selection of words might be refined enough, so that the correct candidate of a speech recognizer is identified with almost a hundred percent certainty. Only then we can speak of an adequate language model.

6. Select Bibliography

Abney, Steven (1996): Partial Parsing via Finite-State Cascades. In: *Journal of Natural Language Engineering*, 2(4), 337-344

Aho, Alfred V./Ullman, Jeffrey D. (1972): *The Theory of Parsing, Translation, and Compiling.* Englewood Cliffs.

Aho, Alfred V./Ullman, Jeffrey D. (1977): *Principles of Compiler Design.* Reading, MA.

Aho, Alfred V./Sethi, Ravi/Ullman, Jeffrey D. (1986): *Compilers. Principles, Techniques, and Tools.* Reading, MA.

Aït -Mokhtar, Salah/Chanod, Jean-Pierre (1997): Incremental finite-state parsing. In: *Proceedings of the 5th Conference on Applied Natural Language Processing* (ANLP-97). Washington, DC, 72-79.

Bar Hillel, Yehoshua/Gaifman, Haim/Shamir, Eli (1964): On Categorial and Phrase Structure Grammars. In: Bar-Hillel, Yehoshua: *Language and Information*, Reading, MA, 99-115.

Barták, Roman (1999), Constraint Programming. In Pursuit of the Holy Grail. In: *Proceedings of the Week of Doctoral Students* (WDS99), Part IV. Prague, 555-564.

Bröker, Norbert/Hahn, Udo/ Schacht, Susanne (1994): Concurrent Lexicalized Dependency Parsing. The ParseTalk Model. In: *Proceedings of the 15th conference on Computational Linguistics*, Volume 1. Kyoto, 379-385.

Charniak, Eugene (1997): Statistical Parsing with a Context-Free Grammar and Word Statistics. In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence*. Menlo Park CA.

Charniak, Eugene (2001): Immediate-Head Parsing for Language Models. In: *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*. Toulouse, 116-123.

Chomsky, Noam (1957): *Syntactic Structures*. The Hague.

Chomsky, Noam (1965): *Aspects of the Theory of Syntax*. Cambridge, MA.

Cohen, Neal J./ Poldrack, Russel A./ Eichenbaum Howard (1997): Memory for items and memory for relations in the procedural/declarative memory framework. *Memory* 5, 131-178.

Collins, Michael J. (1996): A new statistical parser based on bigram lexical dependencies. In: *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*. Santa Cruz CA, 184-191.

Collins, Michael J. (1999): *Head-driven Statistical Models forNatural Language Parsing.* Ph.D. thesis, University of Pennsylvania, Philadelphia.

Covington, Michael (1990): Parsing discontinuous constituents in dependency grammar. In: *Computational Linguistics*, 16 (4), 234-236.

Covington, Michael A. (1990): *A Dependency Parser for Variable-Word-Order Languages*. Technical Report AI-1990-01, University of Georgia, Athens, GA.

Daum, Michael (2004): Dynamic dependency parsing. In *Proceedings of the ACL 2004 Workshop on Incremental Parsing*, Barcelona.

Dietrich, Rainer/Klein, Wolfgang (1974): *Einführung in die Computerlinguistik.* Stuttgart.

Duchier, Denys (2004): Axiomatizing Dependency Parsing Using Set Constraints. In: *Proceedings of the Sixth Meeting on Mathematics of Language*. Orlando, 115-126.

Earley, Jay C. (1970): An efficient context-free parsing algorithm. In: *Communications of the ACM* 13 (2), 94-102.

Eisner, Jason M. (1996): Three New Probabilistic Models for Dependency Parsing. An Exploration. In: *Proceedings of the 16th International Conference on Computational Linguistics* (COLING-96). Copenhagen, 340-345.

Elworthy, David (2000).: A Finite-State Parser with Dependency Structure Output. In: *Proceedings of the 6th International Workshop on Parsing Technology.* Trento.

Fraser, Norman (1989): Parsing and Dependency Grammar. In: *UCL Working Papers in Linguistics* 1, 296-319.

Gaifman, Haim (1965): Dependency Systems and Phrase-Structure Systems. In: *Information and Control* 8, 304-337.

Gross, Maurice (1984): Lexicon-Grammar and the syntactic analysis of French. In: *Proceedings of the 10th International Conference on Computational Linguistics*, Stanford, 275-282.

Hajič, Jan (1998). Building a syntactically annotated corpus: The Prague Dependency Treebank. In: Hajičová, E. (ed.): *Issues of Valency and Meaning. Studies in Honour of Jarmila Panevová.* Prague,106-132.

Hays, David G. (1964): Dependency theory: A formalism and some observations. In: *Language* 40, 511-525.

Hays, David G. (1966): Parsing. In: Hays, David G. (ed.): *Readings in Automatic Language Processing*. New York, 73-82.

Hellwig, Peter (1978): *Formal-desambiguierte Repräsentation. Vorüberlegungen zur maschinellen Bedeutungsanalyse auf der Grundlage der Valenzidee.* Stuttgart.

Hellwig, Peter (1980): PLAIN - A Program System for Dependency Analysis and for Simulating Natural Language Inference. In: Bolc, Leonard (ed.) : *Representation and Processing of Natural Language.* München, Wien, 271-376.

Hellwig, Peter (1989): Parsing natürlicher Sprachen. In: Bátori, I.S./Lenders, W./Putschke, W. (eds.): *Computational Linguistics. An International Handbook on Computer Oriented Language Research and Applications* (= Handbücher zur Sprach- und Kommunikationswissenschaft*).* Berlin, 348-431.

Hellwig, Peter (2003): Dependency Unification Grammar. In: Ágel, Vilmos/Eichinger, Ludwig M./Eroms, Hans-Werner/Hellwig, Peter/Heringer, Hans Jürgen/Lobin, Henning. (eds.): *Valency and Dependency. An International Handbook of Contemporary Research.* Volume 1 (= HSK; 25.1). Berlin, New York, 593-635.

Hudson, Richard (2003): Word Grammar. In: Ágel, Vilmos/Eichinger, Ludwig M./Eroms, Hans-Werner/Hellwig, Peter/Heringer, Hans Jürgen/Lobin, Henning. (eds.): *Valency and Dependency. An International Handbook of Contemporary Research.* Volume 1 (= HSK; 25.1). Berlin, New York, 508-526.

Jackendoff, Ray (1977): *X-Bar Syntax. A Study of Phrase Structure*. Cambridge, MA.

Jurafsky, Daniel S./Martin, James H. (2000): *Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, NJ.

Kahane, Sylvain/Polguère, Alain (eds.) (1998): *Processing of Dependency-Based Grammars*. Proceedings of the Workshop. COLING-ACL'98, Montréal.

Karlsson, Fred/ Voutilainen, Atro/Heikkila, Juha/Anttila, Atro (eds.) (1995): *Constraint Grammar. A Language-independent System for Parsing Unrestricted Text.* Berlin, New York.

Knuth, Donald E. (1968): Semantics of context-free languages. In: *Mathematical systems theory* 2, 127-145.

Koller, Eugene/Niehren, Joachim (2002): Constraint programming in computational linguistics. In: Barker-Plummer, D./Beaver, D./van Benthem, J./Scotto di Luzio, P. (eds.): *Words, Proofs, and Diagrams*. Stanford, 95-122.

Kratzer, Angelika/Pause, Eberhard/Stechow, Arnim von (1974): *Einführung in die Theorie und Anwendung der generativen Syntax.* 2 Bde. Frankfurt/M.

Lafferty, John/Sleator, Daniel/Temperley, Davy (1992): Grammatical trigrams: A probabilistic model of LINK grammar. In: *Proceedings of the AAAI Fall Symposium on Probabilistic Approaches to Natural Language.* Menlo Park, CA, 89- 97.

Laporte, Éric (1996): Context-Free Parsing With Finite-State Transducers. In: *Proceedings of the 3rd South American Workshop on String Processing* (= International Informatics Series 4). Montréal, Ottawa, 171-182.

Lombardo, Vincenco/Lesmo, Leonardo (1996): An Earley-type recognizer for dependency grammar. In: Proceedings of the 16th conference on Computational linguistics, Vol. 2. Copenhagen, 723-728.

Lombardo, Vincenco/Lesmo, Leonardo (1996): Formal aspects and parsing issues of dependency theory. In: Proceedings of the 17th conference on Computational linguistics, Vol. 2. Montréal, 787-793.

Lombardo, Vincenzo (1992):Incremental dependency parsing. In: Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics. Newark, 291-293.

Magerman, David M. (1995): Statistical decision-tree models for parsing. In: *Proceedings of the Association of Computational Linghuistics.* Boston, MA, 276–283.

Manning, Christopher/Schütze, Hinrich (1999): *Foundations of Statistical Natural Language Processing*. Cambridge MA.

Marcus, Mitch/Santorini Beatrice/Marcinkiewicz, Mary Ann (1993): Building a large annotated corpus of English: The Penn Treebank. In: *Computational Linguistics*, 19(2), 313-330.

McCord, Michael (1990). Slot Grammar. A System for Simpler Construction of Practical Natural Language Grammars. In: *Proceedings of the International Symposium on Natural Language and Logic* (= Lecture Notes in Computer Science, Vol. 459). London, 118-145.

Neuhaus, Peter/Bröker, Norbert (1997): The complexity of recognition of linguistically adequate dependency grammars. In: *Proceedings of the eighth conference on European chapter of the Association for Computational Linguistics*. Madrid, 337-343.

Nivre, Joakim (2003): An Efficient Algorithm for Projective Dependency Parsing. In: *Proceedings of the 8th International Workshop on Parsing Technologies*. Nancy, 149–160.

Nivre, Joakim (2004): Incrementality in Deterministic Dependency Parsing. In: *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*. Barcelona, 50-57.

Oflazer, Kemal (2003): Dependency Parsing with an Extended Finite-State Approach. In: *Computational Linguistics*, 29 (4), 515–544.

Pagan, Frank G. (1981): *Formal specification of programming languages.* New York.

Roche, Emanunel (1997): Parsing with Finite-StateTransducers. In: Roche, E./Schabes, Y. (eds.): *Finite-State Language Processing*. Cambridge, MA.

Schabes, Yves (1990): *Mathematical and Computational Aspects of Lexicalized Grammars.* . Ph.D. thesis, University of Pennsylvania, Philadelphia.

Schneider, Gerold (2004): Combining Shallow and Deep Processing for a Robust, Fast, Deep-Linguistic Dependency Parser. In: *European Summer School in Logic, Language and Information (ESSLLI)*, Nancy.

Schröder, Ingo (2002): Natural Language Parsing with Graded Constraints. PhD thesis, Hamburg.

Schulte im Walde, Sabine (2003): *Experiments on the Automatic Induction of German Semantic Verb Classes.* PhD thesis (= Arbeitspapiere des Instituts für maschinelle Sprachverarbeitung Vol. 9 No. 2), Stuttgart.

Shieber, Stuart M. (1986): *An introduction to unification-based approaches to grammar*. Chicago.

Sikkel, Klaas (1997): *Parsing Schemata. A Framework for Specification and Analysis of Parsing Algorithms*. Berlin.

Sleator, Daniel/ Temperley, Davy (1993): Parsing English with a Link Grammar. In: *Proceedings of the Third International Workshop on Parsing Technologies.* Tilburg.

Small, Steven (1987): A distributed word-based approach to parsing. In: Leonard Bolc (ed.): *Natural Language Parsing Systems.* Berlin, 161-201.

Small, Steven/Rieger, Chuck (1982): Parsing and Comprehending with Word Experts. In: Lehnert, W./Ringle, M. (eds.): *Strategies for Natural Language Processing.* Hillsdale, NJ, 89-147.

Tapanainen, Pasi/Järvinen, Timo (1997): A non-projective dependency parser. In *Proceedings of the 5th Conference on Applied Natural Language Processing.* Washington, DC., 64–71,

Tesnière, Lucien (1959): *Eléments de syntaxe structurale.* Paris.

Tomita, Masaru (1996): *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Boston.

Wijngaarden, Aard van (1969): Report on the algorithmic language ALGOL 68. In: *Numerische Mathematik* 14, 79-218.

Winograd, Terry (1972): *Understanding Natural Language.* New York, London.

Woods, William A. (1970): Transition Network Grammars for Natural Language Analysis. In: *Communications of the Association for Computing Machinery* 13, No. 10, 591-606.

Zeevat, Henk/Klein, Evan/Calder, Jo (1987): Unification Categorial Grammar. In: Haddock, N./Klein, E./Morrill, G. (eds.): *Categorial Grammar, Unification Grammar, and Parsing* (= Edinburgh Working Papers in Cognitive Science, Vol. 1). Edinburgh, 195-222.

Zwicky, Arnold M. (1985): Heads. In: *Journal of Linguistics* 21, 1-30.

Peter Hellwig, Heidelberg