Tutorial at COLING/ACL-98 Montreal 1998. Revised and extended version, 18 Sep 2004.

NATURAL LANGUAGE PARSERS

A "Course in Cooking"

Peter Hellwig, Heidelberg

0 Introduction

There are numerous technical reports about particular parsers and each conference on computational linguistics adds a few more to the list. It is not our aim to survey this huge variety of existing implementations. What we want to achieve is a basic understanding of the essentials and to provide guidance for evaluating existing parsers as well as for putting together new ones. For that purpose we have to take apart the known algorithms, identify the issues that arise in any parser development, and compare the alternative solutions that exist for each identified task. Metaphorically speaking, this paper should be read like a cookbook, and the reader is encouraged to recombine the various ingredients in order to "cook" a new parser.

Chapter one surveys the most important issues and examines the choices one has, when constructing a parser. Chapter two describes prototypical combinations of the basic principles resulting in special types of parsers. Oftentimes the easiest way to understanding a theory is by observing how it functions in practice. Therefore, chapter two is in the form of concrete examples. This kind of presentation has extended the length of this paper. The busy reader may skip the details and concentrate on the conclusions. The student, however, is invited to try out each algorithm by solving the exercises in the appendix. Chapter three introduces criteria for evaluating parsers and applies these criteria to the prototypes.

1 Parsing Issues

1.1 What is Parsing?

The definition of parsing depends on the discipline that is involved. The following answers to the question are common

(i) in computer science:

Parsing is the assignment of a structural description to a character string.

(ii) in linguistics:

Parsing is the assignment of a syntactic description to a sentence.

(iii) in knowledge-based systems:

Parsing is the assignment of a knowledge representation to an utterance.

1.2 Prerequisites of a parser

A language consists of character strings structured in a specific way. The well-formed character strings and their structures are defined by a grammar. A parser can therefore be defined more specifically as a computer program that assigns a structural description to a given string relative to a given grammar.

This definition implies the following prerequisites of a parser:

(i) A grammar formalism

As a first step, a notation for drawing up grammars must be created.

(ii) A grammar

Next, a grammar in the chosen formalism must be written for each language which the parser should be able to handle.

(iii) An algorithm

Finally, an algorithm must be developed which determines, whether a given input is covered by the grammar, and if so, which description applies to it.

In the framework of Artificial Intelligence, these three tasks are known as knowledge representation, knowledge, and knowledge processing. Different requirements characterize each of these tasks, e.g. (i) expressiveness for the grammar formalism, (ii) adequacy for the grammar, (iii) efficiency for the algorithm.

1.3 Connection between grammar and parser

There are three ways of connecting the description of grammatical data and the parsing procedure:

(i) Interpreting parser

Grammar and parsing procedure are separate. The grammar is data which is retrieved and interpreted by the parsing routine. The grammar is "declarative", i.e. it is formulated independently from the aspect of analysis. The algorithm is based exclusively on the syntax of the grammar formalism, not on the contents of the individual grammar. (Aho/Sethi/Ullman 1986:3f)

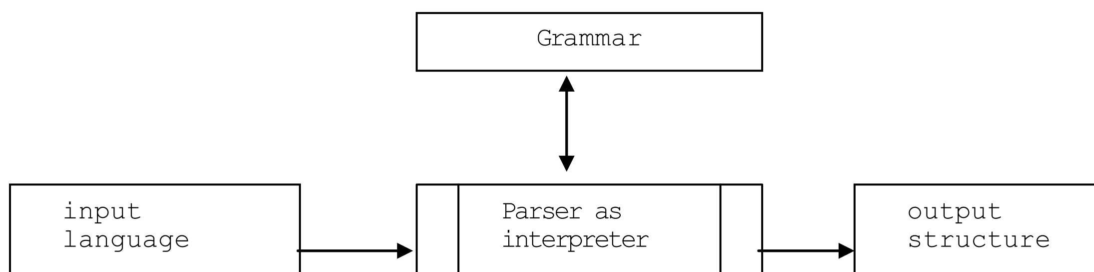


Fig. 1: Interpreting parser

(ii) Procedural parser

Grammatical data and parsing procedure are not separate. The grammar is integrated in the algorithm. The grammar is "procedural", i.e. 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.

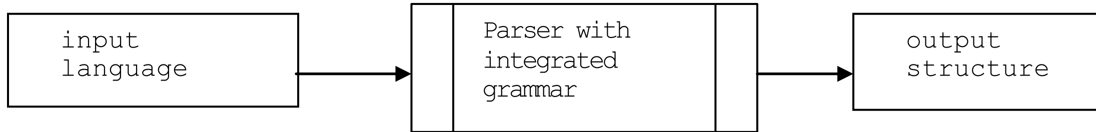


Fig. 2: Procedural parser

(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: 257f.). In the final parser, the grammar is part of the algorithm.

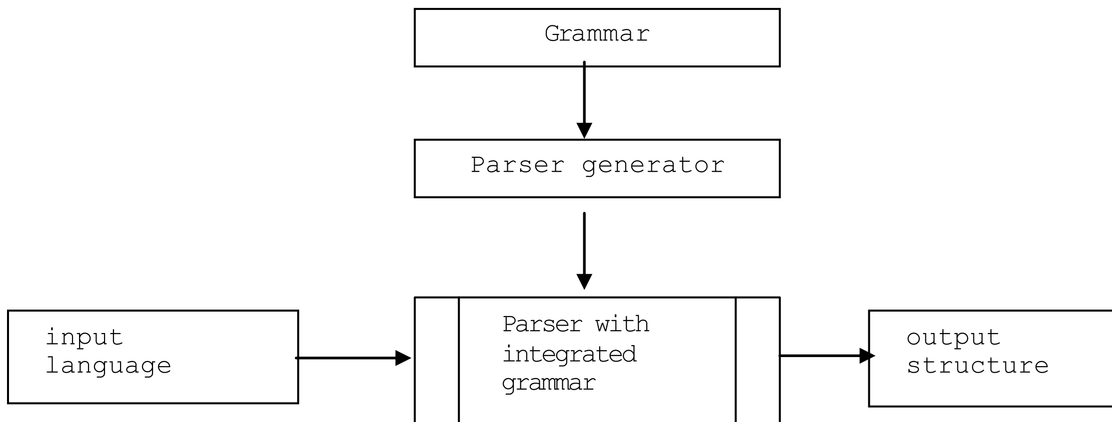


Fig. 3: Compiled parser

1.4 The type of the structural description

It is common in linguistics to represent syntactic structure by tree diagrams. Two different types of structural descriptions are widely used (*Hudson* 1980).

(i) Constituency structure

The units that correspond to the nodes in a constituent tree are smaller or larger segments of the input strings. The edges in a phrase structure tree represent the composition of larger phrases from elementary ones. Each level in a phrase structure tree denotes another 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.

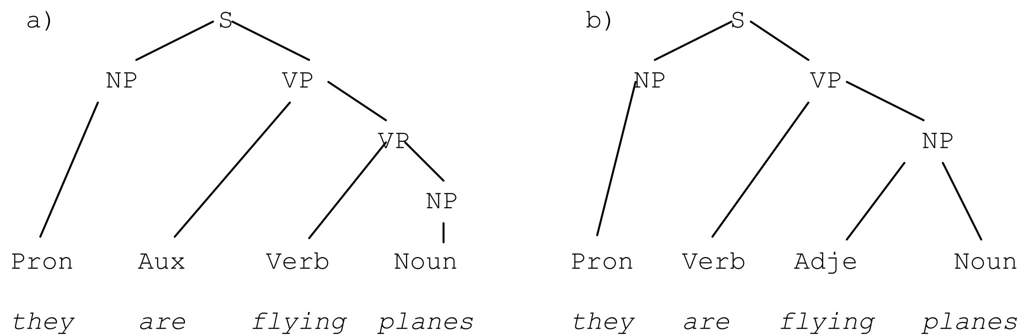


Fig. 4: Constituency trees

(ii) Dependency structure

The units that correspond to the nodes in a dependency tree are all elementary segments of the input string. The arcs denote the syntagmatic relationship between an elementary segment and its complements (*Tesnière* 1959). All nodes belong to the same unique and complete segmentation. Composed segments are present only as a combination of elementary segments which are associated with the nodes of a partial or complete dependency tree.

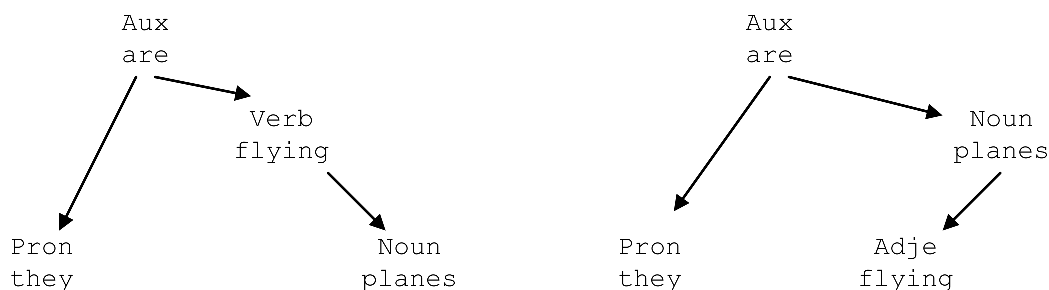


Fig. 5: Dependency trees

Although the nodes in the tree are labeled with words, dependency is not a relationship between individual words but between an individual word and a phrase dominated by the word (*Hellwig* 1993). For example, a dependency relationships hold between a verb and its subject and between the verb and its object. Obviously subjects and objects do not consist of single words only. They are complex structures which are represented again by dependency trees. Technically, dependency is a relationship between nodes and subtrees (i.e. a node together with all the nodes it dominates). Dependency grammar was often misinterpreted in this respect and therefore seldom used in computer systems.

Both types of structuring ask for complementary information. If constituency is the basic relationship then the dependencies must be symbolized by additional labels (e.g. a label for which constituent is the head of a phrase). If dependency is the basic relationship, an implicit constituency can be derived from the tree in that any single node as well as any subtree corresponds to a constituent. The linear precedence, which is depicted in the case of constituency by the tree itself, must be explicitly stated in the decoration of the nodes in the dependency tree.

1.5 Complex categories and unification

A basic element as well as a phrase can be classified according to a host of features, e.g. verb or noun, singular or plural, masculine or feminine, nominative or accusative etc. It is advantageous to describe a syntactic unit by a category that is composed of a set of subcategories so that various cross classifications are possible. In addition, one should distinguish between feature types and concrete features or, in a common terminology, between attributes and values. It is then possible to express generalizations, e.g. the agreement of two items in number or gender, by means of attributes only. Attributes are, in a sense, variables that can, in the concrete case, be instantiated by various values. The mechanism of instantiation and calculating agreement is often referred to as unification. The essence of unification lies in the fact that, 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 a variable by a value can go in both directions and the directions may differ for different attributes in the same category. Agreement may be propagated in this way across a long distance. The whole grammar adopts the character of an equation rather than that of a generative device. (*Kratzer et al.* 1974 , *Hellwig* 1980, *Shieber* 1986)

1.6 Grammar specification formats and basic recognition strategies

(i) Production rules

The grammar consists of a set of so-called production rules by which all of the well-formed character strings of a language are generated. The goal of the parser is to reconstruct the derivation of the input character string from a given category (usually S or "sentence") on the basis of these rules. If this goal can be achieved, the input is accepted and the corresponding structure is assigned to it (*Aho/Ullmann* 1972).

This strategy adheres to the sentence-oriented approach of generative grammar which was introduced into linguistics by Noam Chomsky (*Chomsky* 1957, 1965).

(ii) Transition networks

The fundamental idea is the simultaneous advancement within two symbol sequences: the character string of the input and a pattern. The grammatical data is often represented as a network. The arcs of the network denote linguistic units, the nodes of the network represent states of the parser in the course of inspecting the input. The arcs are labeled by symbols which define under what conditions the parser is allowed to move from one state to the next. (*Aho/Sethi/Ullman* 1986: 183ff)

In the case of finite and recursive transition networks (FTN, RTN), the symbols in the patterns are corresponding directly to the symbols in the input, i.e. these networks are equivalent to declarative grammars. Augmented transition networks (ATN), augmented with conditions and procedures, are patterns for processes which eventually lead to the recognition of the input string, i.e. they are in fact flow charts of the program's actions. (*Bobrow/Fraser* 1969, *Woods* 1970)

(iii) Complement slots

This approach is based on the assumption that the syntactic structure of a language originates from the inherent combinatory capacity of the basic elements. The lexicon describes the complements with which a lexical item combines. The parser checks whether any syntactic units occur in the environment of the element that fulfill the specifications. If this is the case a new unit is formed covering both the original element and the complements. The complements can be thought of as fillers which fit into slots which are opened up by lexical items.

This strategy corresponds to the word-oriented treatment of syntax in traditional linguistics. The lexical specification of complements also corresponds to strict subcategorization in generative grammar. (*Hellwig* 1974, 1983, 1994, *Hudson* 1984, *McCord* 1980, *Starosta/Nomura* 1986)

1.7 Constructing a parse tree

The goal of the analysis is a hierarchical structure which is usually represented as a labeled tree. The root of the tree is often known in advance. (In the case of the derivation-oriented analysis the category of the root is identical with the starting symbol of the generation.) The exterior nodes of the tree are also known. They are identical with the elementary units of the input string and are identified and classified by means of the lexicon. What is unknown is the interior space of the tree. In the following illustrations the area of the unknown is dotted. We have the following point of departure:

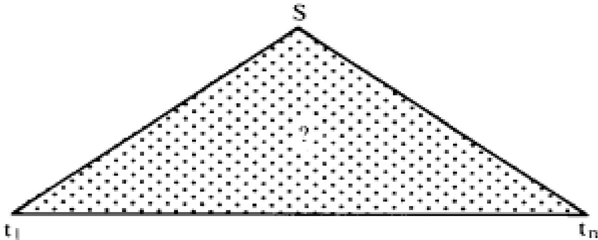


Fig. 6: The tree to be filled in

The interior of the tree has to be filled, step by step, 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 tree is the starting point. Proceeding from top to bottom, one tries to add more nodes according to the grammar, until the sequence of terminal nodes is reached. For example, after a rule S -> NP VP has been applied, we have the following situation:

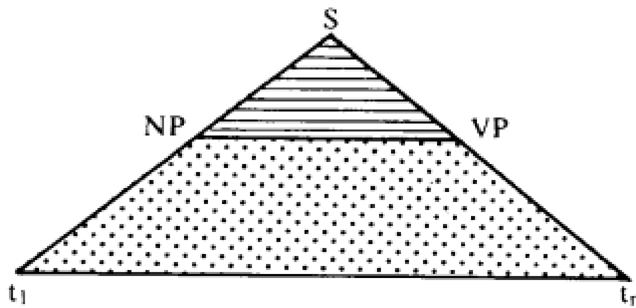


Fig. 7: Top-down analysis

The top-down strategy makes use of derivation 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 -> Det N has been applied, the following situation exists:

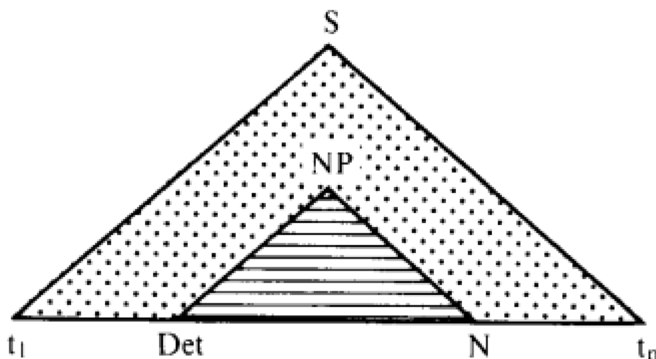


Fig. 8: Bottom-up analysis

The bottom-up strategy makes use of derivation rules in the direction from right to left. Connecting the symbols in the tree corresponding with the right hand side of a rule with the symbol on the left side of the rule 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 of the symbols created 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 are processed at the earliest possible stage and will 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.

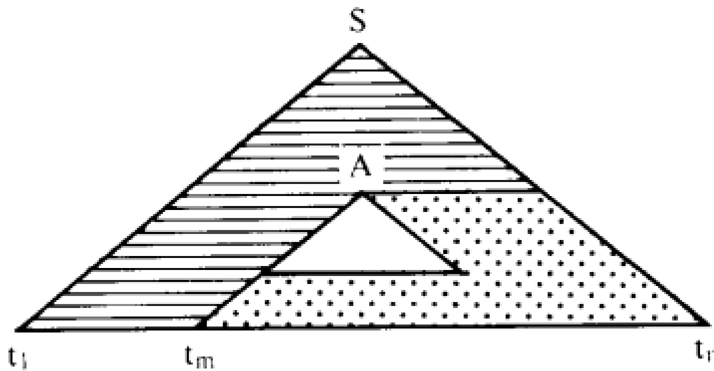


Fig. 9: Depth first

(iv) Breadth first

Here symbols are processed in the sequence of their creation. 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 following tree:

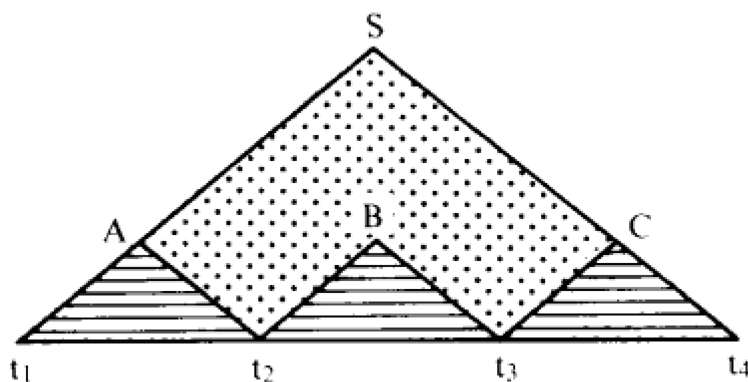


Fig. 10: Breadth first

Both top-down analysis and bottom-up analysis have advantages and disadvantages. Ideally both principles should be combined. The different strategies apply to dependency trees as well, although linguistically, they mean something different here, as compared to phrase structure trees. In the dependency framework, top-down analysis proceeds 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.

1.8 Processing the input

How the parser processes the input is another issue to be decided. The following possibilities exist, which can partly be combined:

(i) The input is processed from left to right (i.e. from position 1 to n).

(ii) The input is processed from right to left (i.e. from position n to 1).

(iii) There is just one pass through the input during the whole analysis.

In this case the transition from one position to the next is the primary principle of control. Sentence boundaries may be crossed readily and entire texts can be parsed on-line.

(iv) There are several passes through the input, because other control principles guide the process.

(v) In addition to one path processing from left to right, it may be stipulated that the next word will be accepted only if the complete context on the left of it has already been accepted. A parser with this property is called left-associative.

(vi) Processing the input is not continuous at all but starts from several points in the string and proceeds from there to the left and to the right (Island-parsing), or it is tried to identify the edges of a constituent first and then fill the inner space. Cascaded application of parsers (e.g. FSA-transducers) is one of the techniques.

1.9 Handling of alternatives

Alternatives occur while filling the syntax tree because there is not (yet) enough information about the 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 alternatives 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 state at which the alternative arose and the path of the next alternative is tracked. This procedure is continued until all alternatives have been checked.

(ii) Parallel-processing

All alternatives are pursued simultaneously. This means that 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, i.e. the control constantly switches between paths.

(iii) Reduction of alternatives by looking ahead

The choice of a rule of the grammar is made dependent on the next k categories in the input beyond the segment to be covered by the rule. If there is only one possibility left, after taking into account the next k symbols, 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.

1.10 Control of results

The parser must keep track of the intermediate results obtained at a given moment and the work still to be done until the final result is reached. There are two possibilities:

(i) Goal oriented recognition

The parser works on the final result right from the beginning. Intermediate hypotheses about the syntactic structure of the input are introduced, rejected or refined until the final analysis is achieved. One way to organize this task is via the consumption technique: Potential categories that have to be verified are put in a workspace; those categories that were recognized are then discarded from the workspace. An input is accepted when there are no more hypotheses to be tested and the workspace is empty. Of course, somewhere the information must be saved that is necessary for yielding a syntactic description as the final result.

(ii) Storage of all intermediate results in a chart

The parser operates with a working area in which all intermediate results are stored separately and remain accessible at all times. Every intermediate result can be re-used in any new combination. This data structure is called a "well-formed substring table" or a "chart". At the end of the execution the table should contain, among other things, the complete result. This organization guarantees that no work is done more than once.

1.11 Well-formed substring table (Chart)

The well-formed substring table associates structural descriptions with input segments. Each entry in the table contains an indication of an input segment, e.g. by storing the positions of the segment's start and its end, and the information so far created by the parser for this particular segment, e.g. its grammatical category or its complete parse tree.

A well-formed substring table is often visualized as a chart with vertices and arcs. The vertices mark the borderlines between the segments. The arcs bridge the span between the beginning and the end of each segment. The vertices are numbered, the arcs are labeled with the categories of the constituents. The smallest arcs bridge the segments associated with the lexical elements. Longer arcs bridge non-terminal constituents. If an arc bridges the entire input string a complete result has been found. (*Varile* 1983). The final chart for the ambiguous sentence "They are flying planes" looks like this (arcs have been drawn above and below the vertices for the sake of perspicuity, the arcs involved in the adjective reading are bold:

S
VP
NP
VP
NP
Pron
Aux
Verb
Noun
THEY ARE FLYING PLANES
Verb
Adje
NP
VP
S

Fig. 11: A chart

The non-ambiguous constituents NP:they and NP:planes are built into both sentences although they were created just once. The arcs could also be labeled by the complete parse trees that adhere to the spanned segments. This is to be recommended if the grammatical features of the partial segments change (by means of the unification technique) when they are included in a larger construction.

1.12 Overall Control

There are many possibilities for organizing the overall control of the parser. The steps of the parser depends on specific mix of the chosen basic alternatives (*Winograd* 1983: 363ff.). It is very common to use the continuous progression in the input as the primary organizing principle. More exotic parsers use special data structures (e.g. a so-called agenda, *Kay* 1977) in order to arrive at a more flexible behavior of the parser. Object-oriented architectures (e.g. so-called word-experts, *Small/Rieger* 1982) as well as parallel processing open up a wide field for innovations. Creating such new designs is left to the students of this tutorial.

1.13 Checklist

Here is a list of the parsing issues mentioned in this chapter. In chapter three we will use this list in order to characterize each prototype.

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

(2) The type of the structural description
- constituency descriptions
- dependency descriptions
- complex categories

(3) Grammar specification format
- production rules
- transition networks
- complement slots

(4) Recognition strategy
- category expansion (top-down)
- category reduction (bottom-up)
- state transition
- slot filling

(5) Processing the input
- from left to right or from right to left
- one-pass (depth-first)
- several passes (breadth-first)
- left-associative
- non-continuously (island parsing, edges-first, cascaded)

(6) Handling of alternatives
- backtracking
- parallel processing
- looking ahead
- well-formed substring table

(7) Control of results
- goal oriented recognition of final result(s)
- all intermediate results stored (chart)

2 Prototypical parsers

The aim of this chapter is purely illustrative. On the one hand, there are many other typical parsers (for example, *Hellwig 1989* contains 16 prototypes). On the other hand, each prototype can be implemented in various programming styles. The algorithms presented below are certainly not to be implemented in the same fashion as they are described. The main criterion of the presentation has been perspicuity for the human reader rather than proximity to the machine. What I wanted to achieve was a non-technical presentation of technical issues. In fact, I have experimented with various representations of the algorithms, for example with pseudocode and Nassi-Shneiderman diagrams (cf. the recursive algorithm for PT-1). In my feeling, given the purpose of this tutorial, it is too time-consuming for the reader to work through such technical specifications which are certainly more adequate from the view of a modern style of programming. Eventually I structured the presentation of algorithms in form of blocks with labels and go-tos. The contents of each block are mandatory, their control by go-tos is not. I recommend as general exercise for the student who is going to become a programmer to reformulate each algorithm so that all gotos are eliminated.

PT-1. Top-down parser with backtracking

Illustrates:

(1) Connection between grammar and parser: Interpreting parser; grammar separate from procedure.
(2) Linguistic structure assigned: Constituency trees.
(3) Grammar specification format: Context-free production rules.
(4) Recognition strategy: Category expansion (top-down); expectation driven.
(5) Processing the input: Left-to-right, one pass (depth-first); left-associative.
(6) Treatment of alternatives: Schematic backtracking.
(7) Control of results: Goal oriented recognition; consumption technique.

References: *Aho/Ullman* 1972-73:289ff, *Hellwig* 1989: 378ff, *Winograd* 1983:94ff

Prerequisites:

(P-1) An ordered set of **production rules** of a context-free phrase structure grammar without left recursion. A key to identify each rule. A variable indentifying the current rule. At the beginning the current rule is the first rule.

(P-2) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-3) An input table which contains the **words of the input together with their categories** and their position in the sequence. The categories have been assigned to the words according to the lexicon in a preceding phase.

(P-4) A variable P for the **current position** in the course of moving forward from the beginning to the end of the input. At the beginning P identifies the first word.

(P-5) A **working space** consisting of a number of entries. The last entry is the current one, identified by the variable A. The working space functions as a stack. A preceding entry can become the current one again (by backtracking), all subsequent entries will then be overwritten. Each entry contains a derivation, i.e. a sequence of symbols generated according to the grammar rules. The current entry contains the latest derivation. The symbol farthest left in the current entry is the symbol to be processed next, called the "due symbol". At the beginning, the working space consists of one entry only, containing the initial symbol of the grammar (e.g. "S").

(P-6) A **backtracking store**, i.e. a stack which is augmented whenever an alternative arises. Each entry contains all the information necessary to reconstruct the state of affairs that existed before one of several applicable rules has been chosen. This includes: the current position in the input, the current entry in the working space, the next rule which is an alternative to the rule chosen. Only the last entry is accessible in a stack. In order to access former entries, later entries must be removed from the stack. At the beginning the backtracking store is empty. We use the variable B in order to identify the states of the backtracking store.

Algorithm:

(A-1) **Expansion (of non-terminal categories).** If the current entry in the working space is empty then go to (A-4) for backtracking. Loop through the rules starting with the current rule and incrementing the current rule variable. If the due symbol in the current derivation is the same as the symbol on the left side of the current rule then create a new derivation by replacing the due symbol in the current derivation by the sequence of symbols on the right side of the rule. If the due symbol is situated on the left side of a further rule, make a new entry in the backtracking store containing the current position, the current row in the working table and the key of the next applicable rule. Then add a new entry to the working space and store the new derivation in it. The first symbol of the new derivation is now the new due symbol. If a replacement has taken place in this block then go to (A-1) in order to try an expansion of the new derivation. If no expansion was possible then go to (A-2) in order to check whether the due symbol is a terminal category.

(A-2) **Recognition (of terminal categories).** If the due symbol in the current entry agrees with the category at the current position in the input table then add a new entry to the working space, discard the recognized symbol from the derivation and store the rest of the symbols as the new derivation. As a consequence, the next symbol in the sequence becomes the due symbol. Increase the current position by 1 and go to (A-3). If the due symbol does not match with the category in the input table then go to (A-4) for backtracking.

(A-3) **Final condition.** If the current position does not exceed the number of elements in the input table then go to (A-1) in order to repeat the steps expansion and recognition. If the current position exceeds the number of elements in the input table by one (i.e. the input is exhausted) then there should be no symbols in the current derivation, i.e. all expectations should be met by the input. If symbols are left in the derivation then go to (A4) for backtracking. Otherwise accept the input and create an output. If all readings of a possibly ambiguous input should be detected then go to (A-4) to check the alternatives, else finish the procedure.

(A-4) **Backtracking.** If the backtracking store is empty then finish the procedure. Reject the input if the final condition in (A-3) was never met. If the backtracking store is not empty then reconstruct the situation in the working space according to the top-most entry in the backtracking store: Set the current position, the current entry in the working space and the current rule to the values specified in the backtracking store. Then remove the entry from the backtracking stack.

As a consequence, all entries in the working space being created after the last alternative are lost. The one-but-latest alternative if any is now accessible on the backtracking store. Go to (A-1).

Example

GRAMMAR G1

Rules	Lexicon
(R-1) S -> NP VP (R-2) VP -> vi (R-3) VP -> vt NP (R-4) VP -> vt NP PP (R-5) NP -> n (R-6) NP -> det n (R-7) NP -> det adj n (R-8) PP -> prep NP	vi = *{sleep, fish}* vt = *{study, visit, see, enjoy}* det = *{the, no, my, many}* adj = *{foreign, beautiful}* n = *{tourists, pyramids, friends, fish, cans, Egypt, we, they}* prep = *{in, by, with}*

INPUT TABLE

Input:	*they*	*visit*	*friends*	*in*	*Egypt*
Lexicon:	**n**	**vt**	**n**	**prep**	**n**
Position:	1	2	3	4	5

WORKING SPACE

A	Derivations	Explanation	P
1	**S**	state at start	1
2	**NP VP**	expansion R-1	
3	**n VP**	expansion R-5	
4	**VP**	recognized n	2
5	**vi**	expansion R-2	

A	Derivations	Explanation	P
4	**VP**	B=2 back to A=4	2
5	**vt NP**	expansion R-3	
6	**NP**	recognized vt	3
7	**n**	expansionR-5	
8	**-**	recognized n	4

A	Derivations	Explanation	P
6	**NP**	B=3 back to A=6	3
7	**det n**	expansion R-6	

A	Derivations	Explanation	P
6	**NP**	B=3 back to A=6	3
7	**det adj n**	expansion R-7	

BACKTRACKING STORE

B	Back to
0	
1	P=1 **A=2** R-6
2	P=2 **A=4** R-3

B	Back to
1	cf. above
2	P=2 **A=4** R-4
3	P=3 **A=**6 R-6

B	Back to
2	cf. above
3	P=3 **A=6** R-7

B	Back to
2	cf. above

4	**VP**	B=2 back to A=4	2
5	**vt NP PP**	expansion R-4	
6	**NP PP**	recognized vt	3
7	**n PP**	expansion R-5	
8	**PP**	recognized n	4
9	**prep NP**	expansion R-8	
10	**NP**	recognized prep	5
11	**n**	expansion R-5	
12	-	recognized n	6

1	cf. above
2	P=3 **A=6** R-6
3	P=5 **A=10** R-6

One parse found. Construct parse tree with R-1, R-5, R-4, R-5, R-8, R-5.

10	**NP**	B=3 back to A=10	5
11	**det n**	expansion R-6	

2	cf. above
3	P=5 **A=10** R-7

10	**NP**	B=3 back to A=10	5
11	**det adj n**	expansion R-7	

2	cf. above

6	**NP PP**	B=2 back to A=6	3
7	**det n PP**	expansion R-6	

1	cf. above
2	P=1 **A=6** R-7

6	**NP PP**	B=2 back to A=6	3
7	**det adj n PP**	expansion R-7	

1	cf. above

2	**NP VP**	B=1 back to A=2	1
3	**det n VP**	expansion R-6	

0	
1	P=0 **A=2** R-7

2	**NP PP**	B=1 back to A=2	1
3	**det adj n PP**	expansion R-7	

0	

No more parses found.

Legend: The diagram shows the states of the working space and the backtracking store at subsequent instances in time. Each row illustrates one state. A = current entry in the working space, P = current position in the input table, B = top-most entry in the backtracking store, R-i = key of a rule.

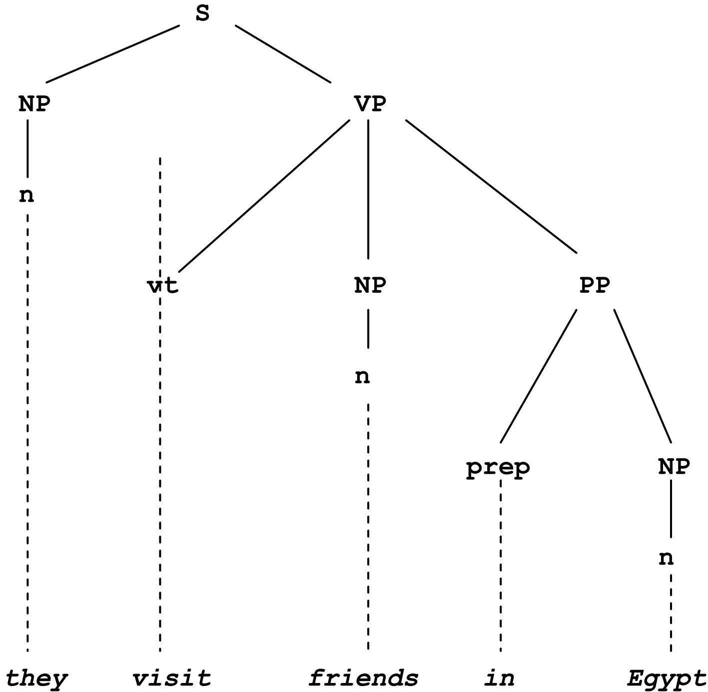
Generating a parse tree

The described algorithm consumes the terminal categories of the input table from left to right until the workspace is empty. It is a pure recognition algorithm. If the parser should output a structural description of the input sentence then the required information must be stored somehow. The following possibilities exist: (i) Each recognized symbol is not really discarded but is just temporarily made "invisible". A tree is created in parallel with the expansion steps of the parser

(and revised with each backtracking instruction). (ii) A key is stored consisting of all successfully applied rules; a parse tree is generated at the end by applying these rules. The latter procedure yields the following output for the above example:

```
Parse key: R-1, R-5, R-4, R-5, R-8, R-5
```

Parse tree: **Rule:**



R-1
R-5
R-4
R-5
R-8
R-5

Implementation of PT-1 by recursive procedure calls

The following algorithm uses recursive procedure calls for expansion and backtracking instead of a working space and a backtracking store. The stacks that have been maintained manually in PT-1 are now taken care of by the operating system which keeps track of the procedure calls and returns.

Prerequisites:

(P-1) An ordered set of **production rules** of a context-free phrase structure grammar without left recursion.

(P-2) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-3) The following **functions**:

parser()	- the main procedure
read()	- reads one sentence from the input
look_up_lexicon()	- associates words with part-of-speech categories
expansion()	- replaces non-terminal categories recursively according to the production rules and returns the variable success

(P-4) The following **variables**:

Preterminals	- contains the string of word categories
Constituents	- contains the categories resulting from the expansion of the start symbol
Position	- stores the actual word position
Parse	- contains a concatenation of numbers identifying the rules applied in the expansion
SaveConstituents	- saves the value of Constituents for backtracking
SavePosition	- saves the value of Position for backtracking
SaveParse	- saves the value of Parse for backtracking
Success	- yields 1 in case of success and 0 in case of failure

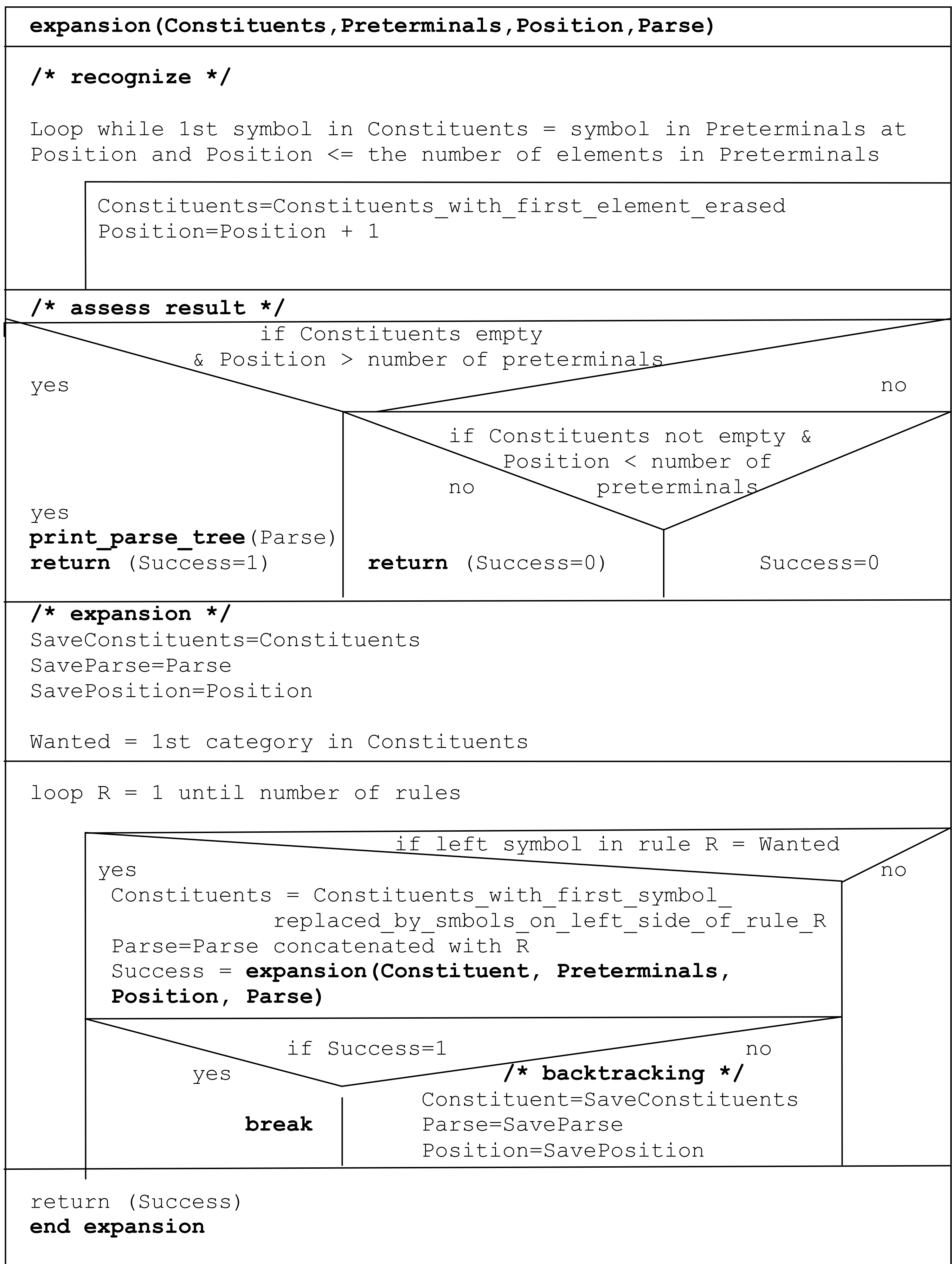
Algorithm

1. Structogram of the parsing procedure:

```
parser()
-----------------------------------------------------------------
Sentence = read() /* a sentence is read from the input */
-----------------------------------------------------------------
Preterminals = look_up_lexicon (Sentence)
/* this function outputs a sequence of preterminal categories
which are associated with the words in the sentence according to
the lexicon */
-----------------------------------------------------------------
Constituents="S"   /* initial configuration */
Position=1
Parse=0


Success = expansion(Constituents, Preterminals, Position, Parse)
-----------------------------------------------------------------
end parser
```

2. Structogram of the expansion procedure:



Here is a trace of the recursive PT-1 parser. Compare the flow with the original version.

```
parser
Sentence="they visit friends"
Preterminals="n,vt,n"
Constituents="S"
Position=1
Parse=0
    Success=expansion("S", "n,vt,n", 1, 0)
    /*expansion*/
    SaveConstituents="S"
    SaveParse=0
    SavePosition=1
    Wanted="S"
    Loop R=1
    Constituents="NP,VP"
    Parse=1
        Success=expansion("NP,VP", "n,vt,n", 1, 1)
        /*expansion*/
        SaveConstituents="NP,VP"
        SaveParse=1
        SavePosition=1
        Wanted="NP"
        Loop R=1,2,3,4,5
        Constituents="n,VP"
        Parse=1+5
            Success=expansion("n,VP", "n,vt,n", 1, 1+5)
            /*recognition*/
            Constituents="VP"
            Position=2
            /*expansion*/
            SaveConstituents="VP"
            SaveParse=1+5
            SavePosition=2
            Wanted="VP"
            Loop R=1,2
            Constituents="vi"
            Parse=1+5-2+
                Success=expansion("vi", "n,vt,n", 2, 1+5+2)
                /*expansion*/
                SaveConstituents="vi"
                SaveParse=1+5+2
                SavePosition=2
                Wanted="vi"
                Loop R=1,2,3,4,5,6,7,8
                return Success=0
                end expansion
            /*backtracking*/
            Constituent="VP"
            Parse=1+5
            Position=2
            Loop R=3
            Constituents="vt, NP"
            Parse=1+5+3
```

```
Success=expansion("vt,NP","n,vt,n",2,1+5+3)
/*recognition*/
Constituents="NP"
Position=3
/*expansion*/
SaveConstituents="NP"
SaveParse=1+5+3
SavePosition=3
Wanted="NP"
Loop R=1,2,3,4,5
Constituents="n"
Parse=1+5+3+5
    Success=expansion("n","n,vt,n",3,1+5+3+
    5)
    Constituents=""
    Position=4
    print_parse_tree(1+5+3+5)
    return Success=1
    end expansion
break
return Success=1
end expansion
        break
        return Success=1
        end expansion
    break
    return Success=1
    end expansion
break
return Success=1
end expansion

end parser
```

Evaluation

(1) **Efficiency.** Blind generation of expansions. Schematic backtracking, many times. The same work is done repeatedly because all intermediate results are lost that were created beyond the point of backtracking. Nevertheless, the first complete result is usually reached much earlier than the time needed to explore all possible alternatives. This is due to the left-to-right depth-first expansion of categories. The parser arrives at terminal categories that can be verified relatively soon and only such terminal categories are generated that fit into the left context. (Besides, the algorithm of PT-1 is, in principle, the same as the proof mechanism of PROLOG. Parsers that rely on the build-in mechanisms of PROLOG (*Pereira/Warren 1983)* share the degree of efficiency with PT-1.)

(2) **Coverage.** Context-free grammar, however left-recursive rules are not admitted. A lot of phenomena of natural language are not covered by the example prototype. For example, the consumption technique is not suitable for calculating agreement.

(3) **Drawing up lingware.** A well-known grammar type; side effects will result from the change of rules.

PT-2. Top-down parser with parallel-processing

Illustrates:

(1) Connection between grammar and parser: Interpreting parser; grammar separate from procedure.
(2) Linguistic structure assigned: Constituency trees.
(3) Grammar specification format: Context-free production rules.
(4) Recognition strategy: Category expansion (top-down); expectation driven.
(5) Processing the input: Left-to-right, one pass (depth-first); left associative.
(6) Treatment of alternatives: Parallel processing (the only difference to P-1).
(7) Control of results: Goal oriented recognition; consumption technique.

References: *Hellwig* 1989: 381ff, *Kuno/Oettinger* 1963*, Winograd* 1983:103ff

Prerequisites:

(P-1) An ordered set of **production rules** of a context-free phrase structure grammar without left recursion.

(P-2) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-3) An input table which contains **the words of the input together with their categories** and their position in the sequence. The categories have been assigned to the words according to the lexicon in a preceding phase.

(P-4) A variable P for the **current position** in the course of moving forward from the beginning to the end of the input. At the beginning P identifies the first word.

(P-5) A **working space** consisting of a varying number of entries. As opposed to PT-1, all entries are accessible at the same time. They contain the set of all derivations that can be generated according to the grammar up to the current position in the input. The symbols farthest to the left in each row are the due symbols. At the beginning, the working space consists of one entry only, containing the initial symbol of the grammar (e.g. "S").

Algorithm:

(A-1) **Expansion (of non-terminal symbols).** Replace any due symbol in any entry of the working space according to any production rule that is applicable. Store each expanded derivation in a new entry. Repeat the same process for the derivations in all new entries until no due symbol can be further expanded. Go to (A-2).

(A-2) **Recognition (of terminal symbols).** Remove from the working space all entries whose due symbol does not agree with the category at the current position in the input table. If no entry is left, then reject the input and quit the procedure. In the remaining entries remove the due symbol from the derivation. As a consequence, the next symbol in each derivation becomes the due symbol if there is any. Increase the current position by 1, and go to (A-3).

(A-3) **Final condition.** If the current position does not exceed the number of words in the input table then go to (A-1). If the current position exceeds the number of elements in the input table by one (i.e. the input is exhausted) and at least one entry with no symbol exists in the working

space (i.e. in which all symbols are recognized) then accept the input and quit the procedure. Else reject the input and quit the procedure.

Generating a parse tree. If the number of all the applied rules is stored with each derivation one eventually arrives at a key that can serve for creating a parse tree in the same way as described for PT-1 (compare the last column in the example below).

Example

GRAMMAR G1 (see above)
INPUT: *they visit friends in Egypt*

WORKING SPACE

Position	Input	Lexicon	Derivation	Explanation, keys of applied rules
1	*they*	**n**	S NP VP **n VP** det n VP det adj n VP	start 1 1,5 1,6 1,7
			VP	1,5 recognized n
2	*visit*	**vt**	vi **vt NP** **vt NP PP**	1,5,2 1,5,3 1,5,4
			NP NP PP	1,5,3 1,5,4 recognized vt
3	*friends*	**n**	**n** det n det adj n **n PP** det n PP det adj n PP	1,5,3,5 1,5,3,6 1,5,3,7 1,5,4,5 1,5,4,6 1,5,4,7
			- PP	1,5,3,5 1,5,4,5 recognized n
4	*in*	**prep**	**prep NP**	1,5,4,5,8
			NP	1,5,4,5,8 recognized prep
5	*Egypt*	**n**	**n** det n det adj n	1,5,4,5,8,5 1,5,4,5,8,6 1,5,4,5,8,7
			-	**1,5,4,5,8,5** recognized n, save parse key

Legend: Each block in the diagram illustrates one cycle of expansions and a subsequent recognition. The shaded portions contain the results of the recognition procedure; they form the initial state for the next cycle of expansions.

Evaluation

(1) **Efficiency.** This parser always creates the maximum number of expansions. As a consequence, the same amount of effort is needed for finding the first result as for finding all results. On the other hand, no work is done twice as with PT-1.
(2) **Coverage and lingware.** The same as PT-1.

PT-3. Top-down predictive analyzer (with Greibach normal form grammar)

Illustrates:

(1) Connection between grammar and parser: Interpreting parser; grammar separate from procedure.
(2) Linguistic structure assigned: Constituency trees.
(3) Grammar specification format: Context-free production rules in Greibach normal form.
(4) Recognition strategy: Combination of category expansion (top-down), expectation driven, and category reduction (bottom-up), data driven.
(5) Processing the input: Left-to-right, one pass (depth-first); left associative.
(6) Treatment of alternatives: Looking ahead by means of a special form of the grammar.
(7) Control of results: Goal oriented recognition; consumption technique.

References: *Hellwig* 1989: 382ff, *Kuno/Oettinger* 1963, *Greibach* 1964, *Kuno* 1965, *Dietrich/Klein* 1974: 81ff, *Kuno* 1976

Prerequisites:

(P-1) An ordered set of **production rules** of a context-free phrase structure grammar in Greibach normal form. The first category on the right side of each rule is a lexical symbol (parts-of-speech, grammatical features). The other categories on the right side of rules must be non-terminal. This kind of a grammar is created by recursively substituting the first category on the right side of the rules of the original grammar top-down depth-first until a lexical symbol is reached and by introducing new non-terminal categories for the rest of the rule, if necessary. The idea is to dislocate the process of top-down expansion from the parser into the grammar. At run-time the selection of rules is narrowed down to those rules that start with the lexical category of the actual word in the input. The following format of rules emphasizes the predictive character of this grammar:

Rules must have the form

```
(A,a)  |  v            or
(A,a)  |  -
```

where "A" is a single non-terminal category, "a" is a single lexical category and "v" is a sequence of one or more non-terminal categories. "A" is the left-side symbol of a common production rule, "a" is the first category on the right side of a common rule (the "left handle") and "v" covers the rest of the common rule.

(P-2) through (P-6) are the same as with PT-1.

Algorithm

(A-1) **Prediction:** Form a pair of the due symbol in the working space and the category at the current position in the input table. If there is a rule with for this pair then create a derivation by replacing the due symbol on a new line in the working space by the symbols on the right side of the rule. If there is another rule for the same pair then make a new entry in the backtracking store containing the current position, the current row in the working table and the key of the next applicable rule. Increase position by one, go to (A2). If there is no rule matching the pair but there

is an alternative lexical category then form a new pair with this category and go to (A-1). If there is no matching rule and no lexical alternative then go to (A-3)

(A-2) **Final condition.** If the current position does not exceed the number of elements in the input table then go to (A-1). If the current position exceeds the number of elements in the input table by one (i.e. the input is exhausted) then there should be no symbols in the current derivation, i.e. all expectations should be met by the input. If symbols are left in the derivation then go to (A3) for backtracking. Otherwise accept the input and create an output. If all readings of a possibly ambiguous input should be detected then go to (A-3) to check the alternatives, else finish the procedure.

(A-3) **Backtracking.** If the backtracking store is empty then finish the procedure. Reject the input if the final condition in (A-2) was never met. If the backtracking store is not empty then reconstruct the situation in the working space according to the top-most entry in the backtracking store: Set the current position, the current entry in the working space and the current rule to the values specified in the backtracking store. Then remove the entry from the backtracking stack. As a consequence, all entries in the working space being created after the last alternative are lost. The one-but-latest alternative if any is now accessible on the backtracking store. Go to (A-1).

Example

`GRAMMATIK G1-GNF` (Greibach normal form, weakly equivalent with G1):

Rules

```
(R-1)     (S,n)     | VP
(R-2)     (S,det)   | N  VP
(R-3)     (S,det)   | AN VP

(R-4)     (VP,vi)   | -
(R-5)     (VP,vt)   | NP
(R-6)     (VP,vt)   | NP PP

(R-7)     (NP,n)    | -
(R-8)     (NP,det)  | N
(R-9)     (NP,det)  | AN

(R-10)    (N,n)     | -

(R-11)    (PP,prep)| NP

(R-12)    (AN,adj)  | N
```

Lexicon

```
vi   =  {sleep, fish}
vt   =  {study, visit, see, enjoy}
det  =  {the, no, my, many}
adj  =  {foreign, beautiful}
n    =  {tourists, pyramids,
        friends, fish, cans,
        Egypt, we, they}
prep =  {in, by, with}
```

`INPUT TABLE`

Input:	*they*	*visit*	*friends*	*in*	*Egypt*
Lexicon:	**n**	**vt**	**n**	**prep**	**n**
Position:	1	2	3	4	5

WORKING SPACE **BACKTRACKING STORE**

A	Prediction	Input	P
1	**S**	n	1
2	**VP**	vt	2
3	**NP**	n	3
4	**-**	prep	4

B	Back to
0	
1	P=2 **A=2** R-6

A	Prediction	Input	P
2	**VP**	vt	2
3	**NP PP**	n	3
4	**- PP**	prep	4
5	**NP**	n	5
6	**-**	-	6

B	Back to
0	

Evaluation

(1) **Efficiency.** The predictive analyzer is expectation-driven and data-driven at the same time. That is why it is relatively efficient. Instead of 25 symbol substitutions and 10 times backtracking that are performed by PT-1 with the same input, the predictive analyzer has explored all alternatives with 8 substitutions and backtracks just once. (Even this one time could be avoided by changing the grammar a little bit.)

(2) **Coverage.** The same as PT-1.

(3) **Lingware.** The biggest drawback. The structural description of a Greibach form grammar differs completely from the original grammar and contains unnatural constituents.

Note: The predictive analyzer can also be implemented on the basis of a parallel processing of alternative predictions similar to PT-2.

PT-4. Top-down parser with divided productions

(Earley's algorithm, "Active Chart Parser")

Illustrates:

(1) Connection between grammar and parser: Interpreting parser; grammar separate from procedure.
(2) Linguistic structure assigned: Constituency trees.
(3) Grammar specification format: Context-free production rules.
(4) Recognition strategy: Top-down strategy with bottom-up phases; expectation and data-driven.
(5) Processing the input: Left-to-right, one pass (depth-first); left associative.
(6) Treatment of alternatives: No backtracking; use of a well-formed substring table.
(7) Control of results: All intermediate results accessible.

References: *Earley* 1970, *Aho/Ullman* 1972: 320ff., *Winograd* 1983: 105ff, *Hellwig* 1989: 396, *Covington* 1994

Prerequisites:

(P-1) An ordered set of **production rules** of a context-free phrase structure grammar without restrictions.

(P-2) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-3) An input table which contains **the words of the input together with their categories** and their margins in the sequence. The categories have been assigned to the words according to the lexicon in a preceding phase.

(P-4) A **working table** (well-formed substring table, chart) consisting of an incremental number of rows. For search purposes the table is arranged in sections. All rows with the same right margin r form a section i, where r = i. Each row contains a description of one segment of the input. The segment is identified by its left and right margin. (Margins can be represented in various ways, for example as a simple count of boundaries between words.) The description consists of a so-called divided production, i.e. a production rule furnished with a dot. The dot divides the immediate constituents of the rule into a left portion that is verified by the actual segment and a right portion that is still unmatched. A dot can also occur before or after all constituents of a rule. In the first case, the rule is not yet processed, in the second case the rule is completely verified. The symbol following the dot in a divided production is the due symbol. At the beginning, the working table contains the divided production `'# -> .S'` in the first row, where `'S'` is the initial category of the grammar and `'#'` is a additional non-terminal symbol. This means that a sentence is expected but nothing of it has been seen yet. The left and the right margin for this production equal 0.

(P-5) Two variables, CUR and NEW, which identify rows in the table. CUR refers to **the production to be processed**, NEW refers to **the resulting production** which is added to the table at the end. At the beginning both are set to 1. Subsequently NEW advances ahead of CUR, CUR is incremented continuously running through former values of NEW. Eventually CUR catches up NEW and the procedure ends.

Algorithm:

(A-1) **Predictor.** If a non-terminal constituent is due in the production of row CUR then do the following, else go to (A-2). Extract from the grammar all productions that match with the due symbol. Insert a dot in front of the right-hand side of each production. Store the productions in new rows of the table (increase NEW accordingly). Set both the left and the right margin in row NEW to the value of the right margin of CUR. If an identical production with the same left and right margins already exists in the table then undo the step. Go to (A-4).

(A-2) **Scanner.** If a lexical constituent is due in the production of row CUR then do the following, else go to (A-3). If the word in the input table at the position of the right margin of CUR plus 1 agrees with the category of the due constituent, then copy the production in row CUR into a new row (increase NEW accordingly), move the dot in the production beyond the recognized constituent, adopt the left margin for row NEW from row CUR and set the right margin in NEW to the value of CUR plus 1. Go to (A-4).

(A-3) **Completer.** If the production in the row CUR is complete, i.e. the dot has been moved beyond all constituents, then do the following. Check all previous productions whose right margin is identical to the left margin of the complete production as to whether they contain a due constituent with the same category as the category of the complete production. Copy such productions into a new row of the table (increase NEW accordingly), move the dot beyond the recognized constituent, adopt as the left margin in NEW the left margin of the copied production, set the right margin in NEW to the right margin of the complete production.

(A-4) **Final condition.** If CUR equals NEW then the work table is exhausted. Go to (A-5). Else increase CUR by 1 and go to (A-1).

(A-5) **Check for a complete result.** If there is at least one production `'#' -> S.'` in the table with the left margin equal to 0 and the right margin equal to the number of words in the input table then accept the input and finish the procedure. Else reject the input and finish the procedure.

Example:

GRAMMAR G2 (ambiguous, includes G1)

Rules		
(R-1)	S -> NP VP	
(R-2)	VP -> vi	
(R-3)	VP -> vt NP	
(R-4)	**VP -> VP PP**	
(R-5)	NP -> n	
(R-6)	NP -> det n	
(R-7)	NP -> det adj n	
(R-8)	**NP -> NP PP**	
(R-9)	PP -> prep NP	

Lexicon		
vi	=	*{sleep, fish}*
vt	=	*{study,visit, see, enjoy}*
det	=	*{the, no, my, many}*
adj	=	*{foreign, beautiful}*
n	=	*{tourists, pyramids, friends, fish, cans, Egypt, we, they}*
prep	=	*{in, by, with}*

INPUT TABLE

Input:	*they*	*study*	*fish*	*in*	*cans*
Lexicon:	**n**	**vt**	**vi/n**	**prep**	**n**
Margins:	0 1	1 2	2 3	3 4	4 5

WORKING TABLE

Section 0:

	Divided productions	Left margin	Right margin	Explanation
(1)	# -> .S	0	0	state at start
(2)	S -> .NP VP	0	0	predictor for (1) by R-1
(3)	NP -> .n	0	0	predictor for (2) by R-5
(4)	NP -> .det n	0	0	predictor for (2) by R-6
(5)	NP -> .det adj n	0	0	predictor for (2) by R-7
(6)	NP -> .NP PP	0	0	predictor for (2) by R-8
(*)	NP -> .n	0	0	predictor for (6) by R-5
(*)	NP -> .det n	0	0	predictor for (6) by R-6
(*)	NP -> .det adj n	0	0	predictor for (6) by R-7
(*)	NP -> .NP PP	0	0	predictor for (6) by R-8

Section 1:

(7)	**NP -> n.**	0	1	scanner for(3), *they*
(8)	S -> NP. VP	0	1	completer for (7) in (2)
(9)	NP -> NP. PP	0	1	completer for (7) in (6)
(10)	VP -> .vi	1	1	predictor for (8) by R-2
(11)	VP -> .vt NP	1	1	predictor for (8) by R-3
(12)	VP -> .VP PP	1	1	predictor for (8) by R-4
(13)	PP -> .prep NP	1	1	predictor for (9) by R-9
(*)	VP -> .vi	1	1	predictor for (12) by R-2
(*)	VP -> .vt NP	1	1	predictor for (12) by R-3
(*)	VP -> .VP PP	1	1	predictor for (12) by R-4

Section 2:

(14)	VP -> vt. NP	1	2	scanned for (11), *study*
(15)	NP -> .n	2	2	predictor for (14) by R-5
(16)	NP -> .det n	2	2	predictor for (14) by R-6
(17)	NP -> .det adj n	2	2	predictor for (14) by R-7
(18)	NP -> .NP PP	2	2	predictor for (14) by R-8
(*)	NP -> .n	2	2	predictor for (18) by R-5
(*)	NP -> .det n	2	2	predictor for (18) by R-6
(*)	NP -> .det adj n	2	2	predictor for (18) by R-7
(*)	NP -> .NP PP	2	2	predictor for (18) by R-8

Section 3:

(19)	**NP -> n.**	2	3	scanner for (15), *fish*
(20)	**VP -> vt NP.**	1	3	completer for (19) in (14)
(21)	NP -> NP. PP	2	3	completer for (19) in (18)
(22)	**S -> NP VP.**	0	3	completer for (20) in (8)
(23)	VP -> VP.PP	1	3	completer for (20) in (12)
(24)	PP -> .prep NP	3	3	predictor for (21) by R-9
(25)	**# -> S.**	0	3	completer for (22) in (1)

Section 4:

(26)	PP -> prep. NP	3	4	scanner for (24), *in*
(27)	NP -> .n	4	4	predictor for (26) by R-5
(28)	NP -> .det n	4	4	predictor for (26) by R-6
(29)	NP -> .det adj n	4	4	predictor for (26) by R-7
(30)	NP -> .NP PP	4	4	predictor for (26) by R-8
(*)	NP -> .n	4	4	predictor for (30) by R-5
(*)	NP -> .det n	4	4	predictor for (30) by R-6

(*)	NP -> .det adj n	4	4	predictor for (30) by R-7
(*)	NP -> .NP PP	4	4	predictor for (30) by R-8

Section 5:

(31)	**NP -> n.**	4	5	scanner vor (27), *cans*
(32)	**PP -> prep NP.**	3	5	completer for (31) in (26)
(33)	NP -> NP. PP	4	5	completer for (31) in (30)
(34)	**NP -> NP PP.**	2	5	completer for (32) in (21)
(35)	**VP -> VP PP.**	1	5	completer for (32) in (23)
(36)	PP -> .prep NP	5	5	predictor for (33) by R-9
(37)	**VP -> vt NP.**	1	5	completer for (34) in (14)
(38)	NP -> NP. PP	2	5	completer for (34) in (18)
(39)	**S -> NP VP.**	0	5	completer for (35) in (8)
(40)	**S -> NP VP.**	0	5	completer for (37) in (8)
(*)	PP -> .prep NP	5	5	predictor for (38) by R-9
(41)	**# -> S.**	0	5	completer for (39) in (1)
(42)	**# -> S.**	0	5	completer for (40) in (1)

Legend: The shaded productions are complete. The entries with index (*) in the table are discarded because an identical production is already present. The sections, one for each of the right margins, correspond to the set of states in the original implementation of Earley. Earley's Algorithm is the same as Winograd's "Active Chart Parser", just the terminology diverges.

Evaluation

(1) **Efficiency.** A lot of spurious productions are generated. However no work is done twice since all intermediate results are saved and may be reused. The problem of backtracking does not arise in this approach. The parser combines expectation-driven phases (the predictor) with data-driven phases (when the completer checks the usability of a substring in all environments). As a left-to-right depth-first parser, PT-4 does not generate intermediate results that are obsolete with respect to the left context.

(2) **Coverage.** Unrestricted context-free rules. Incapable to handle phenomena that go beyond context-free grammars.

(3) **Drawing up lingware.** The same as PT-1.

PT-5. Bottom-up parser with a well-formed substring table

(Algorithm according to Cocke, Kasami and Younger)

Illustrates:

(1) Connection between grammar and parser: Interpreting parser; grammar separate from procedure.
(2) Linguistic structure assigned: Constituency trees, dependency trees.
(3) Grammar specification format: Context-free production rules.
(4) Recognition strategy: Category reduction (bottom-up); data driven, no expectations.
(5) Processing the input: Left-to-right, one pass; but not left associative.
(6) Treatment of alternatives: no backtracking; well-formed substring table.
(7) Control of results: All intermediate results accessible.

References: *Kasami* 1965, *Hays* 1966, *Younger* 1967, *Hellwig* 1989: 400ff.

Prerequisites:

(P-1) The set of rules of **a context-free phrase structure grammar in Chomsky normal form**, i.e. all productions contain exactly two immediate constituents. The right one of these constituents is called the right "handle" of the rule.

(P-2) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-3) An input table which contains **the words of the input together with their categories** and their margins in the sequence. The categories have been assigned to the words according to the lexicon in a preceding phase.

(P-4) A variable P for **the current position** in the course of moving forward from the beginning to the end of the input. At the beginning P identifies the first word.

(P-5) A **working table** (a well-formed substring table, a chart) consisting of an incremental number of rows. For search purposes the table is arranged in sections. All rows with the same right margin r form a section i, where r = i. Each row contains the category of one segment of the input. The segment is identified by its left and right margin. (Margins can be represented in various ways, for example as a simple count of boundaries between words.) If the segment is the result of a category reduction then the row contains two pointers, one to the left immediate constituent and one to the right immediate constituent in the working table out of which the segment was constructed.

(P-6) Three variables, CUR, NEW and NEIGHBOR, which identify rows in the table. CUR refers to **the entry in the working table currently processed**, NEW refers to the row of **the resulting segment** which is added to the table at the end. NEIGHBOR is needed for identifying a segment that is **a left neighbor** of the CUR segment. The category in the row CUR is the due symbol. At the beginning CUR is 1 and NEW and NEIGHBOR are 0. NEW is incremented when new rows must be stored. It advances ahead of CUR. CUR is incremented in order to run through former values of NEW to find out if the categories of CUR and NEIGHBOR can be reduced. Eventually no new categories are produced, CUR catches up with NEW and the procedure ends.

Algorithm:

(A-1) **Shift** (a terminal category onto the working table)**.** Look up the word at the current position P of the input table. Store its category in a new row of the working table (increase NEW). Set the right margin in the row to P and the left margin to P minus 1. Set left immediate constituent and right immediate constituent to zero. If there are several categories in the input table due to lexical ambiguity then create a separate entry in the working table for each of them.

(A-2) **Reduce.** The symbol in the row CUR is chosen as a candidate for reduction. Find the (next) production in which the due symbol is the right handle of the rule (i.e. it matches with the right-most immediate constituent). If there is none (any more) then go to (A-3). Focus on the category of the left immediate constituent in the rule. Find a row NEIGHBOR in the working table that contains the same category; the rows that must be searched are in the section whose right margins are identical with the left margin in the row CUR. If such an entry exists then create a new row (increase NEW) and store the category of the left side of the rule in it (i.e. the category to which the immediate constituents can be reduced). Adopt as left margin in row NEW the left margin in row NEIGHBOR and as right margin in row NEW the current position P. Set the pointer "left immediate constituent" to NEIGHBOR and the pointer "right immediate constituent" to CUR. Go to (A-2).

(A-3) **Final condition.** Increase CUR by 1. If CUR does not exceed NEW (i.e. the number of row in the table) then go to (A-2). Else increase the current position by 1. If the current position has moved behind the end of the input then go to (A-4), else go to (A-1).

(A-4) **Check for a complete result.** If the working table contains one or more rows with the left margin equal to 0 and the right margin equal to the number of words in the input then accept the input and finish the procedure. Else reject the input and finish the procedure.

Example:

GRAMMAR G3 (weakly equivalent to G2; lexically and grammatically ambiguous)

Rules			
(R-1)	S	->	$N_{u,d}$ $\mathbf{V_i}$
(R-2)	V_i	->	$\mathbf{V_t}$ $N_{u,d}$
(R-3)	V_i	->	$\mathbf{V_i}$ PP
(R-4)	N_d	->	det $\mathbf{N_{u,a}}$
(R-5)	N_a	->	adj $\mathbf{N_u}$
(R-6)	N_u	->	$\mathbf{N_u}$ PP
(R-7)	PP	->	**prep** $N_{u,d}$

Lexicon		
V_i	=	*{sleep, fish}*
V_t	=	*{study, visit, see, enjoy}*
det	=	*{the, no, my, many}*
adj	=	*{foreign, beautiful}*
N_u	=	*{tourists, pyramids, friends, fish, cans}*
N_d	=	*{Egypt, we, they}*
prep	=	*{in, by, with}*

Legend: The subscripts denote grammatical features: i = verbal phrase without an object or with the object recognized, u = nominal phrase without determiner but expecting one, d = nominal phrase with a determiner or with no need of a determiner, a = nominal phrase with an adjective. The immediate constituent printed in bold is the head of the corresponding phrase.

INPUT TABLE

Input:	*they*		*study*		*fish*		*in*		*cans*	
Lexicon:	N_d		V_t		V_i/N_u		prep		N_u	
Margins:	0	1	1	2	2	3	3	4	4	5

WORKING TABLE

Section 1:

	Category	Left margin	Right margin	Left constituent	Right constituent	Explanation
(1)	N_d	0	1	-	-	shift *they*

Section 2:

(2)	V_t	1	2	-	-	shift *study*

Section 3:

(3)	V_i	2	3	-	-	shift *fish*
(4)	N_u	2	3	-	-	shift *fish*
(5)	V_i	1	3	**2**	4	reduce by R-2
(6)	S	0	3	1	**5**	reduce by R-1

Section 4:

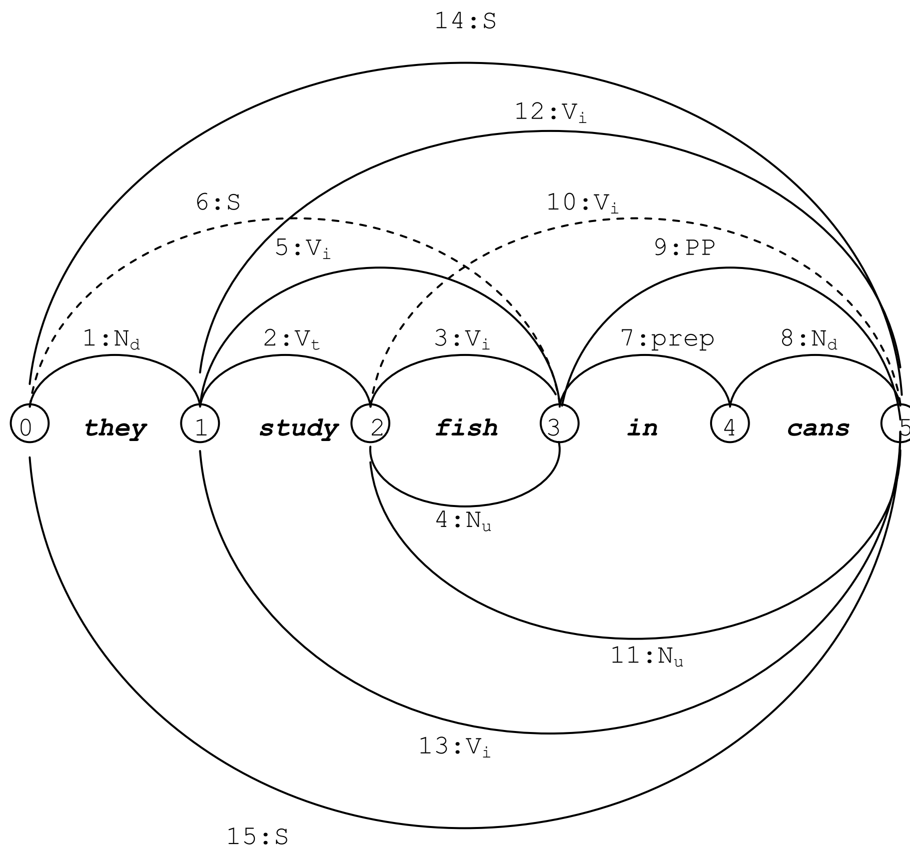
(7)	prep	3	4	-	-	shift *in*

Section 5:

(8)	N_d	4	5	-	-	shift *cans*
(9)	PP	3	5	**7**	8	reduce by R-7
(10)	V_i	2	5	**3**	9	reduce by R-3
(11)	N_u	2	5	**4**	9	reduce by R-6
(12)	V_i	1	5	**5**	9	reduce by R-3
(13)	V_i	1	5	**2**	11	reduce by R-2
(14)	S	0	5	1	**12**	reduce by R-1
(15)	S	0	5	1	**13**	reduce by R-1

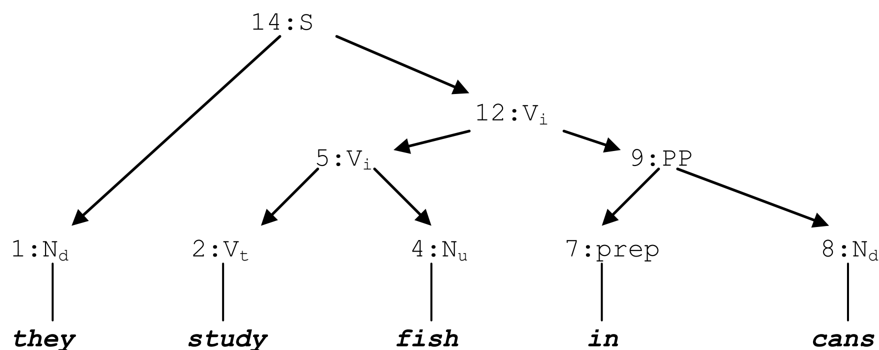
Chart

The working table can be turned into the following graphical chart representation (dotted lines denote obsolete reductions):



Producing a constituency and a dependency description as output

So far we concentrated on the recognition mechanism of our parsers. However, a parser must not only accept or reject the input but also output a structural description. Of course, it is easy to generate a binary constituent tree using the information in the working table of the Cocke algorithm. One must start with a row containing a segment that covers the whole input. The category of this segment is made the root of the tree. Then one subordinates the nodes for the left and the right immediate constituent, which are identified by the pointers in the row. For each new node the same subordination procedure is repeated until lexical categories are reached (i.e. the pointers to the left and the right immediate constituent are empty). The constituency tree rooted in the category of row 14 is the following:



The Chomsky normal form of constituency grammars, as it is used in the Cocke algorithm, is a good occasion to explore the possibility of creating a dependency tree as the output of the parser. A constituency grammar in Chomsky normal form contains binary rules only. Obviously each rule represents a syntagmatic relationship between two elements, one of which, from a dependency viewpoint, must be the dominant element and the other one the dependent element in the relation. In the grammar G3 the dominant element (i.e. the head constituent) is printed bold. Correspondingly, the pointers to the head constituents in the working table are printed bold and shaded. By means of these pointers a dependency tree is created as follows.

Again, one begins with a row that indicates a complete result. The number of this row is stored as the root node of a tree. This node is then replaced (rather than being subordinated as in the case of the constituency tree!) by the pointer to the dominant constituent in the same row, i.e. the one printed bold in the example. The pointer to the other constituent in the row is stored in a new node which is made subordinate to the dominating one. (The effect is that the heads are propagated upwards in the constituency structure while the dependents stay where they are.) For each leaf in the resulting tree the row is looked up that corresponds with the current value in the node. If there are pointers to immediate constituents in this row the same substitution of the head constituent and subordination of the non-head is performed and this process is repeated until rows are reached which represent lexical categories. Eventually, the lexical categories are substituted by lexemes and, thus, a dependency tree has emerged.

In the following illustration we use a bracketed notation and we add the category of the corresponding constituent to each node. For the two complete results existing in the working table above, the process is as follows:

1. (14:S)

   $(12{:}V_i\ (1{:}N_u))$

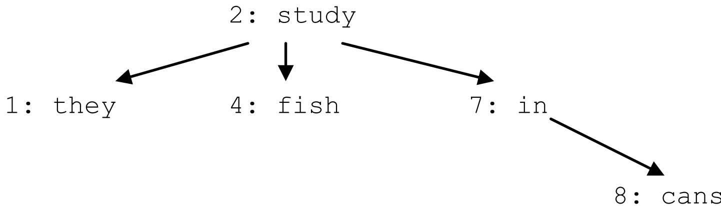
   $(5{:}V_i\ (1{:}N_u)\ (9{:}PP))$

   $(2{:}V_t\ (1{:}N_u)\ (4{:}N_u)\ (9{:}PP))$

   $(2{:}V_t\ (1{:}N_u)\ (4{:}N_u)\ (7{:}prep\ (8{:}N_u)))$

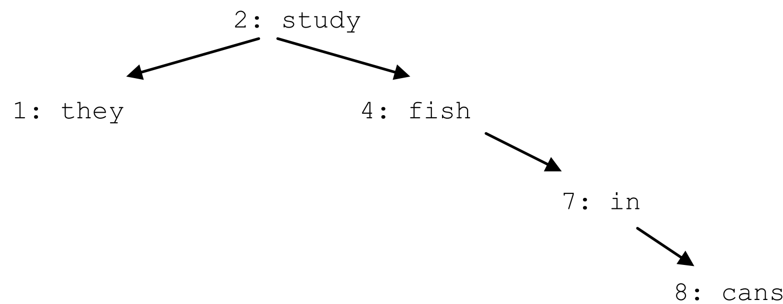
   (2: study (1: they)(4: fish)(7: in (8: cans)))

The resulting expression is in familiar appearance:



```
2.   (15:S)
     (13:Vi (1:Nu))
     (2:Vt (1:Nu) (11:Nu))
     (2:Vt (1:Nu) (4:Nu (9:PP)))
     (2:Vt (1:Nu) (4:Nu (7:prep (8:Nu))))
     (2: study (1: they) (4: fish (7: in (8: cans))))
```

This is the same as:



Evaluation

(1) **Efficiency.** As a chart parser PT-5 does not create anything twice; instead it reuses any intermediate result in all possible combinations. This can lead to a considerable amount of over-generation, though, because there is no restriction by the continuous left context.

(2) **Coverage.** Context-free rules; however no deletion rules admitted. Incapable to handle phenomena that go beyond context-free grammars. The possibility to generate dependency trees on the basis of a grammar with binary rules is a nice feature. As a bottom-up parser with a chart, PT-5 is not biased in favor of a single kind of utterance. It can detect the category of any constituent in the input, for example sentences as well as headings and other special constructions occurring in normal texts.

(3) **Drawing up lingware.** The same as PT-1. The use of complex categories and the implementation of a unification mechanism should pose no problem.

PT-6. Table-controlled shift-reduce parser

Illustrates:

(1) Connection between grammar and parser: An action table and a goto table is compiled from the grammar.
(2) Linguistic structure assigned: Constituency trees.
(3) Grammar specification format: Context-free production rules.
(4) Recognition strategy: Category expansion (transformed into state transitions) combined with category reduction (bottom-up) at run-time.
(5) Processing the input: Left-to-right, one pass; left associative.
(6) Treatment of alternatives: Avoiding conflicts by looking ahead (implicit in the control table); parallel processing if necessary.
(7) Control of results: Goal oriented recognition.

References: *Aho/Ullman* 1977: 198-248, *Tomita* 1986, *Hellwig* 1989: 389ff.

Prerequisites:

(P-1) An ordered set of **production rules** of a context-free phrase structure grammar without restrictions.

(P-2) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-3) An input table which contains **the words of the input together with their categories** and their position in the sequence. The categories have been assigned to the words according to the lexicon in a preceding phase.

(P-4) A variable P for **the current position** in the course of moving forward from the beginning to the end of the input. At the beginning P identifies the first word.

(P-5) A control table, consisting of a matrix of states (in the rows) and the lexical symbols as well as the non-lexical symbols (in the columns). The part of the matrix with lexical symbols in the columns is called **the action table**, the part with the non-terminal symbols is called **the goto table**. The fields in the matrix contain instructions for the parser. One column in the action table is labeled with the symbol `'$'` which denotes the end of the input string.

(P-5) A **working space** which has the form of a network. The nodes are labeled with states and the arcs are labeled with structural descriptions (e.g. with parse trees in bracketed notation). The states correspond to certain points in the input. Hence, the structural descriptions at the arcs represent temporary analysis results of the strings between two margins. At the beginning the working area contains only one node denoting the state 0 and no arcs.

(P-6) A variable for **the current state** of the parser, which is initially set to the value 0.

Algorithm:

(A-1) **Consultation of the control table.** Determine the lexical category of the next word in the input. (This is equivalent to a look ahead of k=1.) Look up the field in the action table that is at

the intersection between the current state and the lexical category. If the end of the input is reached, then look up the field in the column `'$'`. Go to (A-2).

(A-2) **Final condition.** If the field is empty then reject the input and finish the procedure. If the field contains the entry "accept" (abbr. `"acc"`) then print out the structure in the work space as the result of the analysis and finish the procedure. Else go to (A-3).

(A-3) **Shift.** If the field contains the instruction "shift z" (abbr. `"sh z"`), where z is a state, then add a new arc to the last node in the working space (which represents the current state) and label it with the category of the next element in the input. Link the new arc to a new node labeled z. Make z the new current state. Increase the current position by 1. Go to (A-1).

(A-4) **Reduce.** If the field contains the instruction "reduce r" (abbr. `"re r"`), where r is the number of a rule, then combine the arcs in the work area that correspond to the immediate constituents in the rule r in order to form a new arc. As many arcs have to be traced back in the network as there are constituents in the rule. The arcs passed in this process have to be removed. Form a new description by substituting the labels of the former arcs under the category of the left-hand side of the rule. Label the new arc with this description. The node at the beginning of the reduced constituents is defined as the temporary state. Look up the field in the goto table that is at the intersection of the temporary state and the category of the reduction. Make the state z in this field the new current state. Add z as new node to the working area and link the new arc to it. Go to (A-1).

Example

GRAMMAR G1 (rules to be used in the reduce step):

```
(re1)    S  -> NP VP            (re5)  NP -> n
(re2)    VP -> vi               (re6)  NP -> det n
(re3)    VP -> vt NP            (re7)  NP -> det adj n
(re4)    VP -> vt NP PP         (re8)  PP -> prep NP
```

ACTION TABLE

Z	det	adj	n	prep	vi	vt	$
0	sh9		sh8				
1							acc
2					sh4	sh5	
3							re1
4							re2
5	sh9		sh8				
6				sh13			re3
7							re4
8				re5	re5	re5	re5
9		sh11	sh10				
10				re6	re6	re6	re6
11			sh12				
12				re7	re7	re7	re7
13	sh9		sh8				
14							re8

GO TO TABLE

NP	PP	VP	S
2			1
		3	
6			
	7		
14			

INPUT TABLE

Input:	*they*	*visit*	*friends*	*in*	*Egypt*
Lexicon:	**n**	**vt**	**n**	**prep**	**n**
Position:	1	2	3	4	5

WORKING SPACE

P:	States and Descriptions:	Explanation:
0	0	starting state
1	0 —*n*— 8	shift 8
1	0 —*NP(n)*— 2	reduce 5, goto 2
2	0 —*NP(n)*— 2 —*vt*— 5	shift 5
3	0 —*NP(n)*— 2 —*vt*— 5 —*n*— 8	shift 8
3	0 —*NP(n)*— 2 —*vt*— 5 —*NP(n)*— 6	reduce 5, goto 6
4	0 —*NP(n)*— 2 —*vt*— 5 —*NP(n)*— 6 —*prep*— 13	shift 13
5	0 —*NP(n)*— 2 —*vt*— 5 —*NP(n)*— 6 —*prep*— 13 —n— 8	shift 8
5	0 —*NP(n)*— 2 —*vt*— 5 —*NP(n)*— 6 —*prep*— 13 —*NP(n)*— 14	reduce 5, goto 14
5	0 —*NP(n)*— 2 —*vt*— 5 —*NP(n)*— 6 —*PP(prep NP(n))*— 7	reduce 8, goto 7
5	0 —*NP(n)*— 2 —*VP(vt NP(n) PP(prep NP(n)))*— 3	reduce 4, goto 3
5	0 —*S(NP(n) VP(vt NP(n) PP(prep NP(n))))*— 1	reduce 1, goto 1 accept

Construction of the control table

The action table and goto table can be automatically compiled from a given grammar. The underlying idea is the following: There are several states during the processing of each rule of the grammar, which differ in how many of the immediate constituents have already been checked and how many still follow. First, we have to find out how many states there are. Each state can be represented by a so-called divided production (compare PT-4).

Given the rule: `X -> Y Z`

we mark the boundaries between the constituents with a dot and arrive at the following possible states:

```
X -> .Y Z
X -> Y .Z
X -> Y Z.
```

Trying to obtain the set of all states, we first add a rule to the grammar that can be turned into a dived production at state (Z-0), that is the situation at the beginning of the parse before any constituent of the initial category has been seen:

```
(Z-0)    S' -> .S
```

The same state, i.e. the same position in the input string, is projected into all rules that will be used in the course of the recursive expansion of the symbol behind the dot. If we base ourselves on grammar G1, the following states are included in the state (Z-0), too:

```
S  -> .NP VP
NP -> .n
NP -> .det n
NP -> .det adj n
```

The next state is obtained by moving the dot to the right by one constituent. The result is in the case of (Z-0):

```
(Z-1)    S' -> S.
```

A projection into further rules is not possible here, because there are no constituents on the right of the dot any more. One now continues to divide the other rules into states. Since the state before the first use of a rule is already included in a previous projection, it is sufficient to begin with the state after the first immediate constituent in the rule. The rest of G1 is thus transformed into the following divided productions. As far as two rules of the grammar coincide in the constituents before the dot, they are subsumed under the same state (cf. states 5, 6 and 9).

```
(Z-2)  S  -> NP .VP
                          VP -> .vi
                          VP -> .vt NP
                          VP -> .vt NP PP
(Z-3)  S  -> NP VP.
```

```
(Z-4)   VP -> vi.
(Z-5)   VP -> vt. NP
        VP -> vt. NP PP
                                NP -> .n
                                NP -> .det n
                                NP -> .det adj n

(Z-6)   VP -> vt NP.
        VP -> vt NP. PP
                                PP -> .prep NP
(Z-7)   VP -> vt NP PP.

(Z-8)   NP -> n.
(Z-9)   NP -> det. N
        NP -> det. adj n
(Z-10)  NP -> det n.
(Z-11)  NP -> det adj. n
(Z-12)  NP -> det adj n.
(Z-13)  PP -> prep. NP
                                NP -> .n
                                NP -> .det n
                                NP -> .det adj n
(Z-14)  PP -> prep NP.
```

The states (Z-0) through (Z-14) form the rows in the action table as well as in the goto table. The columns of the action table are labeled with the lexical categories and '$' (i.e. the end of the input). The columns of the goto table are labeled with the non-terminal categories of the grammar. The values of the fields in the two tables are computed as follows (compare the action and goto table in the example above):

Walk through the above collection of states and through the divided productions in each state.

(1) If the dot is in front of a lexical category then enter "shift z" in the field that is at intersection between the current state and this lexical category in the action table. z is the number of the state that would be reached after the lexical category is processed by the parser. This state is represented by the same rule as the current one except for the dot which is moved behind the category. To find out the number of this state one must look in which of the collection of states the rule with the moved dot occurs.

(2) If the dot is in front of a non-terminal category then enter "goto z" in the field that is at intersection between the current state and this non-terminal category in the goto table. z is the number of the state that would be reached after the non-terminal category is processed by the parser. This state is represented by the same rule as the current one except for the dot which is moved behind the category. To find out the number of this state one must look in which of the collection of states the rule with the moved dot occurs.

(3) If the dot is at the end of a production r then enter "reduce r" in certain columns in the row of the current state of the action table, where r is the key of the rule in question. The idea is to postpone the reduction step until the next constituent has been seen that cannot belong to the constituents of the actual rule. This is what looking-ahead means. Therefore, "reduce r" should be assigned to all categories in the action table that can occur as the first lexical category of any constituent that can follow category X, where X is the head category of the production r. If X also occurs at the end of the sentence then "reduce r" must be entered in the column "$" as well. The respective lexical categories are found by means of the functions FIRST and FOLLOW, see below.

(4) If the dot occurs at the end of the S'-production (e.g. `S' -> S.`) then insert "accept" in the field at the intersection of the corresponding state and the column "$".

The function FOLLOW yields the set of possible constituents following a given constituent according to the rules of a grammar. If in any rule an immediate constituent is following the given one then this constituent is among the returned values. If the given constituent is the last of the immediate constituents in a rule then the function FOLLOW is invoked recursively with the head of the rule as the argument. Let us illustrate this by G1:

```
(re1)     S  -> NP VP
(re2)     VP -> vi
(re3)     VP -> vt NP
(re4)     VP -> vt NP PP
(re5)     NP -> n
(re6)     NP -> det n
(re7)     NP -> det adj n
(re8)     PP -> prep NP
```

The function FOLLOW returns the following values:

```
FOLLOW(S)  = $
FOLLOW(VP) = FOLLOW(S) = $
FOLLOW(NP) = VP
FOLLOW(NP) = FOLLOW(VP)= $
FOLLOW(NP) = PP
FOLLOW(NP) = FOLLOW(PP)= FOLLOW(VP) = $
FOLLOW(PP) = FOLLOW(VP) = FOLLOW(S) = $
```

FIRST receives the output of FOLLOW and returns the set of the initial lexical constituents for each non-terminal category according to the grammar rules:

```
FIRST($)   = $
FIRST(VP)  = vi, vt
FIRST(NP) = det, n
FIRST(PP)  = prep
```

Algorithm for non-deterministic grammars

(akin to the Tomita parser)

If the creation of the action table and the goto table, according to the described procedure, results in no more than one instruction in any field of the matrix then the parser proceeds deterministically and the described language is of the LR(k)-type (left-to-right processing, right derivation, looking k symbols ahead), with k=1. If the fields of the matrix receive multiple entries, the language includes ambiguous string. This is no reason for abandoning table-controlled shift-reduce parsing. The conflicting instructions in the tables (e.g. in the goto table of the example below) can simply be followed in a parallel fashion. For this purpose the network in the working area is allowed to branch, i.e. more than one arc may leave a node and arrive at diverging states. If at a later position in the input an unambiguous portion occurs, the divergent states may be connected and be continued by a joined arc. A reduction may follow either of the alternative paths. The effect of this representation is the same as that of a well-formed substring table: no work needs to be done twice. (Tomita's parser, which belongs to the same prototype, achieves this goal with his so-called "forests" of parse trees.)

Example:

GRAMMAR G2 (ambiguous, includes G1)

Rules	
(R-1)	S -> NP VP
(R-2)	VP -> vi
(R-3)	VP -> vt NP
(R-4)	VP -> VP PP
(R-5)	NP -> n
(R-6)	NP -> det n
(R-7)	NP -> det adj n
(R-8)	NP -> NP PP
(R-8)	PP -> prep NP

Lexicon		
vi	=	*{sleep, fish}*
vt	=	*{study,visit, see, enjoy}*
det	=	*{the, no, my, many}*
adj	=	*{foreign, beautiful}*
n	=	*{tourists, pyramids, friends, fish, cans, Egypt, we, they}*
prep	=	*{in, by, with}*

The states of processing G2:

```
(Z-0)  S' -> .S
              S  -> .NP VP
              NP -> .n
              NP -> .det n
              NP -> .det adj n
              NP -> .NP PP
(Z-1)  S' -> S.
(Z-2)  S  -> NP .VP
              VP -> .vi
              VP -> .vt NP
              VP -> .VP PP
(Z-3)  S  -> NP VP.
(Z-4)  VP -> vi.
(Z-5)  VP -> vt. NP
              NP -> .n
              NP -> .det n
              NP -> .det adj n
              NP -> .NP PP
(Z-6)  VP -> vt NP.
(Z-7)  VP -> VP .PP
              PP -> .prep NP
(Z-8)  VP -> VP PP.
```

```
(Z-9)    NP -> n.                    (Z-15)   NP -> NP PP.
(Z-10)   NP -> det. n                (Z-16)   PP -> prep. NP
         NP -> det. adj n                          NP -> .n
(Z-11)   NP -> det n.                              NP -> .det n
(Z-12)   NP -> det adj. n                          NP -> .det adj n
(Z-13)   NP -> det adj n.                          NP -> .NP PP
(Z-14)   NP -> NP. PP                (Z-17)   PP -> prep NP.
               PP -> .prep NP
```

ACTION TABLE

Z	det	adj	n	prep	vi	vt	$
0	sh10		sh9				
1							acc
2					sh4	sh5	
3							re1
4				re2			re2
5	sh10		sh9				
6				re3			re3
7				sh16			
8				re4			re4
9				re5	re5	re5	re5
10		sh12	sh11				
11				re6	re6	re6	re6
12			sh13				
13				re7	re7	re7	re7
14				sh16			
15				re8	re8	re8	re8
16	sh10		sh9				
17				re9	re9	re9	re9

GO TO TABLE

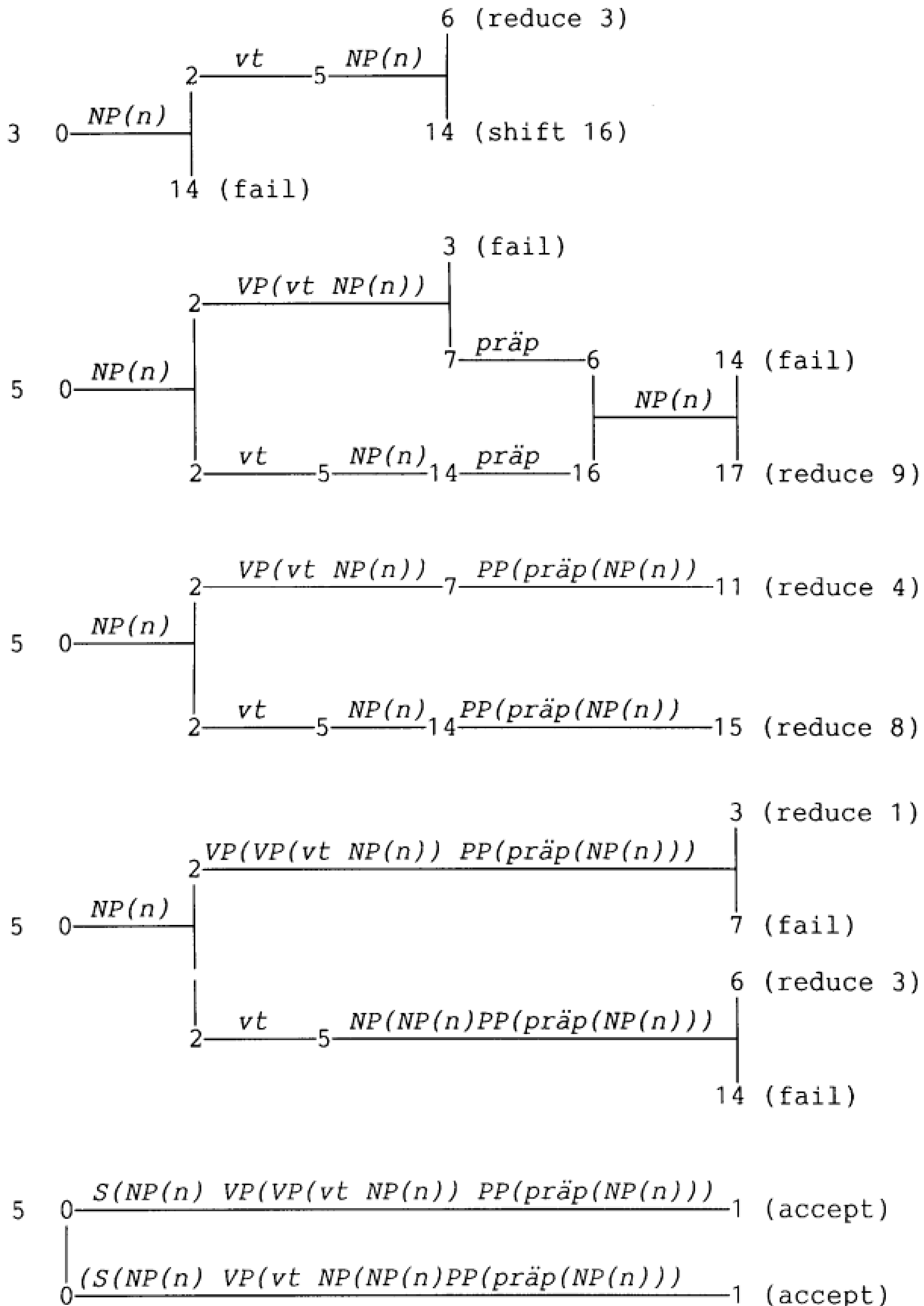
NP	PP	VP	S
2,14			1
		3,7	
6,14			
	8		
	15		
14,17			

INPUT TABLE

Input:	*they*	*study*	*fish*	*in*	*cans*
Lexicon:	**n**	**vt**	**vi/n**	**prep**	**n**
Position:	1	2	3	4	5

WORK AREA (subset)

P: States and Descriptions: (Next step):



Evaluation

(1) **Efficiency.** This kind of parser, which is well-known in computer science and is also advocated for NLP (Tomita 1985) is impressively efficient. PT-6 processes the example, that caused 10 times backtracking in PT-1, in a completely deterministic way. There are several reasons for this efficiency. PT-6 combines an expectation driven strategy with a bottom-up strategy. The top-down expansion of categories is translated into the procedural form of the action table and goto table. At run-time the parser is data-driven. The most interesting feature is the way that looking ahead is integrated into the action table. The command of reducing a category is deferred not only until the current constituent is complete but rather until the first element of a constituent shows up (FIRST of FOLLOW) that can by no means be included in the current constituent under construction. In this way PT-6 can be pretty sure that the reduction is correct. In addition, the parse is left-associative. So, there will be very little over-generation.

(2) **Coverage.** Unrestricted context-free rules.

(3) **Drawing up lingware.** A disadvantage of PT-6 is the necessity to construct a new control table each time a change is made in the grammar. The amount of effort for this task grows exponentially with the size of the grammar. This is a serious drawback because usually one would like to switch between updating the lingware and testing it within minutes.

PT-7. FTN-parser for regular expressions

Illustrates:

(1) Connection between grammar and parser: Compiled parser; the grammar is transformed into a state transition table representing a finite state transition network (FTN)
(2) Linguistic structure assigned: Per se no structure is detected.
(3) Grammar specification format: Regular expression or corresponding finite state transition network (FTN)
(4) Recognition strategy: Pattern-oriented analysis; transition in a network.
(5) Processing the input: Left-to-right, one pass; left associative.
(6) Treatment of alternatives: A deterministic FTN can be derived from any FTN.
(7) Control of results: Goal oriented recognition; the goal is to reach an end state in the network

References: *Aho/Sethi/Ullman* 1986: 124ff, *Hellwig* 1989: 406ff.

Regular expressions

Regular expressions are patterns that describe the strings of a regular language. A regular language is the type of language generated by a regular grammar (Chomsky hierarchy Type 3). Regular expressions are a way to formulate such a grammar. The metalanguage M of regular expressions describing the object language L is defined as follows:

(i) 'ε' is an expression of M and denotes the empty character string.

(ii) If a is an element of the vocabulary of L then a is an expression of M and denotes the occurrence of the character string "a" in an instance of L.

(iii) If K is a subset of the vocabulary of L then K is an expression of M and denotes the occurrence of an arbitrary element of category K in an instance of L.

(iv) If r and s are expressions of M then rs is an expression of M and denotes the concatenation of the strings denoted by r and s in an instance of L.

(v) If r and s are expressions of M then r|s is an expression of M and denotes the occurrence of a the string denoted by r or by s in an instance of L.

(vi) If r is an expression of M then r^+ is an expression of M and denotes the concatenation of one ore more occurrences of the string denoted by r in an instance of L.

(vii) If r is an expression of M then r^* is an expression of M and denotes the concatenation of one ore more occurrences of the string denoted by r or no occurrence at all in an instance of L. . r^* is the same as $r^+ | \varepsilon$.

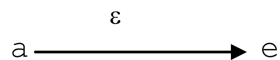
(viii) If r is an expression of M then $r^?$ is an expression of M and denotes the optional occurrence of the string denoted by r in an instance of L. r? is the same as r|ε.

(ix) Brackets '(', ')' and '{', '}' are used to group expressions with respect to concatenation or alternation. If r is an expression of M then (r) and {r} are expressions of M.

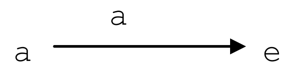
Finite state transition networks

A network consists of nodes and labeled arcs connecting nodes. The nodes denote states of matching the network with some input. The labels at the arcs are description of the input. The arcs themselves denote the concatenation of input segments. A finite state transition network can be designed from scratch. It can also be created automatically from a regular expression. Construction of networks from regular expressions is done as follows. At first the elementary expressions of a formula are associated with networks according to (i) through (iii). Then the partial networks are combined according to (iv) through (ix). Let the starting state of a network be a, the end state be e, and an intermediate state be s_i. A network is associated with each regular expression with exactly one starting state and one end state.

(i) The expression 'ε' is mapped by the following network:

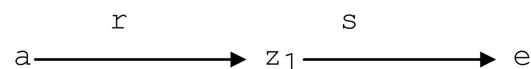


(ii) The expression a (i.e. a lexical element of L) is mapped by the following network:

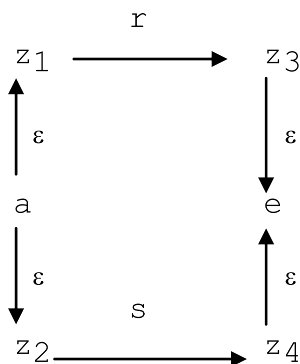


(iii) The expression (i.e. an element of the subset K of the elements of L) is mapped by the following network:

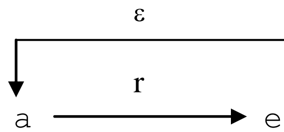
(iv) The expression rs is mapped by the following complex network:



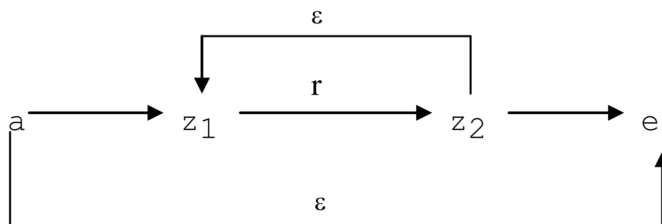
(v) The expression r|s is mapped by the following network:



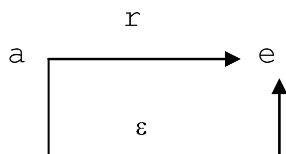
(vi) The expression r^+ is mapped by the following network:



(vii) The expression r^* is mapped by the following network:



(viii) The expression r? is mapped by the following network:



(ix) Expressions that are surrounded by brackets '(', ')' and '{', '}' must be mapped by a complete network before this network can be combined with the networks corresponding to other parts of the regular expression. Multiple bracketing must be processed proceeding from inner to outer brackets.

Construction of a deterministic finite state transition network

One starts with a non-deterministic FTN, for example one generated from a regular expression as described above. Starting with state a in the non-deterministic FTN, a new network is constructed. The individual states of the new network correspond to sets of states of the original one, namely the set formed by a given state and all the states that are reachable from the given one via an ε-arc (a so-called jump-arc - in the resulting network there will be no jump arcs), or via an arc with the same basic symbol (in the resulting network there will be just one arc for identical expressions). The nodes in the new network are connected by arcs in such a way, that all and only those transitions are possible that were possible between the member states that now form the new states. (The example below will make this clear.)

FTN Recognizer

Prerequisites:

(P-1) A deterministic **finite state transition networks** stored as a **state transition table**. Each row of the table represents one arc in the network. Each entry consists of a starting state, a label, and a target state. Possible end states of the network are marked by "/e".

(P-2) A **lexicon** which associates categories (parts-of-speech, grammatical features) with the words in the input.

(P-3) A variable for the actual state. At the beginning the value of this variable is the starting state of the whole network.

(P-4) A variable containing the current position in the input.

Algorithm:

(A-1) **Transition.** If there is an entry in the transition table with the actual state as the starting state, the category of the word in the actual position as the label and a target state then make the target state the actual state, increase the current position by one and go to (A-2). If there is no such entry go to (A-3).

(A-2) **Final condition.** If the current position is not greater than the number of element in the input then go to (A-1). Else if the actual state is an end state in the network then accept the input and return, otherwise reject the input and return.

Example

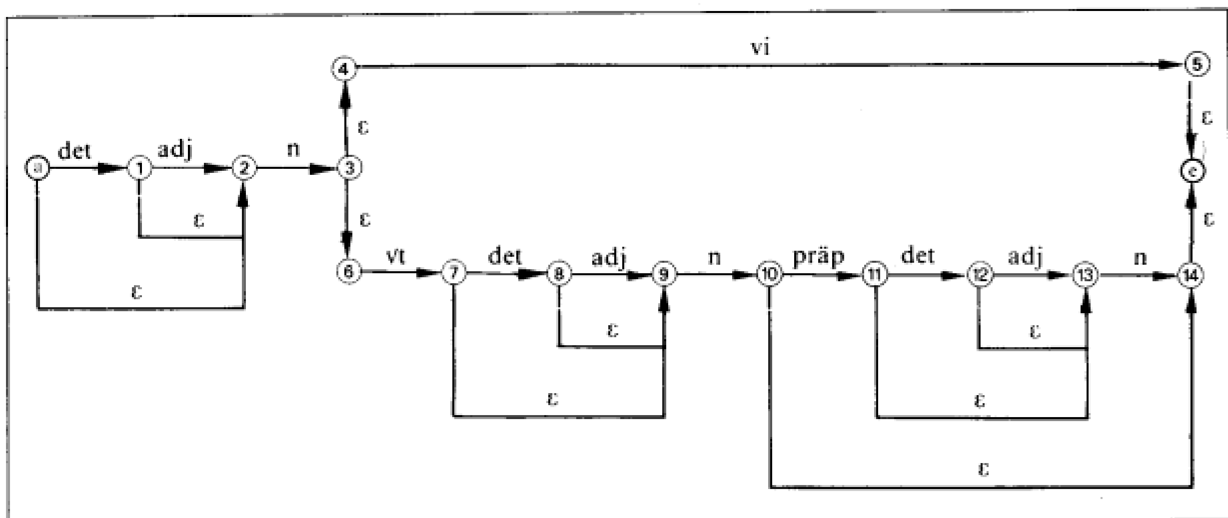
GRAMMATIK G4 (equivalent with G1)

Regular expression:

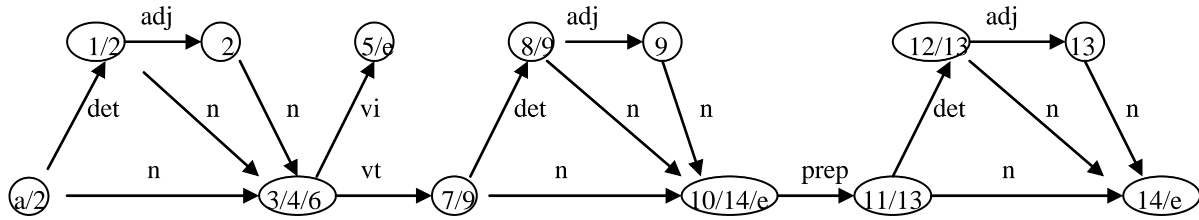
(det adj?)? n {vi | vt (det adj?)? n (präp (det adj?) n)? }

Lexicon as in G1

Non-deterministic finite state transition network:



Deterministic finite state transition network:



(The names of states are composed of the original state names in the non-deterministic net for the sake of perspicuity.)

Finite state transition table:

starting state	label	target state
a/2	det	1/2
a/2	n	3/4/6
1/2	adj	2
1/2	n	3/4/6
2	n	3/4/6
3/4/6	vi	5/e
3/4/6	vt	7/9
7/9	det	8/9
7/9	n	10/14/e
8/9	adj	9
8/9	n	10/14/e
9	n	10/14/e
10/14/e	prep	11/13
11/13	det	12/13
11/13	n	14/e
12/13	adj	13
12/13	n	14/e
13	n	14/e

Trace of the recognizer:

Input:		*they*	*visit*	*friends*	*in*	*Egypt*
Lexicon:		**n**	**vt**	**n**	**prep**	**n**
Actual position:	0	1	2	3	4	5
Actual state:	a/2	3/4/6	7/9	10/14/e	11/13	14/e

Evaluation

(1) **Efficiency.** Very efficient if the input and the task at hand can resort to regular expressions.

(2) **Coverage.** Very limited. Suits regular languages only and usually does not assignment a syntactic structure.

PT-8. Augmented transition networks (ATN)

Illustrates:

(1) Connection between grammar and parser: Procedural parser, no separation of grammar and parser.
(2) Linguistic structure assigned: Capability to recognize any input and to produce any output.
(3) Grammar specification format: Program instructions, for example in LISP.
(4) Recognition strategy: Pattern-oriented; transition in a network, usually top-down, depth-first.
(5) Processing the input: Left-to-right, one pass (depth-first); left associative.
(6) Treatment of alternatives: Backtracking can be avoided to a large extent by saving context information in registers.
(7) Control of results: Goal oriented recognition.

References: *Woods* 1969**,** *Bates* 1978 , *Winograd* 1983: 195ff., *Hellwig* 1989: 412ff,

Prerequisites:

(P-1) The definition of an **ATN formalism**.

(P-2) An **augmented transition network** consisting of code in the chosen formalism.

(P-3) A **lexicon** which associates basic grammatical categories (parts-of-speech, grammatical features) with lexical items.

(P-4) An input table which contains **the words of the input together with their categories** and their position in the sequence. The categories have been assigned to the words according to the lexicon.

(P-5) A list of **register lists**, one for each new level of recursion. A register list is a list of pairs comprising a register name and a register content. (This organization allows for arbitrarily many registers on each level.) The action SETR puts a new pair on top of the list. Older pairs are accessible again if backtracking reactivates a former state of the register list.

(P-6) A **HOLD-list** which contains information about formerly processed constituents.

(P-7) Variables whose values define a so-called **configuration**. A configuration consists of the current node, the current arc, and the current state of the register list.

(P-7) A **pop-up stack**. The current configuration is put on top of the stack before a PUSH-arc leads into a deeper level of recursion.

(P-8) A **backtracking store**. If there is an alternative arc in the network than the current configuration and the alternative arc are stored in the backtracking store.

Definition of an ATN formalism

An augmented transition network (ATN) is conceptually based on a recursive transition network (RTN) with nodes, arcs and the recursive invocation of sub-nets. The information associated with an arc is augmented by conditions to be tested first and actions to be performed before or after passing the arc. Tests and actions are formulated in a programming language, usually in LISP. It is possible to allocate registers which can be referred to in conditions and actions and there may even be global variables. Therefore, an ATN is actually a Turing machine. It can generate any enumerable language. However, such an automaton is too powerful for processing natural languages. Therefore, the ATN-formalism is subject to self-imposed restrictions. Here is a definition of the most common features (terminal categories in italics):

<augmented transition network> := <set of arcs>*

<set of arcs> := <*node*> <arc>*

<arc> := <WRD-arc> | <CAT-arc> | <JUMP-arc> | <VIR-arc> | <PUSH-arc> | <POP-arc>

<WRD-arc> := WRD <*lexical element*> <test>* <action>* <transition>

<CAT-arc> := CAT <*lexical category*> <test>* <action>* <transition>

<JUMP-arc> := JUMP <transition> <test>* <action>*

<VIR-arc> := VIR <*constituent*> <test>* <action>* <transition>

<PUSH-arc> := PUSH <*node*> <test>* <pre-action>* <test>*<action>* <transition>

<POP-arc> := POP <form> <test>*

<test> := T | <*condition*>

<pre-action> := <send register>

<action> := <*condition*> <set register> | <*condition*> <lift register> | <HOLD-action>

<set register> := SETR <*register*> <form> | ADDR <*register*> <form>

<send register> := SENDR <*register*> <form>

<lift register> := LIFTR <*register*> <form>

<HOLD-action> := HOLD <*constituent*> <form>

<form> := <current lexical item> | <current category> | <content of register> | <build construction>

<current lexical item> := *

<current category> := GETF <*attribute*> | "<*value*>"

<content of register> := GETR <*register*>

<build construction> := BUILDQ <*frame*> <*register*>*

<transition> := TO <*node*>

Algorithm:

(A) **Navigating through the network**

(A-1) **Produce a starting configuration.** The current position equals 1; the current node is the first one in the net; the current arc is the first one that leaves the node; the register list is empty.

(A-2) **"Snapshot".** If there is another arc leaving the current node after the current arc then save the current configuration, the contents of the HOLD-list, the contents of the pop-up stack and the alternative arc in the backtracking store.

Process the current arc as follows:

WRD-arc: If the specified lexical element is identical to the current word in the input then continue; otherwise go to (A-3). Evaluate the specified tests. If a test fails then go to (A-3). Perform the specified actions. Increment the current position by 1. Turn the node after TO into the new current node and its first arc into the current arc. Go to (A-2).

CAT-arc: If the specified lexical category is identical to the category of the current word then continue; otherwise go to (A-3). Evaluate the specified tests. If a test fails then go to (A-3). Perform the specified actions. Increment the current position by 1. Turn the node after TO into the new current node and its first arc into the current arc. Go to (A-2).

JUMP-arc: Evaluate the specified tests. If a test fails then go to (A-3). Perform the specified actions. Without incrementing the current position, turn the node after TO into the new current node and its first arc into the current arc. Go to (A-2).

VIR-arc: If the specified constituent is contained in the HOLD-list then continue; otherwise go to (A-3). Evaluate the specified tests. If a test fails then go to (A-3). Perform the specified actions. Without incrementing the current position, turn the node after TO into the new current node and its first arc into the current arc. Go to (A-2).

PUSH-arc: Evaluate the specified tests. If a test fails then go to (A-3). Perform the specified pre-action. Put the current configuration on top of the pop-up stack. Produce a new configuration for the next deeper level of recursion, i.e. turn the current node into the node specified in the arc and its first arc into the current arc. Leave the current position as it is. Start a new register list. Go to (A-2).

POP-arc: Evaluate the specified tests. If a test fails then go to (A-3). If the top-most level of recursion is reached (i.e. the pop-up stack is empty) then go to (A-4). Otherwise restore the configuration of the next higher level of recursion according to the configuration on top of the pop-up stack. Remove that configuration from the stack. Transfer the specified form as well as the position reached to the higher level. Continue processing along the PUSH-arc that called the subordinated network and perform the actions that are specified there. Turn the node after TO into the new current node and its first arc into the current arc. Go to (A-2).

(A-3) **Backtracking.** If there is an entry in the backtracking store then restore the configuration specified in the top-most entry of the backtracking store, restore the specified state of the pop-up stack as well as the state of the register list. Remove the entry from the backtracking store. Go to (A-2). If there is no entry then refuse the input and stop.

(A-4) **Final condition.** If the end of the input is reached then one successful parse was found. Output the result. If all readings are to be found then go to (A-3). Otherwise stop. If the end of the input is not reached then go to (A-3).

(B) **Evaluation of tests**

The truth value of a condition is computed. The condition may contain various predicates and arguments. It can refer to registers and other parts of configurations. Conditions can be combined by boolean operators. The test fails if the truth value of the condition is "false", otherwise it succeeds.

(C) **Actions**

SETR-action: Put a new pair on the current register list consisting of the register name and the specified form.

ADDR-action: Add the specified form to the form(s) associated with the specified register name.

SENDR-action: This is a pre-action which is applicable only on PUSH-arcs. Put the register name and its content on the register list of the next lower level of recursion.

LIFTR-action: This is the inverse action of the SENDR-action. Put the register name and its content on the register list of the next higher level of recursion.

HOLD-action: Put the specified form as an instance of the specified constituent on the HOLD-list. The HOLD-list is global, i.e. it is accessible on all levels. The HOLD-list is consulted when the parser processes a VIR-arc.

(D) **The evaluation of forms**

The variable * , as a rule, substitutes for the lexical element at the current input position. The use in PUSH- and POP-arcs is an exception: Within the action of a PUSH-arc, * is identified with the result that was returned by the preceding POP-arc.

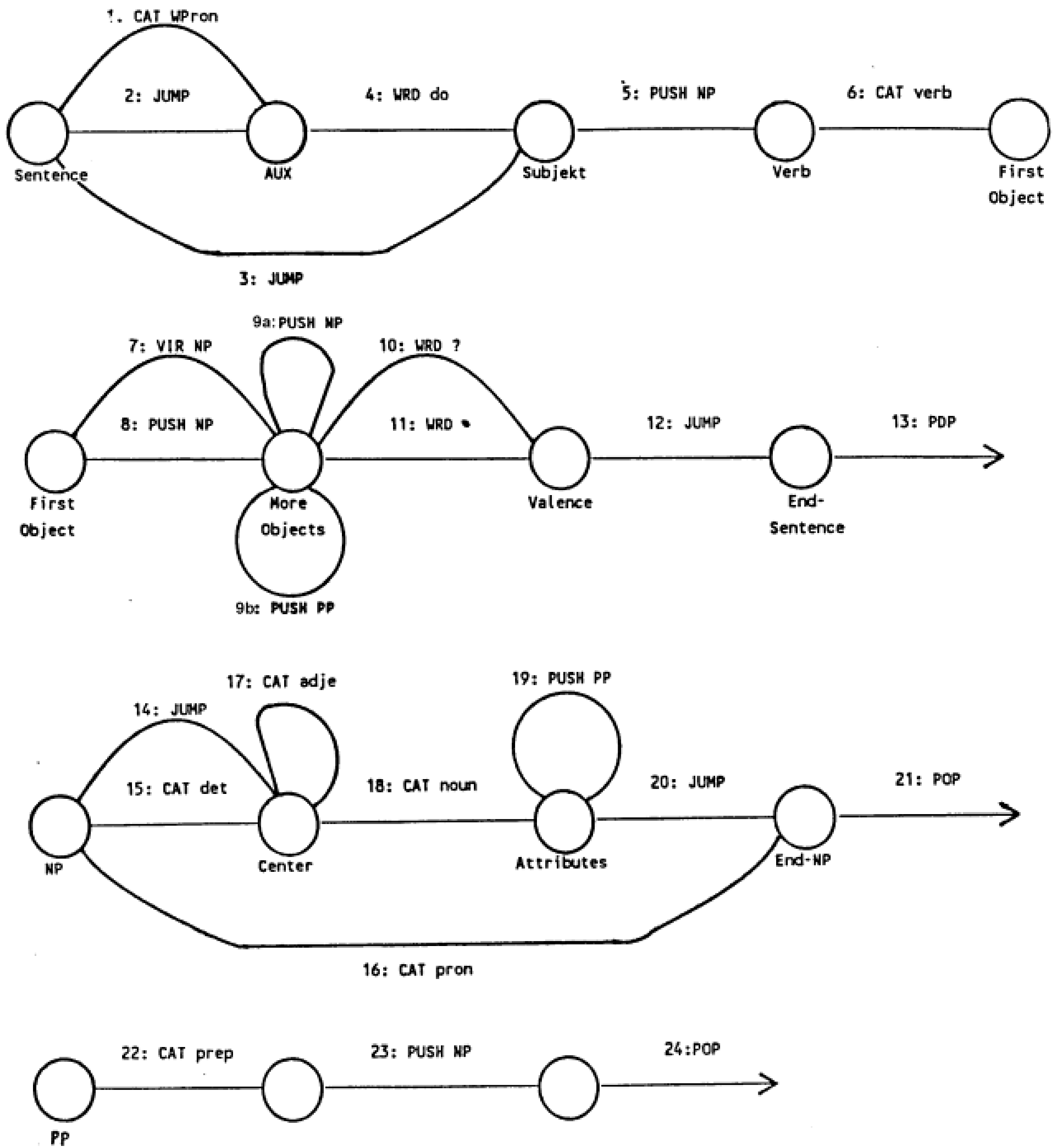
In the case of GETF return the value of the specified attribute assigned to the current lexical element, e.g. its number, gender, case etc.

In the case of GETR return the content of the specified register. If the register is empty return "F".

BUILDQ is a function that is used in order to produce the desired output of the parser. The function expects a frame, i.e. a fragment of a structure with constants and variables, and the registers from which the substitutes for the variables are to be taken. A common symbol for a variable is "+". Replace each occurrences of "+" in the frame by the contents of the corresponding register listed in the BUILDQ instruction. The symbol * is used within the frame in order to refer to the current lexical item. No register is following the frame in this case.

Example

G5 - AN ATN NETWORK IN GRAPHICAL REPRESENTATION



ATN-PROGRAM. i.e. the grammar G5 in procedural form:

Usually LISP is used for programming ATNs. Here, the program is sketched in pseudo-code. Expressions in italics don't belong to the syntax but were added in order to improve readability. Functions that return a value (i.e. functions that are evaluated before the value is used) are in angled brackets.

The basic idea of G5 is to collect the subject and the objects at various positions in the sentence (depending on the word order for questions and statements), store all the information in the register *Complements*, then (at the node *Valence)* compare the contents of the register with the valency subcategorization of the verb, and output a dependency description consisting of syntactic roles and lexemes.

```
node Sentence

      arc 1: CAT W_pron
            action HOLD NP *
            transition TO Aux

      arc 2: JUMP
            transition TO Aux

      arc 3: JUMP
            action SETR illocution "assertion"
            action SETR verb_form "finite"
            transition TO Subject

node Aux

      arc 4: WRD do
            action SETR number <GETF number>
            action SETR person <GETF person>
            action SETR verb_form "infinitive"
            action SETR illocution "question"
            transition TO Subject

node Subject

      arc 5: PUSH NP
            pre-action SENDR number
            pre-action SENDR person
            action ADDR complements "subject"
            action ADDR arguments <BUILDQ (subject: *)>
            transition TO Verb

node Verb

      arc 6: CAT verb
            test <GETR verb_form> equals <GETF form> or
                  {<GETR number> includes <GETF number> and
                   <GETR person> includes <GETF person>}
            action SETR predicate *
            transition TO First_Object

node First_Object

      arc 7: VIR NP
            action ADDR complements "dir_object"
            action ADDR arguments <BUILDQ (dir_object: +) NP>
            action SETR dirobj "no_prep"
```

```
        transition To More_objects

    arc 8: PUSH NP
        action ADDR complements "dir_object"
        action ADDR arguments <BUILDQ (dir_object: *)>
        action SETR dirobj "prep"
        transition TO More_objects

node More_Objects

    arc 9a: PUSH NP
        test <GETR dirobj> equals "no_prep"
        action ADDR complements "indir_object"
        action ADDR arguments <BUILDQ (indir_object: *)>
        transition TO More_objects

    arc 9b: PUSH PP
        test <GETR dirobj> equal "prep"
        test if <GETR preposition> equals "to" then
        action
            {ADDR complements "indir_object"
            ADDR arguments <BUILDQ (indir_object: *)>}
        test else
        action
            {ADDR complements <BUILDQ prep_object + preposition>
            ADDR arguments <BUILDQ (prep_object: + *) preposition>}
        transition TO More_Objects

    arc 10: WRD ?
        test <GETR illocution> equals "question"
        transition TO Valence

    arc 11: WRD .
        test <GETR illocution> equals "assertion"
        transition TO Valence

node Valence

    arc 12: JUMP
        test <GETF valence> includes <GETR complements>
        transition TO End_sentence

node End_sentence

    arc 13: POP
        BUILDQ (illocution: + (predicate: + + ))
        illocution predicate arguments

node NP
    arc 14: JUMP
        transition TO Center

    arc 15: CAT det
        action SETR determiner <BUILDQ (determiner: *)>
        transition TO Center

    arc 16: CAT pron
        test <GETR number> is empty or equals <GETF number>
        test <GETR person> is empty or equals <GETF person>
        action SETR number <GETF number>
        action SETR person <GETF person>
        action SETR structure <GETF *>
        transition TO End_NP
```

```
node Center

      arc 17: CAT adje
            action ADDR attributes <BUILDQ: (attrib: *)>
            transition TO Center

      arc 18: CAT noun
            test <GETR number> is empty or equals <GETF number>
            test <GETR person> is empty or equals "3rd"
            action SETR number <GETF number>
            action SETR person "3rd"
            action SETR head <GETF *>
            transition TO Attributes

node Attributes

      arc 19: PUSH PP
            test GETF lex is not "to"
            action ADDR attributes <BUILDQ: (attrib: *)>
            transition TO Attributes

      arc 20: JUMP
            action SETR structure <BUILDQ + + + head determiner
                  attributes>
            transition TO End_NP

node End_NP

      arc 21: POP structure
            action LIFTR number
            action LIFTR person

node PP

      arc 22: CAT prep
            action SETR preposition <GETF *>
            transition TO Prep_NP

node Prep_NP

      arc 23: PUSH NP
            action SETR structure <BUILDQ ( * )>
            transition TO End_PP

node End_PP

      arc 24: POP structure
            action LIFTR preposition
```

LEXICON:

sleep	lex[sleep] cat[verb] form[infinitive] valence[subject]
sleep	lex[sleep] cat[verb] form[finite] person[1st, 2nd] number[singular]valence[subject]
sleeps	lex[sleep] cat[verb] form[finite] person[3rd] number[singular] valence[subject]
sleep	lex[sleep] cat[verb] form[finite] person[1st, 2nd, 3rd] number[plural] valence[subject]
feed	lex[feed] cat[verb] form[infinitive] valence[subject, direct_object, indirect_object]
feed	lex[feed] cat[verb] form[finite] person[1st, 2nd] number[singular] valence[subject, direct_object, indirect_object]
feeds	lex[feed] cat[verb] form[finite] person[3rd] number[singular] valence[subject, direct_object, indirect_object]
feed	lex[feed] cat[verb] form[finite] person[1st, 2nd, 3rd] number[plural] valence[subject, direct_object, indirect_object]
do	lex[do] cat[verb] form[infinitive]
do	lex[do] cat[verb] form[finite] person[1st, 2nd] number[singular]
does	lex[do] cat[verb] form[finite] person[3rd] number[singular]
do	lex[do] cat[verb] form[finite] person[1st, 2nd, 3rd] number[plural]
Gudrun	lex[Gudrun] cat[noun] person[3rd] number[singular]
cat	lex[cat] cat[noun] person[3rd] number[singular]
fish	lex[fish] cat[noun] person[3rd] number[singular, plural]
I	lex[I] cat[pron] person[1st] number[singular]
you	lex[you] cat[pron] person[2nd] number[singular, plural]
what	lex[what] cat[W_pron]
the	lex[the] cat[det]
her	lex[her] cat[det]
silly	lex[silly] cat[adje]
to	lex[to] cat[prep]
with	lex[with] cat[prep]

INPUT:	what does Gudrun feed her cat ?

TRACE:

P:	Word:	Configurations:	Alter-natives:
1	what	*node* **Sentence**	
		arc 1: CAT W_pron	arc 2
		action HOLD NP[what]	
		transition TO Aux	
2	does	*node* **Aux**	
		arc 4: WRD do	
		action SETR number[singular]	
		action SETR person[3rd]	
		action SETR verb_form[infinitive]	
		action SETR illocution[question]	
		transition TO Subject	
3	Gudrun	*node* **Subject**	
		arc 5: PUSH NP	
		pre-action SENDR number[singular]	
		pre-action SENDR person[3rd]	
		node **NP**	
		arc 14: JUMP	arc 15
		transition TO Center	
		node **Center**	
		arc 17: CAT adje ***fails***	arc 18
		arc 18: CAT noun	
		test number[singular] equals number[singular]	
		test person[3rd] equals person[3rd]	
		action SETR number[singular]	
		action SETR person[3rd]	
		action SETR head[Gudrun]	
		transition TO Attributes	
4	feed	*node* **Attributes**	
		arc 19: PUSH PP	arc 20
		node **PP**	
		arc 22: CAT prep ***fails***	
		arc 20: JUMP	
		action SETR structure[Gudrun]	
		transition TO End_NP	
		node **End_NP**	
		arc 21: POP [Gudrun]	
		action LIFTR number[singular]	
		action LIFTR person[3rd]	
		action ADDR complements[subject]	
		action ADDR arguments[(subject: Gudrun)]	
		transition TO Verb	
		node **Verb**	
		arc 6: CAT verb	
		test verb_form[infinitive] equals form[infinitive]	
		action SETR predicate[feed]	
		transition TO First_Object	

5	her	*node* **First_Object**	
		arc 7: VIR NP	arc 8
		action ADDR complements[subject, dir_object]	
		action ADDR arguments[(subject: Gudrun)(dir_object: what)]	
		action SETR dirobj[no_prep]	
		transition TO More_objects	
		node **More_Objects**	
		arc 9a: PUSH NP	arc 9b
		test dirobj[no_prep] equals "no:prep"	
		node **NP**	
		arc 14: JUMP	arc 15
		transition TO Center	
		node **Center**	
		arc 17: CAT adje ***fails***	arc 18
		arc 18: CAT noun ***fails***	
		arc 15: CAT det	arc 16
		action SETR determiner[(determiner: her)]	
		transition TO Center	
6	cat	*node* **Center**	
		arc 17: CAT adje ***fails***	arc 18
		arc 18: CAT noun	
		test number is empty	
		test person is empty	
		action SETR number[singular]	
		action SETR person[3rd]	
		action SETR head[cat]	
		transition TO Attributes	
7	?	*node* **Attributes**	
		arc 19: PUSH PP	arc 20
		node **PP**	
		arc 22 CAT prep ***fails***	
		arc 20: JUMP	
		action SETR structure[cat (determiner:her)]	
		transition TO End_NP	
		node **End_NP**	
		arc 21: POP [cat (determiner: her)]	
		action LIFTR number[singular]	
		action LIFTR person[3rd]	
		action complements [subject, dir_object, indir_object]	
		action arguments[(subject: Gudrun)(dir_object: what) (indir_object: cat (determiner: her))]	
		transition TO More_Objects	
		node **More_Objects**	
		arc 9b: PUSH PP	arc 10
		node **PP**	
		arc 22: CAT prep fails	
		arc 10: WRD ?	arc 11
		test [question] equals "question"	

```
   transition TO Valence

node Valence
   arc 12: JUMP
   test [subject, dir_object, indir_object] includes
   complements[subject, dir_object, indir_object]
   transition TO End_sentence

node End_sentence
   arc 13: POP [(illocution: question (predicate: feed
   (subject: Gudrun)(dir_object: what)
   (indir_object: cat (determiner: her))]
```

RESULT

```
(illocution: question
      (predicate: feed
           (subject: Gudrun)
           (dir_object: what)
           (indir_object: cat
                  (determiner: her))]
```

One parse is found. However, the program will still check numerous alternative arcs.

Evaluation

(1) **Efficiency.** An augmented transition network is tailored to the syntactic structure of a particular language. Therefore, it can be very efficient. Blind backtracking can be avoided by means of registers which are maintained in order to locate the reasons for a failure. However, this requires additional effort from the programmer since he must anticipate the dead-ends into which the system might run. There is also a "wait-and-see"-style of ATN-programming that simply collects any constituents at first and then interprets them when the end of the network is reached.

(2) **Coverage.** ATNs are capable to deal with any phenomenon since they allow, in principle, of any tests and any actions. Among our prototypes, PT-8 is the only one with a theoretical chance to cope with all of the phenomena of a natural language, as for example discontinuous constituents, coordination, ellipsis etc. The problem is to reduce the Turing power of ATNs by limiting the variety of tests and actions to the minimum needed. If this is not done, it would be impossible to predict the behavior of the parser.

(3) **Drawing up lingware.** With regard to perspicuity, PT-8 is the worst among our prototypical parsers. ATNs are the exact opposite of a modular and lexicalistic style of descriptions. An ATN for a larger fragment of natural language is a huge and complex system. It is difficult to debug a grammar that is formulated as a program and trouble with side effects must be expected. The usual means that ATNs lend for calculating agreement, namely SENDR and LIFTR actions operating on individual attributes, are awkward. The ATN formalism should be augmented by a UNIFY action that operates on complex categories as a whole. ATNs can be very efficient and cover a lot of phenomena when they are completed, but it is awkward to write them. Unfortunately, for each new language a new ATN must be written from scratch.

PT-9. Chart Parsing according to the slot-and-filler principle

Illustrates:

(1) Connection between grammar and parser: Interpreting parser; separation of grammar and parser.
(2) Linguistic structure assigned: Dependency descriptions.
(3) Grammar specification format: Templates associated with lexical items; lexicalized grammar.
(4) Recognition strategy: Completion oriented analysis with slots and fillers; top-down creation of slots, bottom-up insertion of fillers.
(5) Processing the input: Left-to-right, one path.
(6) Treatment of alternatives: No backtracking; well-formed substring table.
(7) Control of results: All intermediate results accessible; parallel processes.

References: *Hellwig* 1980, 1988, 1989: 425, 1993, *McCord* 1980

Prerequisites:

(P-1) A **morpho-syntactic lexicon** relating each basic input segment (e.g. each word form) to a representation of a basic node that will become part of a dependency tree in the course of the analysis. As a rule, the representation includes the indication of a lexeme and a set of morpho-syntactic categories. The latter describe the part of speech and the inherent grammatical features of the corresponding segment.

(P-2) A **set of templates** each describing one potential syntagmatic relationship between a lexical item and a set of constructions which combine with the item in the specific role (e.g. the subjects, objects etc. that go with a verb). Templates define syntagmatic relationships in form of subtrees of dependency trees. Each template consists of a name and a minimal tree with two nodes: a node for a governing word and a so-called slot which substitutes for a potential complement. The governing node contains morpho-syntactic constraints regarding the governing word. The slot contains the role and the morpho-syntactic and semantic requirement for the dependent.

(P-3) A **syntagmatic lexicon** with valency frames each relating a lexical item to a set of templates. This information is equivalent to strict subcategorization in a Chomsky grammar.

(P-4) A **declaration of the morpho-syntactic categories** used in the lexicon and in the templates defining the attributes and values for each category and the way in which correspondences between values are computed during the slot filling process (i.e. the type of unification). Typical devices for processing values are the construction of the intersection or union of features and the verification of positional order.

(P-5) A **working space** for storing the immediate results. Any information collected about a smaller or larger segment of the input is stored in an object called "bulletin". Among other points, a bulletin comprises the lefts and right margins of the corresponding segment in the input, a reference to the previous bulletin that provided the head and slots and a reference to the previous bulletin that provided the filler, a dependency tree representing the current description of the segment including subordinated fillers as well as open slots. A special mark signals the capability of a bulletin for discontinuous attachment of fillers.

(P-6) **A chart**, i.e. a table in which each bulletin is registered

Algorithm:

The program consists of the following processes which are partially executed in parallel and communicate with each other via message queues.

(A-1) **Scanner.** Read the input text until the end of file. Divide the input text into lexical elements according to the morpho-syntactic lexicon (multi-word lexical entries are possible). Extract the lexemes and the morpho-syntactic categories for each lexical item. Allocate a separate bulletin for each reading associated with each particular segment and register it in the chart. Save all available information in the bulletin and send the bulletin to the Predictor via the predictor-queue.

(A-2) **Predictor.** As long as the predictor-queue is not empty, choose a bulletin and extract its lexeme. Look up the valency frames for the lexeme in the lexicon. Inspect all the templates that are mentioned in the frame, compare the head constraints in the template with the information collected for the word in the bulletin and copy all applicable slots into the bulletin. Send the augmented bulletin to the Selector via the selector-queue.

(A-3) **Selector.** As long as the selector-queue is not empty, choose a bulletin. Retrieve all other bulletins (via the chart) whose segments are promising candidates for mutual combination with the given segment, as head or as filler. Send the resulting pairs of bulletins to the Completer via the completor-queue. Note that the selector-queue 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.

(A-4) **Completer.** As long as the completer-queue is not empty, choose a pair of bulletins and try to combine them according to the following slot-filling device. Inspect the dependency description of both trees for slots. If a slot is found then check whether the tree in the other bulletin meets the specified filler requirements. If there are open obligatory slots in the filler tree then discard this candidate right away, except for discontinuous slots which don't need to be filled yet. Apply the unification method appropriate to each attribute while comparing slot and filler and while checking the agreement of filler and head. If a filler fits into a slot and if it agrees with the head then form a new tree in which the filler is inserted into the slot. Remove all open slots from the filler except those which may be filled by discontinuous constituents. Remove all slots from the head that are alternatives to the slot just filled. Allocate a new bulletin and save the new tree as well as the appropriate information in the control portion of the bulletin. (If the segment on the right hand-side was the filler then block the resulting bulletin for further left hand-side completion. This is necessary to avoid double results due to a varying sequence of attachments.) Submit the new bulletin to the Assess-Result function.

(A-5) **Assess-Result.** Check whether the current bulletin contains an utterance marker and whether the corresponding segment spans the whole input since the last utterance has been outputted. If this is the case then output the dependency description of the current bulletin and erase the bulletin and all its antecessors from the working space. Otherwise pass the bulletin to the Selector via the selector-queue.

Example

DECLARATION OF MORPHO-SYNTACTIC CATEGORIES

Main Categories:		
verb, noun, wh_pron, dete, adje, particle, prep		
Grammatical features:		
Attributes:	Values:	Unification:
form	finite, infinitive, past_part	compute intersection
person	1st, 2nd, 3rd	compute intersection
number	singular, plural	compute intersection
case	subject object	compute intersection
mode	quest assert	compute intersection
utterance	+ -	compute intersection
sent_position	1 – 7	verify positional order compute union
np_position	1 – 4	verify positional order compute union
pp_position	1 – 2	verify positional order compute union
discont	left, right	verify discontinuity verify positional order

MORPHO-SYNTACTIC LEXICON
(similar to the lexicon of PT-8 without the valence feature)

sleep	(lex[*sleep*] cat[verb] form[infinitive]);
sleep	(lex[*sleep*] cat[verb] form[finite] person[1st, 2nd] number[singular]);
sleeps	(lex[*sleep*] cat[verb] form[finite] person[3rd] number[singular]);
sleep	(lex[*sleep*] cat[verb] form[finite] number[plural]);

feed	(lex[*feed*] cat[verb] form[infinitive]);
feed	(lex[*feed*] cat[verb] form[finite] person[1st, 2nd] number[singular]);
feeds	(lex[*feed*] cat[verb] form[finite] person[3rd] number[singular]);
supply	(lex[*supply*] cat[verb] form[infinitive]);
supplies	(lex[*supply*] cat[verb] form[finite] person[3rd] number[singular]);
fed	(lex[*feed*] cat[verb] form[past_part]);
do	(lex[*do*] cat[verb] form[infinitive]);
do	(lex[*do*] cat[verb] form[finite] person[1st, 2nd] number[singular]);
does	(lex[*do*] cat[verb] form[finite] person[3rd] number[singular]);
do	(lex[*do*] cat[verb] form[finite] number[plural]);
did	(lex[*do*] cat[verb] form[finite] person[3rd] number[singular]);
has	(lex[*have*] cat[verb] form[finite] person[3rd] number[singular]);
Gudrun	(lex[*Gudrun*] cat[noun] person[3rd] number[singular]);
cat	(lex[*cat*] cat[noun] person[3rd] number[singular]);
cats	(lex[*cat*] cat[noun] person[3rd] number[plural]);
fish	(lex[*fish*] cat[noun] person[3rd]);
I	(lex[*I*] cat[pron] person[1st] number[singular] case[subject]);
me	(lex[*I*] cat[pron] case[object]);
you	(lex[*you*] cat[pron] person[2nd]);
he	(lex[*he*] cat[pron] person[3rd] number[singular] case[subject]);
him	(lex[*he*] cat[pron] case[object]);
what	(lex[*what*] cat[wh_pron] person[3rd] number[singular] mode[quest,C]);
who	(lex[*who*] cat[wh_pron] person[3rd] number[singular] case[subject] mode[quest,C]);
whom	(lex[*who*] cat[wh_pron] case[object] mode[quest,C]);
the	(lex[*the*] cat[dete]);
a	(lex[*a*] cat[dete] number[singular]);
all	(lex[*all*] cat[dete] number[plural]);
her	(lex[*her*] cat[dete]);
silly	(lex[*silly*] cat[adje]);
to	(lex[*to*] cat[prep]);
with	(lex[*with*] *cat*[prep]);
?	(lex[*question*'] cat[particle] utterance[+]);
.	(lex[*assertion*'] cat[particle] utterance[+]);

TEMPLATES

```
(template[+question]
(role[ILLOCUTION] cat[particle] sent_position[7]
          (< slot[oblig] role[PREDICATE] cat[verb] form[finite] mode[quest]
          sent_position[2,C])));

(template[+assertion]
(role[ILLOCUTION] cat[particle] sent_position[7]
          (< slot[oblig] role[PREDICATE] cat[verb] form[finite] mode[assert]
          sent_position[4,C])));

(template[+aux_subject]
(cat[verb] form[finite] sent_position[2]
          (> slot[oblig] role[SUBJECT] cat[noun] mode[quest,C]) number[C]
          person[C] sent_position[3,C])));

(template[+aux_subject]
(cat[verb] form[finite] sent_position[2]
          (> slot[oblig] role[SUBJECT] cat[pron] case[subject] mode[quest,C])
          number[C] person[C] sent_position[3,C])));

(template[+infinitiv ]
(cat[verb] form[finite] sent_position[2]
          (> slot[oblig, nuleus] role[PRED_COMPLEMENT] cat[verb]
          form[infinitive] mode[C] sent_position[4,C])));

(template[+participle ]
(cat[verb] form[finite] sent_position[2]
          (> slot[oblig, nucleus] role[PRED_COMPLEMENT] cat[verb]
          form[past_part] mode[C] sent_position[4,C)));

(template[+subject]
(cat[verb] form[finite] sent_position[4]
          (< slot[oblig] role[SUBJECT] cat[noun] mode[assert,C]) number[C]
          person[C] sent_position[3,C])));

(template[+subject]
(cat[verb] form[finite] sent_position[4]
          (< slot[oblig] role[SUBJECT] cat[pron] case[subject]
          mode[assert,C]) number[C] person[C] sent_position[3,C])));

(template[+subject]
(cat[verb] form[finite] sent_position[4]
          (< slot[oblig] role[SUBJECT] cat[wh_pron] mode[quest,C] person[C]
          number[C] sent_position[1,C])));

(template[+dir_object]
(cat[verb] sent_position[4]
          (> slot[optional] role[DIR_OBJECT] cat[noun]
          sent_position[5,6,C])));

(template[+dir_object]
(cat[verb] sent_position[4]
          (> slot[optional] role[DIR_OBJECT] cat[pron] case[object]
          sent_position[5,C])));

(template[+dir_object]
(cat[verb] form[infinitive] sent_position[4]
          (< slot[optional] role[DIR_OBJECT] cat[wh_pron] mode[quest,C]
          case[object] sent_position[1,C] discont[left])));
```

```
(template[+indir_object]
(cat[verb] sent_position[4]
           (> slot[optional] role[INDIR_OBJECT] cat[noun]
           sent_position[5,C])));

(template[+indir_object]
(cat[verb] sent_position[4]
           (> slot[optional] role[INDIR_OBJECT] lex[to] cat[prep]
           sent_position[6,C])));

(template[+indir_object]
(cat[verb] form[infinitive] sent_position[4]
           (< slot[optional] role[INDIR_OBJECT] lex[to] cat[prep]
           mode[quest,C] sent_position[1,C] discont[left])));

(template[+count]
(cat[noun] number[singular] np_position[3]
           (< slot[oblig] role[DETERMINER] cat[dete] number[C]
           np_position[1,C])));

(template[+count]
(cat[noun] number[plural] np_position[3]
           (< slot[optional] role[DETERMINER] cat[dete] number[C]
           np_position[1,C])));

(template[+mass]
(cat[noun] number[singular] np_position[3]
           (< slot[optional] role[DETERMINER] cat[dete] np_position[1,C])));

(template[+attribute]
(cat[noun] np_position[3]
           (< slot[optional, sequence] role[ATTRIBUTE] cat[adje]
           np_position[2,C])));

(template[+prep_attribute]
(cat[noun] np_position[3]
           (> slot[optional] role[PREP_ATTRIBUTE] cat[prep]
           np_position[4,C])));

(template[+prep_phrase]
(cat[prep] pp_position[1]
           (> slot[oblig] role[PREP_COMPL] cat[noun] pp_position[2,C])));

(template[+prep_phrase]
(cat[prep] pp_position[1]
           (> slot[oblig] role[PREP_COMPL] cat[pron] case[object]
           pp_position[2,C])));
```

Legend: Template names are printed bold. Grammatical roles are in upper case. If a filler is expected on the left of the head the slot is marked by "<", if on the right the slot is marked by ">". Slots can be obligatory or optional. The value "sequence" allows for several complements of the same kind, e.g. several adjectives in front of a noun. The value "nucleus" is used for slots that form a special union with the head like the auxiliary with the main verb. The value 'C' in a slot triggers the unification of values of the corresponding attributes in the filler and the head term.

VALENCY FRAMES
(equivalent to lexical subcategorization in generative grammar)

Word		Templates
sleep	->	+subject
feed	->	+subject, +direct_object, +indirect_object
do	->	+aux_subject +infinitiv
have	->	+aux_subject +participle
cat	->	+count, +attribute
fish	->	+count, +attribute
fish	->	+mass, +attribute
to	->	+prep_phrase
with	->	+prep_phrase
question'	->	+question
assertion'	->	+assertion

How a bulletin is composed:

1 MORPHO-SYNTACTIC LEXICON

does	(lex[*do*] cat[verb] form[finite] person[3rd] number[singular]);

2 SUBCATEGORIZATION (VALENCY FRAME)

```
do          ->          +aux_subject, +infinitiv
```

3 TEMPLATES

```
(template[+aux_subject]
(cat[verb] form[finite] sent_position[2]
             (> slot[oblig] role[SUBJECT] cat[noun] mode[quest,C]) number[C]
             person[C] sent_position[3,C])));

(template[+aux_subject]
(cat[verb] form[finite] sent_position[2]
             (< slot[oblig] role[SUBJECT] cat[pron] case[subjective]
             mode[quest,C]) number[C] person[C] sent_position[1,C])));

(template[+infinitiv]
(cat[verb] form[finite] sent_position[2]
             (> slot[oblig, nucleus] role[PRED_COMPLEMENT] cat[verb]
             form[infinitive] mode[C] sent_position[4,C])));
```

4 BULLETIN

(2)	**does**	1	5	-	-	N

```
(lex[do] cat[verb] form[finite] person[3rd] number[singular] sent_position[2]
         {(> slot[oblig] role[SUBJECT] cat[noun] mode[quest,C]) number[C] person[C]
         sent_position[3,C]) |
          (< slot[oblig] role[SUBJECT] cat[pron] case[subjective] mode[quest,C])
         number[C] person[C] sent_position[1,C]) }

         (> slot[oblig, nucleus] role[PRED_COMPLEMENT] cat[verb] form[infinitive]
         mode[C] sent_position[4,C]));
```

INPUT

Input:	what	does	Gudrun	feed	her	cat
Position:	1	2	3	4	5	6

WORKING AREA (BULLETINS)

A:	Segment:	Left margin:	Right margin:	Head bulletin:	Filler bulletin:	Disconti-nuous slot?
(1)	**what**	1	1	-	-	N

(lex[***what***] cat[wh_pron] person[3rd] number[singular] mode[quest,C]);

A:	Segment:	Left margin:	Right margin:	Head bulletin:	Filler bulletin:	Disconti-nuous slot?
(2)	**does**	2	2	-	-	N

(lex[***do***] cat[verb] form[finite] person[3rd] number[singular] sent_position[2]
{(> **slot**[oblig] role[SUBJECT] cat[noun] mode[quest,C]) number[C] person[C] sent_position[3,C])|
(> **slot**[oblig] role[SUBJECT] cat[pron] case[subject] mode[quest,C]) number[C] person[C]) sent_position[3,C]) }
(> **slot**[oblig, nuleus] role[PRED_COMPLEMENT] cat[verb] form[infinitive] mode[C] sent_position[4,C]));

A:	Segment:	Left margin:	Right margin:	Head bulletin:	Filler bulletin:	Disconti-nuous slot?
(3)	**Gudrun**	3	3	-	-	N

(lex[***Gudrun***] cat[noun] person[3rd] number[singular]);

A:	Segment:	Left margin:	Right margin:	Head bulletin:	Filler bulletin:	Disconti-nuous slot?
(4)	**does Gudrun**	2	3	(2)	(3)	N

(lex[***do***] cat[verb] form[finite] person[3rd] number[singular] mode[quest] sent_position[2,3]
(> role[**SUBJECT**] lex[***Gudrun***] cat[noun] mode[quest,C]) number[singular,C] person[3rd,C] sent_position[3,C])
(> **slot**[oblig, nuleus] role[PRED_COMPLEMENT] cat[verb] form[infinitive] mode[C] sent_position[4,C]));

A:	Segment:	Left margin:	Right margin:	Head bulletin:	Filler bulletin:	Disconti-nuous slot?
(5)	**feed**	4	4	-	-	N

(lex[***feed***] cat[verb] form[infinitive] sent_position[4]
{(> **slot**[optional] role[DIR_OBJECT] cat[noun] sent_position[5,6,C]) |
(> **slot**[optional] role[DIR_OBJECT] cat[pron] case[object] sent_position[5,C]) |
(< **slot**[optional] role[DIR_OBJECT] cat[wh_pron] mode[quest,C] case[object] sent_position[1,C] discont[left])}
{(> **slot**[optional] role[INDIR_OBJECT] cat[noun] sent_position[5,C]) |
(> **slot**[optional] role[INDIR_OBJECT] lex[to] cat[prep] sent_position[6,C]) |
(< **slot**[optional] role[INDIR_OBJECT] lex[to] cat[prep] mode[quest,C] sent_position[1,C] discont[left])});

(6)	**feed**	4	4	-	-	N

```
(lex[feed] cat[verb] form[finite] person[1st, 2nd]  number[singular]
sent_position[4]
     (< slot[oblig] role[SUBJECT] cat[pron] case[subject] mode[assert,C])
     number[C] person[C] sent_position[3,C])
     {(> slot[optional] role[DIR_OBJECT] cat[noun] sent_position[5,6,C]) |
      (> slot[optional] role[DIR_OBJECT] cat[pron] case[object]
     sent_position[5,C]) }
     {(> slot[optional] role[INDIR_OBJECT] cat[noun] sent_position[5,C]) |
      (> slot[optional] role[INDIR_OBJECT] lex[to] cat[prep]
     sent_position[6,C])} );
```

(7)	**does Gudrun feed**	2	4	(4)	(5)	Y

```
(lex[do] cat[verb] form[finite] person[3rd] number[singular]) mode[quest]
sent_position[2,3,4]
     (> role[SUBJECT] lex[Gudrun] cat[noun] mode[quest,C]) number[C] person[C]
     sent_position[3,C])
     (> role[PRED_COMPLEMENT] lex[feed] cat[verb] form[infinitive]
     sent_position[4,C])
           (< slot[optional] role[DIR_OBJECT] cat[wh_pron] mode[quest,C]
           case[object] sent_position[1,C] discont[left])));
```

(8)	**what does Gudrun feed**	1	4	(7)	(1)	N

```
(lex[do] cat[verb] form[finite] person[3rd] number[singular] mode[quest]
sent_position[1,2,3,4]
      (> role[SUBJECT] lex[Gudrun] cat[noun] mode[quest,C]) number[C] person[C]
      sent_position[3,C])
      (> role[PRED_COMPLEMENT] lex[feed] cat[verb] form[infinitive]
      sent_position[4,C]
           (< role[DIR_OBJECT] lex[what] cat[wh_pron] mode[quest,C]
           case[object] sent_position[1,C] discont[left])));
```

(9)	**her**	5	5	-	-	N

```
(lex[her] cat[dete]);
```

(10)	**cat**	6	6	-	-	N

```
(lex[cat] cat[noun] person[3rd] number[singular] np_position[3]
      (< slot[oblig] role[DETERMINER] cat[dete] number[C] np_position[1,C])
      (< slot[optional, sequence] role[ATTRIBUTE] cat[adje] np_position[2,C])
      (> slot[optional] role[PREP_ATTRIBUTE] cat[prep] np_position[4,C]));
```

(11)	**her cat**	5	6	(10)	(9)	N

```
(lex[cat] cat[noun] person[3rd] number[singular] np_position[1,3]
      (role[DETERMINER] lex[her] cat[dete] number[C] np_position[1,C])
      (> slot[optional] role[PREP_ATTRIBUTE] cat[prep] np_position[4,C]));
```

(12)	**feed her cat**	4	6	(5)	(11)	Y

```
(lex[feed] cat[verb] form[infinitive] sent_position[4,5]
      {(< slot[optional] role[DIR_OBJECT] cat[wh_pron]  mode[quest,C]
      case[object] sent_position[1,C] discont[left]) |
      (> slot[optional] role[DIR_OBJECT] cat[noun] sent_position[6,C])}
      (> role[INDIR_OBJECT] lex[cat] cat[noun] person[3rd] number[singular]
      np_position[1,3] sent_position[5,C]
           (role[DETERMINER] lex[her] cat[dete] number[C] np_position[1,C])));
```

(13)	**feed her cat**	4	6	(6)	(11)	N

```
(lex[feed] cat[verb] form[finite] person[1st, 2nd]  number[singular]
sent_position[4,5]
      (< slot[oblig] role[SUBJECT] cat[pron] case[subject] mode[assert,C])
      number[C] person[C] sent_position[3,C])
      (> slot[optional] role[DIR_OBJECT] cat[noun] sent_position[6,C])
      (> role[INDIR_OBJECT] lex[cat] noun person[3rd] number[singular]
      np_position[1,3] sent_position[5,C]
           (role[DETERMINER] lex[her] cat[dete] number[C] np_position[1,C])));
```

(14)	**does Gudrun feed her cat**	2	6	(4)	(12)	Y

```
(lex[do] cat[verb] form[finite] person[3rd] number[singular] mode[quest]
sent_position[2,3,4,5]
      (> role[SUBJECT] lex[Gudrun] cat[noun] mode[quest,C]) number[singular,C]
      person[3rd,C] sent_position[3,C])
      (> role[PRED_COMPLEMENT] lex[feed] verb form[infinitive]
      sent_position[4,6,C]
           (< slot[optional] role[DIR_OBJECT] cat[wh_pron] mode[quest,C]
           case[object] sent_position[1,C] discont[left])
           (> role[INDIR_OBJECT] lex[cat] cat[noun] person[3rd] number[singular]
           np_position[1,3] sent_position[5,C]
                  (< role[DETERMINER] lex[her] cat[dete] number[C]
                  np_position[1,C])));
```

(15)	**what does Gudrun feed her cat**	1	6	(14)	(1)	N

```
(lex[do] cat[verb] form[finite] person[3rd] number[singular] mode[quest]
sent_position[1,2,3,4,5]
      (> role[SUBJECT] lex[Gudrun] cat[noun] mode[quest,C]) number[singular,C]
      person[3rd,C] sent_position[3,C])
      (> role[PRED_COMPLEMENT] lex[feed] verb form[infinitive]
      sent_position[1,4,6,C]
           (< role[DIR_OBJECT] lex[what] cat[wh_pron] mode[quest,C]
           case[object] sent_position[1,C] discont[left])
           (> role[INDIR_OBJECT] lex[cat] cat[noun] person[3rd] number[singular]
           np_position[1,3] sent_position[5,C]
                  (< role[DETERMINER] lex[her] cat[dete] number[C]
                  np_position[1,C])));
```

(16)	**?**	7	7	-	-	N

```
(role[ILLOCUTION] lex[question'] cat[particle] utterance[+] sent_position[7]
      (< slot[oblig] role[PREDICATE] cat[verb] form[finite] mode[quest]
      sent_position[2,C]));
```

(17)	what does Gudrun feed her cat ?	1	7	(16)	(15)	N

```
(role[ILLOCUTION] lex[question'] cat[particle] utterance[+]sent_position[7]
      (< role[PREDICATE] lex[do] cat[verb] form[finite] person[3rd]
      number[singular] mode[quest] sent_position[1,2,3,4,5]
           (> role[SUBJECT] lex[Gudrun] cat[noun] mode[quest,C])
           number[singular,C] person[3rd,C] sent_position[3,C])
           (> role[PRED_COMPLEMENT] lex[feed] verb form[infinitive]
           sent_position[1,4,6,C]
                  (< role[DIR_OBJECT] lex[what] cat[wh_pron] mode[quest,C]
                  case[object] sent_position[1,C] discont[left])
                  (> role[INDIR_OBJECT] lex[cat] cat[noun] person[3rd]
                  number[singular] np_position[1,3] sent_position[5,C]
                         (< role[DETERMINER] lex[her] cat[dete] number[C]
                         np_position[1,C])));
```

RESULT (without morho-syntactic categories):

```
       (ILLOCUTION: question'
            (PREDICATE: do
                 (SUBJECT: Gudrun)
                 (PRED_COMPLEMENT: feed
                        (DIR_OBJECT: what)
                        (INDIR_OBJECT: cat
                               (DETERMINER: her))));
```

Evaluation

(1) **Efficiency.** The slot-and-filler parser starts with the terminal elements in the input; these elements are associated with slots for other elements in the input. So, PT-9 is data-driven and expectation-driven at the same time, or more precisely, even its expectations are data-driven. Therefore, the slots of PT-9 are much more concrete than the expectations derived from the abstract sentence structure in rule-based parsers. Although there is a considerable amount of local over-generation, the potential relevance of each intermediate result is high. As opposed to top-down parsers, the number of rules/slots that must be checked is independent of the size of the grammar; the number of steps of the parser are completely determined by the valency of the encountered words.

(2) **Coverage.** The grammar of PT-9 is an instance of an attribute grammar (*Knuth* 1968, *Van Wijngaarden* 1969, *Pagan* 1981). Its power depends on the kind of attributes used. The position attributes and the discont-attribute exceed context-freeness. Other attributes may be added in the course of dealing with additional phenomena of natural languages. Of course, one should keep the portion of context-sensitiveness to the minimum. In a way, PT-9 comes from the opposite direction as compared to ATN. Instead of forcing the power of a Turing machine down to a desirable level, the context-free level is gradually augmented by a desirable extension into the area of context-sensitive power.

(3) **Drawing up lingware.** The extremely modular and lexicalized approch of PT-9 is the contrary of huge networks like ATNs (PT-8) and tight systems like table-based shift-reduce parsers (PT-6). It is easy to describe the valency of words and not many side effects are to be expected from the addition of more word descriptions. Note that complex categories and the unification mechanism are prerequisites for the direct processing of dependency trees as shown in PT-9. Multiple labeling of the nodes in the dependency tree is the key to achieve a description without non-terminal nodes. Removing non-terminal nodes is, however, very advantageous in order to simplify the description.

3 Comparative evaluation of the parser prototypes

A parser should meet the following requirements:

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

In the sequel we discuss the pros and cons of the various parsing technologies with respect to these goals.

3.1 Efficiency

Parsers do not differ so much in the space they require. In addition, space requirements are not very critical any more. Therefore we concentrate on the time aspect here. There are two main problems that have an impact on processing time: the control of alternative steps and the overgeneration of intermediate results.

3.1.1 Control of alternatives

A parser runs deterministically if it is always clear what will be its next state, given the sequence of previous states and the current input. Humans seem to process language deterministically most of the time; at least no hesitation in understanding the next word is observable (for consequences see *Marcus* 1980). On the other hand a parser for natural language cannot be totally deterministic, because ambiguity is a genuine property of natural language. A speaker can utter an ambiguous sentence on purpose. That is why a deterministic FTN parser (PT-7) can never cope with the whole range of syntactic structures.

Schematic backtracking as well as parallel processing of all alternatives are the worst choices in order to deal with syntactic variation. Under this aspect PT-1 and PT-2 should be ruled out. Both prototypes require 17 rule applications in order to parse the example. The same rule is often applied several times to the same substring because somewhere else in the previous context an alternative path has been followed, i.e. the same work is done over and over again. Recursive transition networks adhere to the same principles and are, therefore, no better. (By the way, the same is true for the built-in resolution mechanism of PROLOG.)

Augmented transition networks (PT-8) can avoid blind backtracking by means of the registers which can be inspected in order to locate the reasons for a failure. However, this requires additional effort from the programmer since he must anticipate the dead-ends into which the system may run. There is also a "wait-and-see"-style of ATN-programming that simply collects constituents at first and then interprets them at the exit of the network. Thus, ATNs can be efficient once they are constructed, but it is awkward to write them.

Table-controlled shift-reduce parsers (PT-6) are virtually similar to transition networks. They are impressively efficient because looking ahead across the border to the next constituent is integrated into the control table. PT-6 processes the example, that causes 10 times backtracking in PT-1, completely deterministically. This type of a parser was recently advocated again

(Tomita 1985). A slight disadvantage of this approach is the necessity to construct a new control table each time a change is made in the grammar.

The remaining possibility to deal with variations in an efficient manner is the use of a well-formed substring table or "chart". Any substring that has been analyzed is kept in the working area and may be used again in various different combinations with its context. The Earley parser (PT-4), the Cocke-Parser (PT-5), and the slot-filler parser (PT-9) are examples of this technique. The problem of backtracking does not arise at all in this approach. The conclusion is that, according to the state of the art, chart parsing is usually recommendable.

3.1.2 Overgeneration

"Overgeneration" is a label to cover a range of problems, "combinatorial explosion" is another. In principle, a grammar would be overgenerating if it allowed for strings that are not part of the language. Although it is not easy to draw up a grammar that assigns a structural description to all and only the well-formed strings of a language, this is not the problem here. What touches efficiency is the fact that in the course of filling the search space (compare figure 6 in section 1.7) substrings and structures may be generated that are superfluous or obsolete in the context. Since the context that is taken into account can only be increased gradually it is obvious that there are situations where alternative readings for parts of the input must be kept. Each alternative asks for further processing and, as a consequence, the number of intermediate results have impact on the efficiency of the parser.

In principle, there are two factors that influence the amount of overgeneration: the parser's strategy and the quality of the grammar.

With respect to the first factor, top-down and bottom-up strategies differ in their advantages and disadvantages. A top-down parser may generate expansions which 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. Instances of the first type are PT-1, PT-2, PT-4, PT-6, PT-8 instances of the latter type are PT-5 and PT-9. The advantages of top-down parsers result from the fact, that these parsers rely, in a sense, on expectations what comes next. This is especially true for PT-6 which includes looking one constituent ahead. 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. Overgeneration will possibly be minimized if a strategy is found that combines both principles in one way or the other.

If we leave aside ATNs that are capable of any tests and actions but are difficult to draw up, PT-4, PT-6 and PT-9 are on the short list of efficient parsers that attempt to combine top-down and bottom-up strategies. The Earley parser (PT-4) alternates between top-down rule expansion (expectation-driven) and bottom-up completion (data-driven). The table-controlled shift-reduce parser (PT-6) confines the top-down approach to the construction of the tables, while it proceeds bottom-up at run-time. The slot-filler parser (PT-9) starts with the elements which are in fact present (data-driven). These elements are associated with slots for other elements in the input (expectation driven). The slot-filling mechanism reconciles expectations and actual data.

The Earley chart parser (PT-4) would not generate intermediate results that are obsolete with respect to the left context. On the other hand it produces superfluous expansions, the number of which depends on the total number of grammar rules. The slot-filler parser (PT-9) produces intermediate segments that are not connected with the beginning of the sentence and, hence, may

be obsolete in the broader context. On the other hand PT-9 applies only rules (in the form of slots) that are relevant for the actual terminal elements. The number of slots that will be checked is independent of the size of the grammar. The table-controlled shift-reduce parser (PT-6) uses the left context, the actual terminal element, as well as the first element of the constituent to follow in order to rule out overgeneration. In this respect, PT-6 seems to be the most efficient parser.

The second factor with respect to the problem of overgeneration is the quality of the grammar. This factor can hardly be overestimated. A bad grammar is often the reason for poor performance of the parser. Slight changes in the lingware can have major effects to the worse as well as to the better. Thus, it is not enough to have a good parser. What is needed also, is a scrutinized tuning of constraints for a large variety of words. If such a sophisticated lingware is at hand the problem of spurious segments can hopefully be reduced to a minimum.

Of course, a parser can not perform better than a human to whom a string is presented in isolation. There may be syntactic overgeneration because disambiguation is impossible an purely syntactic grounds. In this case, the parser must produce as many parses as there are ambiguities. The task of resolving these ambiguities involves extra-syntactic resources, e.g. semantic relationships, discourse understanding, world knowledge. It is an open question whether these resources should intervene with the parsing process or follow it up. In any case, such additional devices must be worked out carefully. If one adds ad hoc restrictions, e.g. semantic features according to a given object domain, the system would not be re-usable in other environments. This would be a severe deficiency. The parser would then undergenerate in other applications, a behavior that is worse than overgeneration.

3.2 Coverage

It is a truism that a parser must be able to cope with the linguistic phenomena. As a first indication, the coverage of parsers can be correlated with Chomsky's hierarchy of grammars (*Chomsky* 1959):

Language	Grammar	Automaton	Network
type-3	regular grammar	finite-state automaton	FTN
type-2	context-free grammar	push-down store	RTN
type-1	context-sensitive grammar	linear restricted automaton	ATN
type-0	unrestricted production system	Turing machine	

Some parsers impose further restrictions on the grammar, for example, they do not allow left-recursive rules or no category deletion ("epsilon" rules). Or they accept a certain normal form only, for example, only binary rules.

PT-1 through PT-6 cover context-free grammars only, PT-7 is restricted to regular grammars. They are not suitable to deal with natural language phenomena as, for instance, discontinuous constituents or unbound dependencies that require greater power. This deficiency is a strong objection to the application of these parsers. Even PT-4 and PT-6, which are good choices for programming languages, will fail for many natural languages if broad coverage is the goal. In English, discontinuous left dislocations might still be tractable with a context-free device (e.g. Gazdar's slash categories), but this attempt will be awkward for languages with free word order,

e.g. German. Nevertheless, there are other phenomena in natural language, for instance coordination and ellipsis, which will eventually rule out the common parsing techniques developed in computer science.

ATNs (PT-8) are capable to deal with any phenomenon since they allow, in principle, for any tests and any actions. The problem with this approach is to reduce the power of the formalism by limiting the variety of tests and actions to the minimum needed. If this is not done it would be impossible to predict the behavior of the parser. The lexicalistic slot-and-filler parser (PT-9) seems to be promising with respect to extended coverage because there will be no syntactic phenomenon that is not reflected by relationships between words and phrases and thus does not fit into the basic framework. The provision for the extension of this parser beyond context-free structures is the possibility to define different ways in which attributes are unified. The positional attributes in the above example are instances for such an extension. Other types of attributes, i.e. other procedures for controlled unification, can be added when needed. This attempt to preserve flexibility for future extensions is similar to the principle open-endedness of ATNs. However, in contrast to ATNs, this possibility for augmentation is precisely located within the formalism. In a way, PT-9 comes from the opposite direction as compared to ATN. Instead of limiting the power of a Turing machine down to a desirably restricted level, the context-free level is gradually augmented by a desirable extension into the area of context-sensitive power.

3.3 Drawing up Lingware

The task of providing lingware to the parser is by far the most costly one. For each language, some 100 000 words must be categorized according to numerous syntactic relationships. This can be achieved only in a long process of research, testing and debugging. And because natural languages constantly change, updates of the grammar will be an important economic factor. If the costs for development and maintenance of a NLP system are to be minimized then the ease of writing and tuning the lingware must have highest priority. In the sequel we comment on three aspects: perspicuity, side-effects, and usability of external resources.

3.3.1 Perspicuity

Although the knowledge about an object domain, i.e. natural language, may be very complex the knowledge representation can vary with respect to perspicuity. The worst case among our prototypical parsers is an ATN (PT-8). The same is true for any parser that does not separate grammar and procedures. (Another case of this type is "Word expert parsing"). It is quite obvious that procedural representations are less perspicuous than declarative ones.

Grammars differ in the type of structure they assign to the input strings. The main choice is constituency structure versus dependency structure. Constituency structure is more complex and hence less perspicuous because it has to introduce a non-terminal node into the parse tree for each syntagmatic entity that is composed by a rule application. Some parsers, e.g. the Cocke-parser (PT-5) and ATNs (PT-8), allow for the construction of dependency trees although they process the input along constituency lines. However, this kind of two-fold thinking does not facilitate the writing of the grammar. A uniform way of constructing dependency trees right from the beginning is the slot-and-filler parser PT-9.

The use of complex categories, each comprising a set of attributes and values, as well as a mechanism to calculate the agreements between attributes of different elements (unification) is a must for all formalisms. Otherwise a combinatorial explosion cannot be avoided, especially in

the case of heavily inflected languages. (For instance, the rule S -> NP VP would have to be replaced by six rules if only person and number agreement between NP and VP is taken into account.) We have not worked out complex categorization and unification in the above examples, except for PT-9, but it may be assumed that the rules of PT-1 through PT-6 can be augmented by attributes and values similar to PT-9. The usual means that ATNs lend for calculating agreement, namely SENDR and LIFTR actions operating on individual attributes, are awkward. They should be augmented by a UNIFY action that operates on complex categories as a whole. Note that complex categories and unification is a prerequisite for the direct processing of dependency trees as shown in PT-9. Multiple labeling of the nodes in the dependency tree is the key to achieve a description without non-terminal nodes. Removing non-terminal nodes, however, is very advantageous in order to simplify the unification process, because the nodes that have to be compared for agreement are not mediated by an overhead of nodes that would only inherit the features in question.

Unification of categories requires the storage of detailed intermediate results. The consumption technique of some recognizers (PT-1, PT-2) is less favorable for this goal. The relationship between parser and grammar also has an impact on perspicuity. It is likely that a procedural representation (PT-8) is less transparent than a declarative one. Therefore, the separation of grammar and procedures is an important criterion.

3.3.2 Side effects

It is a well-known crux that the addition of more grammatical data may cause bad performance of the parser even in areas where it worked well before. This means that after any change in the lingware the whole system must be tested for undesired side effects. Therefore, it is highly desirable that additions have little impact on the rest of the grammar. The best way to reach this goal is to organize the grammatical data in small, modular portions, so that the range of the changes remain calculable.

The most difficult problems with respect to side effects must be expected if the software and the lingware of the parser are not separated as it is the case with ATNs (PT-8). Debugging a procedural parser is awkward because there can always be two quite distinct reasons for a bad performance: either the control structure of the parser as such is faulty or the grammar is not adequate with respect to the language phenomena. The measures which must be taken in the two cases are quite different: In the first case the programmer must locate the error and correct the source code, in the second case the linguist must study the language and alter his descriptions. As long as program and grammar intervene, the task can neither be divided among different people nor will the program ever be finished, since changes in the grammar will always be necessary. Hence, the choice must be a declarative grammar and an interpretative parser in order to minimize side effects by additional data.

Among declarative grammars we have the choice between sentence-oriented, rule-based grammars (PT-1 through PT-7) and word-oriented, lexicalized grammars (PT-9). In the first case, the potential range of structures that may be affected by an alteration is the sentence. In the second case the range is limited to a word and the structures which are potential dependents of the word. The latter is much smaller. As a consequence, we conclude that the lexicalized dependency parser PT-9 is the least vulnerable by side effects.

3.3.3 Usability of external resources

Given the fact that drawing up lingware is the most costly task in developing NLP systems, it is desirable to exploit external resources automatically. Examples are existing dictionaries as well as large data bases which may result from current activities of the European Community (e.g. Eurotra-7, MULTILEX etc.). Here again a lexicon-based system (e.g. PT-9) is advantageous because the information present in external resources is the most akin to the lingware of the system.

3.4 Suitability for the application

Naturally, the required quality of the parsing results depends on the application. For example, a grammar checker needs a very scrutinizing parser. It must not accept any ill-formed input. A parser used in a machine translation system should analyze the syntactic structure of sentences correctly. On the other hand, ill-formed input as well as wrong parses might be tolerated in information retrieval, as long as a sufficient amount of content is recognized. In many cases, it is reasonable to apply statistical methods (n-grams, Markov models etc.) in order to extract isolated phrases or to detect certain features of a text rather than include a full-fledged parser.

An interesting extension of the art of parsing is the ability to correct ill-formed input so that it fits a correct structural description. Three subtasks of error correction can be distinguished:

- the error must be located,
- the type of error must be diagnosed
- the error must be corrected.

The ability to locate the error depends partially on the strategy which the parser applies in order to fill the search space in the parse tree. Parsers that proceed from left to right and construct intermediate results that are linked to the beginning of the input (i.e. PT-2, PT-4, PT-6) will stop at the first deviant element or shortly there after. The latter is the case if there is accidentally a reading beyond the factual error. Parsers with schematic backtracking (PT-1, PT-8) can only locate an error if they store the information about the ultimate position that was successfully reached before they scroll back though former alternatives. Bottom-up chart parsers (PT-5, PT-9) face greater difficulties to locate errors. They must collect adjacent segments of decreasing size that were successfully accumulated in the chart and calculate the most probable positions where two adjacent segments should have been able to combine but failed to do so. The advantage of bottom-up parsers, however, is the fact that they alone have a potential to cope with multiple errors.

The critical property of a parser in the framework of grammar checking is the suitability for diagnosis and correction. The basis for human error diagnosis and correction is the redundancy of natural language which leaves clues for the right version even if part of the information is destroyed. Therefore one can assume in general that a parser is the more suitable to serve the goals of error correction the more information it has accumulated in the search space and, more importantly, that this information consists of expectations and data which can be reconciled in one way or the other.

As a consequence, only parsers that combine the top-down and the bottom-up principle are likely to be suitable for high-quality error correction. Pure top-down parsers (PT-1, PT-2, PT-7) are

probably insufficient because they produce expectations but, as soon as an error occurs, they do not reach the level of terminal elements that would provide the information about the factual situation. The Earley parser (PT-4) and the table-controlled shift-reduce parser, too, associate top-down and bottom-up information only up to that point in the input that could be reached successfully. Pure bottom-up parsers (PT-5) face the problem that they first have to acquire top-down information when they fail in order to reconcile the fragments they have built so far. This may not be infeasible since the source of top-down information, namely the production rules, are still at hand.

The dependency parser PT-9 has a chart structure similar to PT-5; however the necessary top-down information needs not to be collected by additional rule applications but is already present in form of open slots. The slots define precisely what the context of the given words should be like. The information what properties the actual fragments have is also available. By comparing the required features of the slots and the factual features of the existing segments the differences between both sets can be calculated. The least deviant cases are selected and their features are changed. Then the parser resumes its attempts to construct a coherent result. If it succeeds then a new string is generated according to the (partially changed) features of the complete parse tree and is offered as a proposal for correction. It seems that this method is superior to the other approaches.

4 Bibliography

Aho/Ullman 1972	Aho, A.V./Ullman, J.D: The Theory of Parsing, Translation, and Compiling. Englewood Cliffs: Prentice Hall 1972
Aho/Sethi/Ullman 1986	Aho, Alfred V./Sethi, Ravi/Ullman, Jeffrey D.: Compilers: Principles, techniques, and tools. Reading, Mass. 1986.
Aho/Ullman 1977	Aho, A.V./Ullman, J.D: Principles of Compiler Design, reading, Mass, 1977
Bates 1978	Bates, Madeleine: The theory and practice of augmented transition network grammars. In: Bolc [ed.] 1978 191-257.
Bobrow/Fraser 1969,	Bobrow, Daniel G./Fraser, E.: An augmented state transition network analysis procedure. In: Proceedings of the International Joint Conference on Artificial Intelligence, Washington 1969, Bedford, Mass. 1969, 557-567.
Bolc 1978	Bolc, Leonard [ed.]: Natural Language Communication with Computers. Lecture Notes in Computer Science, Vol. 63. Berlin - Heidelberg - Tokyo - New York 1978.
Chomsky 1957	Chomsky, Noam: Syntactic Structures. The Hague 1957.
Chomsky 1959	Chomsky, Noam: On certain formal properties of grammars. In: Information and Control 2, 1959, 137-167.
Chomsky 1965	Chomsky, Noam: Aspects of the Theory of Syntax. Cambridge, Mass. 1965
Covington 1994	Michael A. Covington Natural Language Processing for Prolog Programmers Englewood Cliffs (Prentice-Hall) 1994
Earley 1970	Earley, Jay C.: An efficient context-free parsing algorithm. In: Communications of the ACM 13 (2), 1970, 94-102.
Hays 1966	Hays, David G.: Parsing. In: David G. Hays (ed.): Readings in Automatic Language Processing. New York 1966
Hellwig 1974	Formal-desambiguierte Repräsentation. Vorüberlegungen zur maschinellen Bedeutungsanalyse auf der Grundlage der Valenzidee. (Dissertation Heidelberg 1974.) Stuttgart 1978.
Hellwig 1980	PLAIN - A Program System for Dependency Analysis and for Simulating Natural Language Inference. In: Leonard Bolc (Ed.): Representation and Processing of Natural Language. München, Wien, London 1980, S. 271 - 376.
Hellwig 1988	Chart Parsing According to the Slot and Filler Principle. In Proceedings of the 12th International Conference on Computational Linguistics (COLING 88). John von Neumann Society for Computing Sciences Budapest. Budapest 1988. Vol. I., S. 242-244.
Hellwig 1989	Parsing natürlicher Sprachen: Grundlagen, Parsing natürlicher Sprachen: Realisierungen. In: Computational Linguistics. Ein internationales Handbuch zur computerunterstützten Sprachforschung und ihrer Anwendung. Hrsg. von I. S. Bátori, W. Lenders, W. Putschke. Berlin: De Gruyter 1989. (Handbücher zur Sprach- und Kommunikationswissenschaft.) S. 348 - 431.
Hellwig 1993	Extended Dependency Unification Grammar. In: Eva Hajicova (ed.): Functional Description of Language. Faculty of Mathematics and Physics, Charles University, Prague 1993. S. 67-84.
Hopcroft/Ullman 1979	Hopcroft, John E./Ullman, Jeffrey D.: Introduction to Automata Theory, Languages and Computation. Reading, Mass. 1979.
Hudson 1980	Hudson, Richard A.: Constituency and dependency. In: Linguistics, Vol. 1, 1980, 179-198.
Hudson 1980	Hudson, Richard A.: Word grammar. Oxford 1984.

Kasami 1965	Kasami, T.: An efficient recognition and syntax analysis algorithm for context-free languagwes. Technical report, Air Force Cambridge Research Laboratory, Bedford, Mass. 1965
Kay 1977	Kay, Martin: Morphological and syntactic analysis. In: Zampolli [ed.] 1977, 131-234.
Knuth 1968	Knuth, Donald E.: Semantics of context-free languages. In: Mathematical systems theory 2, 1968, 127-145.
Kratzer et al. 1974	Kratzer, Angelika/Pause, Eberhard/Stechow, Arnim von: Einf • hrung in die Theorie und Anwendung der generativen Syntax. 2 Bde. Frankfurt/M. 1973-74.
Kuno/Oettinger 1963	Kuno, Susumo/Oettinger, Anthony G.: Multiple-path English Analyzer. In: Report No. NSF-8 Mathematical LInguistics and Automatic Translation.
Lehnert/Ringle 1982	Lehnert, Wendy G./Ringle, Martin H. [eds.]: Strategies for Natural Language Processing. Hillsdale, N.J. - London 1982.
Marcus 1980	Marcus, Mitchell P.: A theory of syntactic recognition for natural language. Cambridge, Mass. 1980.
McCord 1980,	McCord, M.C.: Slot grammars. In: AJCL 6, 1, 1980, 31-43.
Naumann/Langer 1994	S. Naumann, H. Langer: Parsing. Stuttgart: Teubner 1994.
Pagan 1981	Pagan, Frank G.: Formal specification of programming languages. Prentice Hall, N.J. 1981.
Pereira/Warren 1983	Pereira, Fernando C.N./Warren, David H.D.: Parsing as deduction. In: Proceedings of 21st Annual Meeting of the Association for Computational Linguistics, Boston, Mass. 1983, 137-144.
Shieber 1986	Shieber, Stuart M.: An introduction to unification-based approaches to grammar. University of Chicago Press, Chicago 1986
Small/Rieger 1982	Small, Steven/Rieger, C.: Parsing and Comprehending with Word Experts. In: Lehnert/Ringle [eds.] 1982, 89-147.
Starosta/Nomura 1986	Starosta, Stanley/Nomura, Hirosato: Lexicase Parsing: A Lexicon-driven Approach to Syntactic Analysis. In: COLING '86, Proceedings of the 11th International Conference on Computational Linguistics, Bonn 1986, 127-132.
Tesnière 1959	Tesnière, Lucien: Eléments de syntaxe structurale. Paris 1959.
Tomita 1986	Tomita, M. : Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems. Kluewer, Boston 1986
Van Wijngaarden 1969	Wijngaarden, A. van: Report on the algorithmic language ALGOL 68. In: Numerische Mathematik 14, 1969, 79-218.
Varile 1983	Varile, Giovanni Battista: Charts: A data structure for parsing. In: King [ed.] 1983, 73-87.
Winograd 1983	Winograd, Terry: Language as a Congnitive Process. Volume I: Syntax. Reading, Mass. 1983.
Woods 1969	Woods, William A.: Augmented Transition Networks for Natural Language Analysis. Report CS-1, Aiken Computer Laboratory, Harvard University. Cambridge, Mass. 1969.
Woods 1970	Woods, William A.: Transition Network Grammars for Natural Language Analysis. In: Communications of the Association for Computing Machinery 13, No. 10, 1970, 591-606.
Younger 1967	Younger, D.H.: Recognition of Context-free Languages in Time n^3. In: Inf. Control 10, 1967, 189-208.
Zampolli 1977	Zampolli, Antonio [ed.]: Linguistic Structures Processing. Amsterdam - New York - Oxford 1977.

5 Exercises

PT-1. Top-down parser with backtracking

Draw up the work table and the backtracking store for the sentence:

"Many foreign tourists see the pyramids"

PT-2. Top-down parser with parallel processing

Draw up the work table for the sentence:

"Many foreign tourists see the pyramids"

PT-3. Top-down predictive analyzer (reibach normal form grammar)

Draw up the work table for the sentence:

"Many foreign tourists see the pyramids"

PT-4. Top-down parser with divided productions

Draw up a work table for the sentence:

"Many tourists enjoy Egypt"

PT-5. Bottom-up parser with a well-formed substring table

Draw up a work table and construct a dependency representation for the sentence:

"Many tourists enjoy Egypt"

PT-6. Table-controlled shift-reduce parser

Draw up a complete network using the ambiguous grammar G2 for the sentence:

"My friends in Egypt sleep"

PT-7. Parsing with finite transition networks (FTN)

Process the following sentence according to the finite state transition table for G4:

"Many foreign tourist enjoy the pyramids"

PT-8. Augmented transition networks (ATN)

Process the following sentence according to the ATN-program:

"Gudrun feeds fish to her silly cat ."

PT-9. Chart parsing according to the slot-and-filler principle

Process the following sentence according to the given valency patterns and valency references:

"Gudrun feeds fish to her silly cat ."

Table of Contents

1 Parsing Issues *1*

1.1 What is Parsing? **1**

1.2 Prerequisites of a parser **2**

1.3 Connection between grammar and parser **2**

1.4 The type of the structural description **4**

1.5 Complex categories and unification **5**

1.6 Grammar specification formats and basic recognition strategies **5**

1.7 Constructing a parse tree .... Fehler! Textmarke nicht definiert.

1.8 Processing the input .... Fehler! Textmarke nicht definiert.

1.9 Handling of alternatives **9**

1.10 Control of results **10**

1.11 Well-formed substring table (Chart) **11**

1.12 Overall Control **11**

1.13 Checklist **12**

2 Prototypical parsers *13*

PT-1. Top-down parser with backtracking **13**

PT-2. Top-down parser with parallel-processing **22**

PT-3. Top-down predictive analyzer (with Greibach normal form grammar) **24**

PT-4. Top-down parser with divided productions **27**

PT-5. Bottom-up parser with a well-formed substring table **32**

PT-6. Table-controlled shift-reduce parser **38**

PT-7. FTN-parser for regular expressions **48**

PT-8. Augmented transition networks (ATN) **53**

PT-9. Chart Parsing according to the slot-and-filler principle **65**

3 Comparative evaluation of the parser prototypes *78*

3.1 Efficiency **78**

3.1.1 Control of alternatives 78

3.1.2 Overgeneration 79

3.2 Coverage **80**

3.3 Drawing up Lingware **81**

3.3.1 Perspicuity 81

3.3.2 Side effects 82

3.3.3 Usability of external resources 83

3.4 Suitability for the application **83**

4 Bibliography *85*

5 Exercises *87*