Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

**Document**
D-92-21

# Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars

Anne Schauder

March 1992

**Deutsches Forschungszentrum für Künstliche Intelligenz GmbH**

Postfach 20 80
D-6750 Kaiserslautern, FRG
Tel.: (+49 631) 205-3211/13
Fax: (+49 631) 205-3210

Stuhlsatzenhausweg 3
D-6600 Saarbrücken 11, FRG
Tel.: (+49 681) 302-5252
Fax: (+49 681) 302-5341

# Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern und Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Daimler Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Krupp-Atlas, Mannesmann-Kienzle, Philips, Sema Group Systems, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct *systems with technical knowledge and common sense* which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- Intelligent Engineering Systems
- Intelligent User Interfaces
- Intelligent Communication Networks
- Intelligent Cooperative Systems.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Prof. Dr. Gerhard Barth
Director

# Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars

**Anne Schauder**

DFKI-D-92-21

Diese Arbeit wurde finanziell unterstützt durch das Bundesministerium für Forschung und Technologie (FKZ ITW-8901 8).

# Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars

Anne Schauder

**Abstract**

This document combines the basic ideas of my master's thesis – which has been developped within the WIP project – with new results from my work as a member of WIP, as far as they concern the integration and further development of the implemented system. ISGT (in German 'Inkrementeller Syntaktischer Generierer natürlicher Sprache mit TAGs') is a syntactic component for a text generation system and is based on Tree Adjoining Grammars. It is lexically guided and consists of two levels of syntactic processing: A component that computes the hierarchical structure of the sentence under construction (*hierarchical level*) and a component that computes the word position and utters the sentence (*positional level*). The central aim of this work has been to design a syntactic generator that computes sentences in an *incremental fashion*. The realization of the incremental syntactic generator has been supported by a distributed parallel model that is used to speed up the computation of single parts of the sentence.

## Contents

# 1 Introduction

## 1.1 Motivation

There are several reasons to develop a natural language generation system. Computer scientists are interested in improving the user interfaces of their systems by using natural language. Even non-specialist users should be able to communicate easily with, e.g., an expert system. It is helpful to study human behavior when developping the concepts for a natural language processing system. The complex processes of perception, production and acquisition of natural language should exactly be described and explained with computational means ([Wahlster 82]) and can thereby contribute to research in linguistics, psychology and medicine. What is needed is a cross-disciplinary co-operation of computer science, linguistics, psychology (psycholinguistics and cognitive psychology) and medicine (neurolinguistics).

A natural language generation system must solve three tasks: It has to decide what to say, i.e., it must plan the contents of an utterance. It has to compute how to say it, i.e., it must design the syntactic form of the utterance. Finally, the utterance has to be articulated (spoken or written). The system presented in this work is designed to primarily solve the second task. The two other levels of natural language generation are treated only as far as they concern the definition of the interfaces.

The module that solves the second task is the so-called syntactic generator. It strongly depends on the underlying grammar formalism controling the combination of words into sentences. Tree Adjoining Grammar (abbreviated TAG) is a promising formalism that seems to ease the simulation of generation processes. In particular, it offers facilities for *incremental* processing. In this case, incremental processing means that the generator receives incomplete parts of the message, transforms them into a syntactic form and utters first parts of the sentence as soon as possible. Later incoming parts of the message are integrated into the existing syntactic structure. This style of processing helps to avoid long initial delays, i.e., long pauses between single sentences. The focus of the presented system lies on incremental generation.

## 1.2 Overview

The system ISGT (in German '**I**nkrementeller **S**yntaktischer **G**enerierer natürlicher Sprache mit **TAG**s') was the topic of my master's thesis worked out at the DFKI Saarbrücken ([Schauder 90]). Being an incremental sentence generator, it represents a first prototype for the syntactic component that is part of the project 'Knowledge-based Presentation of Information' (WIP[1], in German '**W**issensbasierte **I**nformations-**P**räsentation', see [Wahlster et al. 88]). The WIP system is characterized by

- context-directed selection of information to be presented,
- multimodal presentation of information and
- multilingual presentation of information.

[1]The WIP project is supported by the German Ministry for Research and Technology under contract no. ITW8901 8

One presentation mode is the textual encoding of information. A syntactic component is responsible for the generation of text. It incrementally creates surface structures (sentences with associated syntactic structures) by computing parts of the input and integrating each newly incoming piece of information into the set of realized fragments of the sentence. The incremental natural language generator was planned to be based on the grammar formalism TAG.

In Section 2, psycholinguistic and computational aspects of incremental natural language generation are illuminated. They result in a set of criteria for the design of the developped system.

The definitions of the formalism 'Tree Adjoining Grammar' and some extensions are briefly described in Section 3 and their suitability for incremental generation is discussed.

Sections 4 and 5 contain the basic features of the system ISGT. A schematic presentation in Figure 1 eases the orientation.

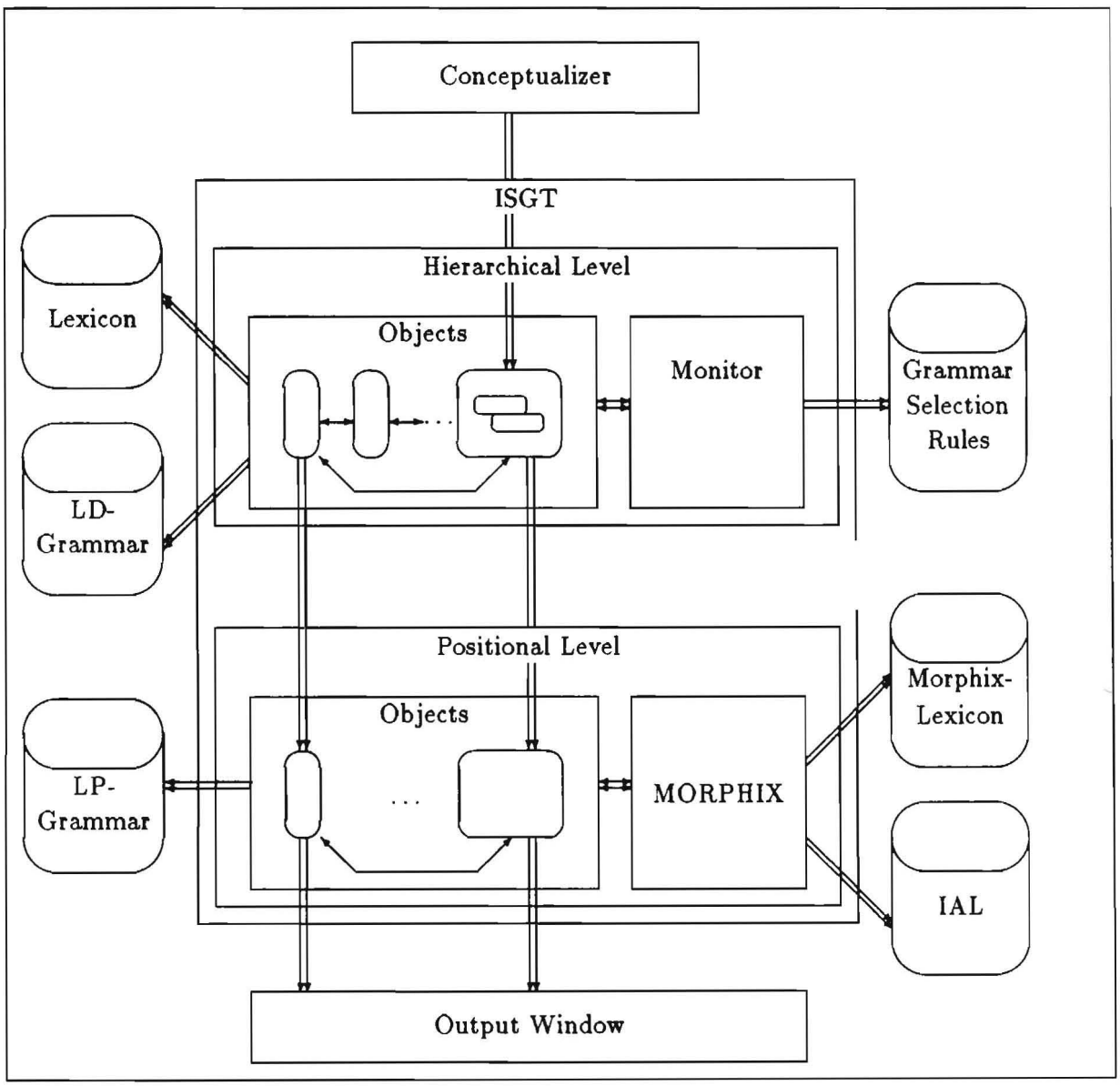


Figure 1: Schematic Presentation of ISGT

The interfaces between the module and the conceptualizer (telling what to say) and between the module and the articulator represent input and output of the developped system. The syntactic generator itself consists of two components. Both are based on a distributed parallel model. Active objects (represented as ovals in Figure 1) compute related parts of the representation structure on both levels.

The hierarchical level is explained in Section 4. Its objects are created directly using the input from the conceptualizer. The goal of the objects is to build the sentence tree. They use knowledge from the lexicon and the LD- (local dominance, see Section 3.4) part of the grammar to build their part of the syntactic tree. The complete sentence tree is constructed by the interaction of all objects that communicate via message passing. The objects are supported by a special object, the 'monitor', that helps them to choose a grammar rule and combine the local parts of the structure.

Partial results from the hierarchical level are handed over to the positional level (represented in Figure 1 by directed arrows from the single objects of the hierarchical level). The positional level is described in Section 5. The objects at this level linearize complete parts of the syntactic tree and bring the partial trees into a correct order, according to the LP- (linear precedence, see Section 3.4) part of the grammar. The ordered leaves of the trees are inflected using the module MORPHIX (see [Finkler & Neumann 89]) and uttered as soon as possible (incrementally).

In the last section (Section 6), the results of this work are summed up. Possible improvements and alternative approaches are mentioned in an outlook.

# 2 Criteria for the Development of an Incremental Natural Language Generator

Demands on a system for incremental natural language generation are first of all consequences of results from research in psycholinguistics and artificial intelligence. Psycholinguistic observations of human language processing make demands upon the performance of natural language human-computer interfaces. Psycholinguistic models offer criteria for the processes involved in human natural language production and for a computational simulation. From the point of view of artificial intelligence, guiding principles must be found for an efficient and adequate realization of natural language generation. Since this work is concerned with natural language generation from the computational point of view, the psycholinguistic aspects are only briefly described in Section 2.1. Sections 2.2 to 2.4 deal with questions of system design.

## 2.1 Language Production from the Psycholinguistic Point of View

Natural language generation means the production of natural language utterances in order to satisfy specific communicative goals ([McDonald et al. 87a]). The human production process cannot fully be observed: The input into the generation module – the message or contents of the planned sentence – cannot be made visible. But it is essential for the interpretation of psycholinguistic experiments that input and output can be controlled. Wide-spread experiments today are the examination of speech errors and the observation of natural language utterances stimulated by nonverbal input (e.g., during the description of visual scenes).

Just one psycholinguistic model is presented here, because it is most similar to the scheme of the presented system. The components of Kempen's model ([Kempen 77]) work as follows: The conceptualizer chooses conceptual structures that are to be uttered. It

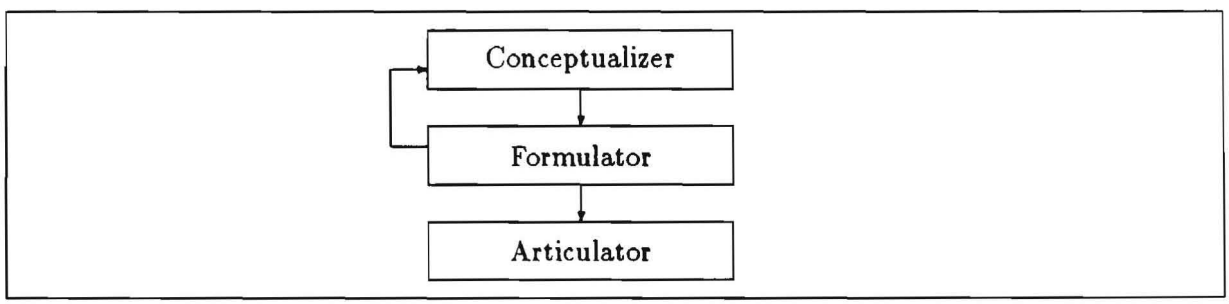


Figure 2: Kempen's Model for Language Production

creates the idea of the sentence, which is handed over to the next stage as the 'message'. The formulator translates the nonverbal idea into verbal structures. The result of this process is a syntactic representation that should satisfy the communicative goal as good as possible. Finally, the articulator controls the utterance of the sentence.

The three levels of this model reflect three tasks that have to be solved to generate a natural language sentence:

1. It must be decided, what to say, i.e., the relevant contents of the utterance have to be identified.

2. It must be decided, how to say it, i.e., the message has to be translated into a natural language sentence.

3. The sentence must be articulated, that means spoken or written.

Each natural language generation system must have these abilities. They are the kernel of all psycholinguistic models and of each computational natural language generation system (see Section 2.2). The modular approach is very useful when developping distinct parts of a generator. The presented syntactic component is a part of the formulator.

One special feature of Kempen's model is a feedback loop between the formulator and the conceptualizer. Kempen argues, that the conceptualization itself can depend on decisions made during the process of formulation. He motivates *interdependent processes of language production* explaining , e.g., that speakers can revise their concepts if otherwise the sentence cannot be completed in a synactically correct way.

Furthermore, Kempen's model allows for parallelism of the processes on the different stages. This is called *incremental processing*: The processes work in parallel and incrementally compute partial results as soon as possible and hand them over to succeeding processes. In natural language generation, the time that passes during the utterance of first parts of a sentence is used to compute further parts of the message that have to be integrated into the same sentence. Thereby, communication can be sped up. Since this effect is important for computer systems as well, incremental generation plays a central role in today's research.

## 2.2 Natural Language Generation in AI

The following section deals with important principles for modern natural language generation systems that contribute to an adequate behavior in human-computer dialogue. Especially the demands on the formulator are considered.

The formulator decides how a message should be expressed in a given language. The formulator transforms selected conceptual items into an adequate syntactic form and must thereby be able to choose among linguistic alternatives. Semantic knowledge must be converted into syntactic knowledge, adequate lexemes (i.e., minimal meaningful items) must be chosen for the conceptual elements and the relations between them, the syntactic structure must be constructed, the leaves of the syntactic tree must be linearized and inflected. ISGT starts after word choice is done, so the first two tasks won't be examined any further.

The construction of the syntactic tree, linearization and inflection are based (among others) on linguistic knowledge bases containing

- morphological knowledge (e.g., information for the inflection of words), and
- a grammar that constrains which words can be combined how, i.e., that defines linguistic alternatives and constraints.

The system ISGT is based on the Tree Adjoining Grammar formalism. The use of this formalism can be motivated by linguistic and computational reasons (see Section 3.6).

The next two sections briefly characterize the embedding and functionality of the formulator as part of a natural language generation system.

### Embedding of the Formulator

The most adequate model for our demands seems to be the cascade. In a cascade (used, e.g., in POPEL, see [Reithinger 88]) every component may have only one predecessor and one successor. The form of the cascade reflects the view that the components of a generator are arranged vertically. From the computational point of view, a cascade has several advantages. Cascades allow for incremental processing, as partial results can be handed over from one component to the succeeding one. In this way, all components can work in parallel. The connection of every stage with its predecessor supports feedback through the whole system. Components on a deeper level can thereby influence the work of higher modules.

A discussion of alternative models like hierarchical, blackboard, heterarchical and integrated model can be found in [Finkler 89].

### Alternative Concepts for the Formulator

The functionality of ISGT can be characterized by the term 'description-directed approach' introduced by [McDonald 87b].

The basic idea for the description-directed approach consists in inserting an additional level of explicit linguistic representation between message and utterance. McDonald claims that using a syntactic description of the actual developped text is the most effective means for introducing grammatical information and grammatical constraints into the realization process. There can be several stages of representation between message and word sequence showing the text on distinct levels of abstraction. McDonald calls this approach "multi-level, description-directed generation". Formulation is then organized as a sequence of decisions made by specialists, the output of each specialist being a linguistic representation of the message. The data on the distinct levels are – as a specialization of data-driven control – directly interpreted as instructions for the virtual machine that models the generator. This principle is used by [McDonald & Pustejovsky 85] in the system MUMBLE.

In the description-directed approach, each existing level should deal only with those tasks that have to do with its *natural capacity*. Not only modularity is an advantage for the system design but also the fact that descriptive syntactic representations are used instead of procedural ones. Description-directed control allows for an incremental style of processing, as the order of decisions is not fully defined. By adequately distributing the task over several processes there can by parallelism between and on the various stages of the generator.

An additional motivation for the choice of a description-directed approach is the presupposition of the descriptive grammar formalism TAG and the orientation at the description-directed, multi-level system POPEL-HOW (see [Neumann 89] and [Finkler 89]).

## 2.3 Incremental Syntactic Generation

Description-directed control is motivated by the supposition that grammatical information can best be introduced into the generation process by a syntactic representation of the sentence actually under construction. For a TAG-based generator it must be examined if the grammar formalism remains adequate with respect to the additional demand for incremental generation.

The data on the levels of representation inside the syntactic generator (one level for the hierarchical structure and one for the word position) are linguistic representations of the message. The incremental input from the predecessing stage activates a process that at the best inserts new partial structures into the existing syntactic tree. Otherwise, a partial tree is constructed separatly from the existing one and has to wait until it can be associated. At the worst, the existing syntactic tree must be revised.

Using the TAG formalism means that partial structures are represented as TAG trees, that the constructed sentential tree consists of a modified initial tree, and that partial structures can be combined by adjunction. These terms are defined in Section 3.1.

In the following, the consequences of an incremental style of processing are examined. They will lead to several extensions of the TAG formalism introduced in Section 3. [Kempen & Hoenkamp 82] have laid out a catalogue of demands for the organization of an incremental syntactic generator. Some of them will be discussed in the next sections.

### The Levels of Syntactic Generation

In an incremental generator, hierarchy and order of constituents should be computed by distinct components. If a word is fit into the sentential hierarchy and at the same time the word order is fixed, it is not for sure that the most significant rule has been chosen. It is possible that the succeeding input elements arrive in another than the defined word order. Then they can't be flexibly integrated any more – even if the grammar rules would allow for it. It seems useful to postpone the computation of word order and realize it at a deeper component (this corresponds to the second demand of Kempen and Hoenkamp).

The linguistic terms *connection* and *position* reflect this seperation (see [Engel 77]). Connection means the restriction of word combinations, each word constraining its context semantically and syntactically. Position constitutes linear relations between elements, thereby defining word order.

Incremental generation must be combined with a variable computation of word position. With the exception of free word order languages new syntactic fragments cannot simply be added to the right side of the actual tree. Their insertion depends on the word order rules of the grammar. With increasing freedom of word order the output order can be oriented at the input order and first parts of the output can be produced very fast.

The result of this discussion is the separation of the syntactic generator into two levels: At the *hierarchical level* words are inserted into a syntactic tree that can be interpreted as a mobile (see Figure 3).
At the positional level a correct order of trees and subtrees is computed (see Figure 4).

As will be shown in Section 3.4, this separation can be realized with help of an extension of TAG called LD/LP-TAG. In the *Local Dominance-* (LD) part, trees are treated as mobiles, the *Linear Precedence-* (LP) part defines the positioning rules.

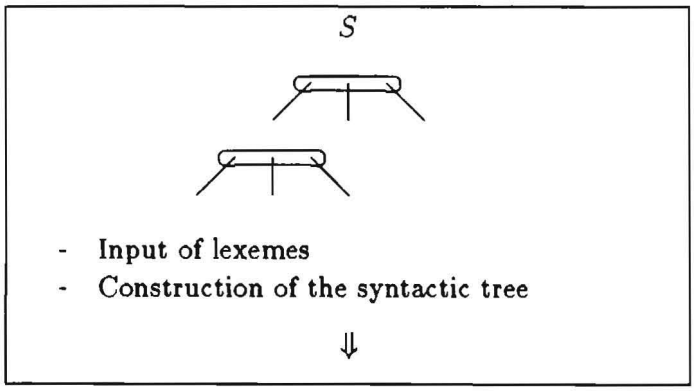


Figure 3: The Hierarchical Level

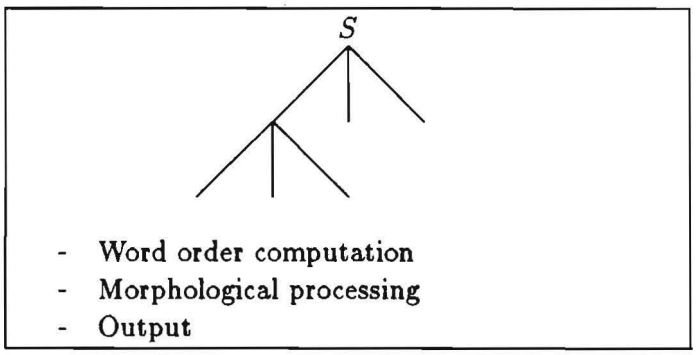


Figure 4: The Positional Level

The following section deals with problems of the representation of the syntactic structure at the hierarchical level. Again, the incremental style of processing plays the central part.

### Conceptual and Lexical Guidance at the Hierarchical Level

Incremental processing at all levels of a system is best supported by handing over partial results in a piecemeal fashion. The efficiency of incremental processing at the hierarchical level depends on the kind and form of input it gets from the word choice process.

When constructing the syntactic tree top-down from the definition of global attributes by stepwise refinement down to the leaves, the choice between alternative subtrees must be made without regarding the respective lexemes. But these lexemes have syntactic features (e.g., the valency of a verb) directly influencing the structure of the syntactic tree. They should guide the construction process. Additionally, the properties of the conceptual input elements should be considered: Functional relationships determine the functional insertion of lexemes into the tree (e.g., the definition of a subject in an active clause). The following strategy results from these considerations: *The incremental input of lexemes and functional relations between them guides the construction of the syntactic tree.* This corresponds to the demands number one and three of Kempen and Hoenkamp.

Although the structures of the TAG formalism are not yet introduced, it will be discussed in the following sections which size the subtrees associated with the lexemes should have. Since the rules of a TAG are represented as syntactic trees, this discussion

can directly be applied to the formalism and motivates a TAG extension that defines a special kind of trees: lexicalized TAG (see Section 3.5).

We need two more terms from linguistics. Phrase structure trees can be called *constituency diagrams* describing a relationship between nodes of trees (in Figure 5, the tree can be read level by level as, e.g., 'S consists of NP and VP', 'NP consists of N', and so on). TAG trees are constituency diagrams, as we will see in Section 3. In contrast with constituency diagrams, dependency diagrams represent every element exactly once. The vertical order (father-son relationship) defines the *dependency relation*, i.e., it describes which element depends on which other. Each father (regent) rules its sons (dependents). There are different linguistic theories underlying dependency relations; for German the main verb of a sentence is often viewed as the central element introducing the 'structural frame' of the sentence. In Figure 5 a sentence is represented as a constituency (left) and as a dependency diagram (right).

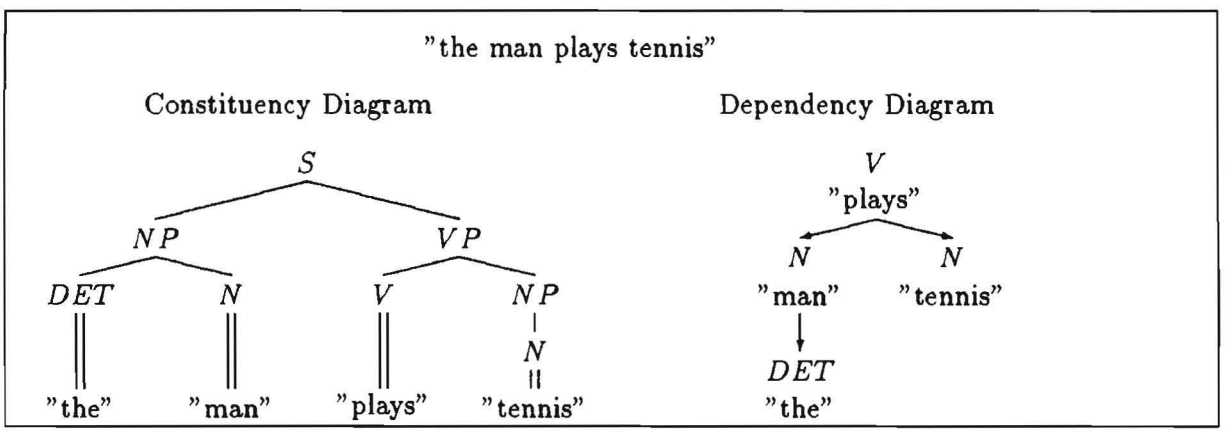


Figure 5: Constituency and Dependency Diagram

The highest element of a node group in a diagram is called *head.* In the example, $V$ is the highest element of the whole diagram, the left $N$ is the highest element of the group consisting of $N$ and $DET$, and so on. The term *phrase* means a part of a sentence, whose name can be derived from its head (e.g., $NP$ for a nominal phrase). The abbreviations $XP$ for $X$-phrases are often used in constituency diagrams.

As is visible in Figure 5, a constituency diagram can consist of several phrases: Internal nodes are roots of subtrees which themselves represent phrases and which are ruled from their central terminal node – their head. This observation has consequences for the size of subtrees that can be chosen according to the given lexemes. If such a tree includes several phrases and detailed descriptions of their constituency structure and nevertheless is chosen for only one lexeme (e.g., the $V$ in example 5), decisions are made that in fact depend on the input of the other lexemes (e.g., the two nouns in Figure 5). This argument is related to the second demand of Kempen and Hoenkamp: The size of rules that are chosen for given lexemes must be oriented at the given information. Therefore *the syntactic subtrees that are chosen for given lexemes may only reflect details of that phrase, whose head the lexeme is.*

The results of this discussion are summed up in the following:

1. The incremental formulator consists of two stages: the hierarchical and the positional level. This distinction allows for the structural insertion of lexemes without

regarding their position in the sentence.

2. Incremental processing at the hierarchical level highly depends on a sensible choice of subtrees for the representation structure. The chosen parts shall describe only that part of information that can be associated with a given lexeme.

Another concept (see also [Kempen & Hoenkamp 82]) is the parallel computation of independent structures, which will be motivated in the next section.

## 2.4 Parallelism in the Syntactic Generator

At the hierarchical level, processing takes place without regarding word position; the constructed tree is considered as a mobile. This is why, a construction *from left to right* is undefined. The insertion of a new element only depends on its functional relation to the rest of the tree. Most efficiently, new lexemes are integrated in their input order. Furthermore, one can take advantage of the relative independence of the subtrees by computing them in parallel.

ISGT is designed as a cascade, because of a cascade's usability for incremental processing and feedback. The model can best be used not only by working in parallel on the distinct levels of the cascade but also by realizing parallelism within those levels. An incremental style of processing presupposes an adequate separation of the data into segments. Single segments can be computed and lead to partial results handed over to the next level. A distribution of these segments onto parallel processes allows for the simultaneous production of partial results, their simultaneous handing over, and further parallel computation on the next level. This kind of parallelism supports incremental processing in the cascade.

The distributed parallel model used for ISGT is based on *object-oriented concurrent programming.* Details can be found in [Yonezawa & Tokoro 87] and [Finkler 89]. There are objects at each level of the formulator that can cooperate or work independently. Each object can be seen as an integral unit consisting of data and procedures that operate on the data. Each object is associated with a process. During the creation of an object, its local variables are initialized and a sequential program is handed over to it. The actual state of an object is defined by the position in its program and the values of its variables. These variables are local and cannot directly be read from the outside, but they can be manipulated by communication with the object.

### The Life Cycle of the Object

Objects are created when there are partial inputs from the higher level. Their goal is to transfer a segment into the deeper level. Therefore, they run through an infinite loop of their program that describes their basic capabilities. The special tasks of the two levels (hierarchical and positional) are realized in respective specializations of the program. At each single level, all objects have identical programs as they all have to solve the same problem, i.e., the next verbalization step.

The life cycle of an object is divided into several phases:

1. initialization,

2. computation of the segment by sending messages to other objects or transferring data into the next level,

3. reading messages,

4. waiting until a new message is received or a defined time span has passed,

5. reading messages, then go to Step 2.

This cycle must be traversed again and again. The object must wait at the end of each pass, because further processing is only senseful if new information is available. But even if the object has successfully managed its task, it must not be terminated: Other objects could continue to send messages to it.

It will be motivated in Section 3.5 that there is no real transfer of single segments to new objects at the positional level. The object that finishes its task changes its goal of computation and uses other methods than before to reach this goal. Thereby, it can make use of its previously filled local variables.

Besides the objects working at the hierarchical and the positional level by changing their functionality, there exists one special object at the hierarchical level, called *monitor*. The monitor controls the incremental processing of the other objects as will be shown in Section 4.

### The Interaction of Objects

The objects communicate by 'message passing' in order to guarantee their data encapsulation. Each object has a local variable 'context' describing its partners of communication either by patterns or by their direct address. Each new object has to be registered at the monitor which transmits its address to other relevant objects.

The kind of interaction between two objects arises from the contents of the message. *Value passing* means the transfer of values, a *remote procedure call* demands the object to call a procedure. Messages to each object are collected in its 'port' and read during the respective phases of its life cycle. Message passing is either synchronous or asynchronous depending on the kind of situation.

### Simulating the Model on the LISP Machine

ISGT is implemented in an object oriented style on a system with one processor. The objects are defined on the basis of a flavor system (see [Steele 90]). The flavors inherit the system-defined *si:process*. That is why, the scheduler regards them as normal processes and supervises them according to the round robin system (see [Symbolics 89]). Parallelism can only be approximated by time sharing. In the next extension of our system we will realize parallelism by distributing the objects over several machines.

### Segmentation at the Hierarchical and the Positional Level

The input into the hierarchical level are lexemes associated with special information. The goal is to create a syntactic structure tree representing the sentence. Incrementally incoming lexemes are added to this tree and completed partial structures are handed over

to the next level. The representation structure must be segmented in such a way that the segments can be computed in parallel and as independently as possible. There are several demands on these segments:

1. The segments have to be oriented at the grammar rules. Each divergence from the original structure causes additional costs for its use.

2. The segments have to be chosen on the basis of the input incrementally and in the given order (see Section 2.3).

3. The segments have to be as independent as possible, so that they can be computed in parallel and without too much communication.

4. The partial results for the next level have to be computable by single objects and not by sets of objects.

The input into the positional level are partial structures from the hierarchical level. The goal is to transform parts of mobiles into an 'ordered' structure tree, to inflect the lexemes and to utter them in a correct order. Especially the structures from the hierarchical level have to be handed over incrementally. This means that partial trees are separated from the mobile and integrated into respective structures at the positional level. These two kinds of structures differ in the following points:

1. The structure tree on the second level represents exactly one possibility of ordering the mobile of the first level.

2. The leaves are substituted by the inflected words.

It will be shown in Sections 3.4 and 3.5 that lexicalized LD/LP-TAG can be used for the definition of segments. As there are no structural differences between the trees of the LD- and the LP-part (the computation of word order is directed by LP-rules associated with the mobiles), there is no use for a real transfer from the hierarchical to the positional level. On the contrary, it would lead to redundancy and problems of consistency (see Section 3.5). The central point is that hierarchical and positional level are not realized as distinct modules with different objects. The separation of the levels is rather implemented by *changing state and program* of the objects.

The formalism Tree Adjoining Grammar and its relevant extensions are defined in the next section. They are motivated with respect to the above formulated demands on an incremental syntactic generator.

# 3 Using Tree Adjoining Grammars for Generation

In the following sections, the TAG formalism and some of its extensions are informally defined. Their relevance for incremental natural language generation will be discussed Formal definitions can be found in the literature referred to.

## 3.1 Tree Adjoining Grammars (TAGs)

*Tree Adjoining Grammars* have been introduced by [Joshi et al. 75]. The basic idea o the formalism is the representation of elementary sentence structures as trees that can b combined to more complex structures (complex sentences).

A Tree Adjoining Grammar or TAG **G** can be defined as 5-tupel (N, T, S, I, A). N anc T represent finite disjunctive sets of *nonterminals* and *terminals*, S is a special symbo from N, the start symbol. The union of I and A is called the set of *elementary trees* which are the rules of the grammar. I and A represent disjunctive sets of *initial* (I) anc *auxiliary* (A) trees. A tree $\alpha$ is an initial tree, iff its root is labeled with the start symbo $S$, all leaves represent terminals and all internal nodes are associated with nonterminal (see left tree in Figure 6). From the linguistic point of view initial trees represent minima sentential trees, that are the basis for each complex sentence.

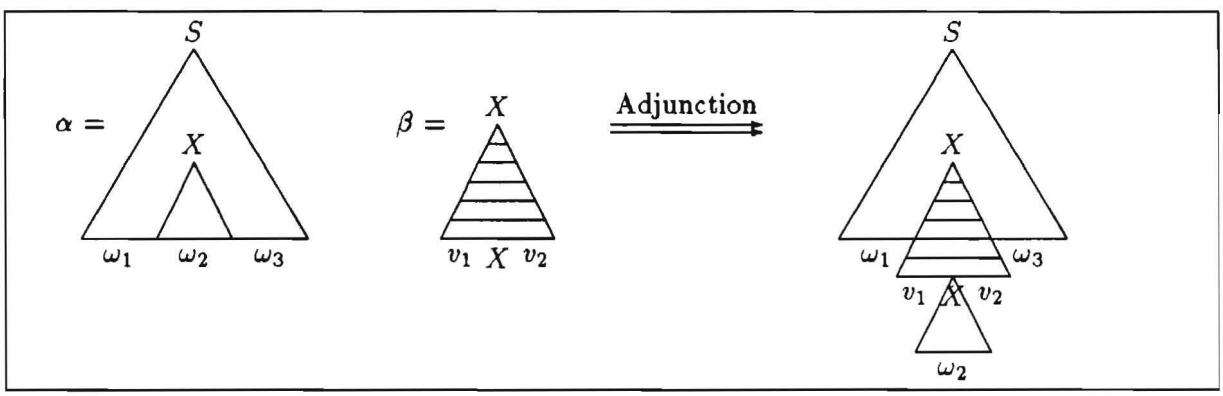


Figure 6: Elementary Trees and Adjunction

The second type of trees (auxiliary trees) is used to create trees that are not explicitly encoded in the grammar. A tree $\beta$ is an auxiliary tree, iff its root is labeled with a nonterminal $X$, there is exactly one leaf – the *foot node* – that is labeled with the same nonterminal as the root, all other leaves of the tree represent terminals (there has to be at least one terminal leaf) and all internal nodes are associated with nonterminals (see the tree in the middle of Figure 6). Auxiliary trees allow for recursion by defining that root and foot node have to be associated with the same label. Seen linguistically, an auxiliary tree corresponds to a minimal recursive or iterative construction. The structures are minimal because they have to be defined without recursion on a nonterminal.

There exists an important linguistic constraint for the size of TAG trees: *Each tree must describe a complete phrase with all obligatory parts.* E.g., a sentential tree for a transitive verb must contain the object, too.

Two trees can be combined by *adjunction* (or *adjoining*). Adjunction (see right tree in Figure 6) inserts an auxiliary tree $\beta$ into an initial or previously modified initial tree

$\gamma$ (the initial tree $\alpha$ in Figure 6). $\gamma$ (or $\alpha$) contains a node $n$ with label $X$, the so-called *node of adjunction.* It is replaced during adjunction by the auxiliary tree $\beta$, whose root and foot node must be labeled with the same symbol. The root node replaces the node of adjunction with respect to its father, the foot node becomes the new root of the subtree that hung under $X$. By this, $\gamma$ is modified or enlarged without losing parts of the tree.

An expanded definition of adjunction also allows to insert a tree into auxiliary or modified auxiliary trees. The formalism's power remains the same. Compared with the above-mentioned definition this leads to further variations with respect to the order of combinations. This can have consequences, e.g., for TAGs with Constraints or TAGs with Unification. The advantage of the expanded definition of adjunction is that the formalism can better be used for *incremental* generation. Trees should be combined as fast as possible, so it is not useful to constrain the order of combinations to a top-down processing. In the following, we always mean by adjunction the expanded definition of adjunction.

TAGs are mildy context-sensitive ([Joshi et al. 75]). This power seems to be adequate for the description of natural language and is another motivation for the use of this formalism.

In the following, some extensions of TAGs are described especially contributing to a more compact representation of complex syntactic structures.

## 3.2 TAGs with Constraints (TAGCs)

Basically, adjunction is allowed if the labels of root and foot node of the auxiliary tree correspond to the label of the node of adjunction. Tree Adjoining Grammars with *local constraints for adjunction* (see [Joshi 85]) allow for the restriction of the set of auxiliary trees that may be adjoined. Each nonterminal node of an elementary tree is associated with one of the following constraints:

**SA(X)** SA is the abbreviation for *selective adjunction.* It is not allowed to adjoin all auxiliary trees with the respective labels in this node, but only the specified subset X of them.

**NA** NA is the abbreviation for *null adjunction.* No adjunctions are allowed at this node.

**OA(X)** OA is the abbreviation for *obligatory adjunction.* At least one tree of the defined set X of structurally adjoinable trees must be adjoined.

The definition of adjunction is the same as for pure TAGs with the addition, that only those auxiliary trees may be adjoined in a node, which are contained in its constraint set. During adjunction the constraints of the node of adjunction are deleted, the constraints of root and foot node are taken over unchanged in the resulting tree.

Constraints enlarge the power of TAGs, but the resulting formalism is still mildly context-sensitive. Especially selective adjunction can be used for the representation of natural language, because it helps to express relations by constraints instead of node labels and thus avoids redundancy.

## 3.3 TAGs with Unification (UTAGs)

The expressive power of TAGs (with Constraints) does not allow for a compact encoding of complex syntactic information. E.g., agreement tests (for the equality of number, gender and case) must be realized by explicitly defining TAG trees with respective node labels (e.g., N.1.sg.nom). The combination of TAGs with the unification formalism helps to avoid redundancy and makes it easier to design a grammar.

### Unification

Unification is presented according to the PATR formalism (see [Shieber 86]). The idea of PATR is that context-free rules can be associated with so-called unification rules. Such a combined rule consists of a *constituent list* and a *specification list.* The constituent list is derived from the context-free rule by taking the left side as first element and all other elements in their given order from left to right (e.g., $(S\ NP\ VP)$ for $S \rightarrow NP\ VP$). The specification list describes the unification rules for each context-free rule. It refers to elements of the constituent list by enumerating them from left to right, beginning with zero (for $(S\ NP\ VP)$, 0 refers to $S$, 1 to $NP$ and 2 to $VP$). A specification is a list of two elements of the form ({ attribute | path } { path | value }). An attribute is a path with one element. A path consists of a list of attributes. The right side of a specification can define an atomic value. These rules specify a feature structure for each context-free rule that can be represented as DAG (Directed Acyclic Graph, see Figure 7).

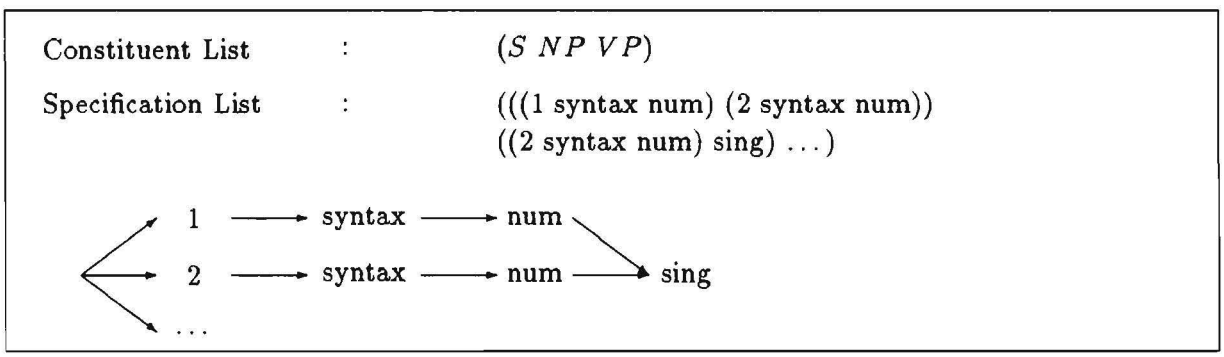


Figure 7: Specification List and DAG

The rule in Figure 7 guarantees that the nominal and the verbal phrase of sentence S have the same number value. The attribute *num* is placed behind *syntax*, because it describes a syntactic information. The second specification defines the value *singular* for the number of the verbal phrase. The first rule makes it possible that this value can also be read for the nominal phrase via the path *(1 syntax num).*

The identification of two paths and the definition of a value inside feature structures are two different sides of a unique process that is used to compile specification rules. It is called *unification.* Unification means the combination of parts of feature structures without contradiction. The result of the unification of two DAGs $d_1$ and $d_2$ is a DAG d, with

1. $d = d_1$, if $d_1 = d_2$,

2. $d = d_1$, if $d_1$ consists of a value and $d_2$ is empty,

3. $d = d_2$, if $d_1$ is empty and $d_2$ consists of a value,

4. if neither $d_1$ nor $d_2$ consist of a value, then:
   $\forall$ attributes l, with: $l \rightarrow n_1 \in d_1, l \rightarrow n_2 \in d_2$ (common path prefices), is $l \rightarrow$ Unification$(n_1, n_2) \in$ d and
   $\forall$ attributes l, with: $l \rightarrow n \in (d_1 \cup d_2) \setminus (d_1 \cap d_2)$ (i.e., path starting in exactly one DAG), there is $l \rightarrow n \in d$,

5. otherwise the unification fails and the result is NIL.

The combination of Tree Adjoining Grammars with unification can easily be motivated. Intuitively, each node of a TAG tree can be taken as left side of a context-free rule, its sons as the right side. In this way each node can naturally be associated with a specification list.

### TAGs with Unification

While TAG trees can easily be associated with specification lists it is much more difficult to define the process of adjunction on trees with feature structures. Unification can be understood as monotonic operation because is enlarges feature structures instead of really changing them. Adjunction can be viewed as nonmonotonic in the following sense: The neighbourhood relations of the node of adjunction to the surrounding nodes are destroyed. The father of the node of adjunction becomes father of the root node of the inserted auxiliary tree, the sons become sons of the foot node. There is no direct way to transfer the feature structures of the node of adjunction to the auxiliary tree in an appropriate way. Similar to the transfer of neighbourhood relations, the relations of the feature structures of the node of adjunction to the surrounding nodes have to be identified and transferred to the auxiliary tree.

This can be done by dividing the feature structure of a node of adjunction X into two parts (see [Buschauer et al. 91]): $\uparrow$X contains all feature structures that relate X with its father, $\downarrow$X contains all feature structures that relate X with its subtree. The left tree in Figure 3.3 shows that the direct association of nodes with PATR-style specification lists leads to local feature structures at nodes describing the relation of this node (represented by the substructure under attribute 0) to its sons (represented by the respective reference numbers 1 to n). Therefore, $\uparrow$X is always a part of the feature structure of the father of X, $\downarrow$X is defined locally with the node X itself.

This separation can be used for the definition of adjunction with unification. An auxiliary tree $\beta$ is adjoined into a node X as follows:

1. X, its specification lists, and its feature structure are deleted from the tree. The relations of the feature structure to the father and to the sons of X are cut off.

2. The auxiliary tree $\beta$ is inserted.

3. The feature structure of the foot node of $\beta$ is unified with $\downarrow$X, thereby creating new relations to the sons of X.

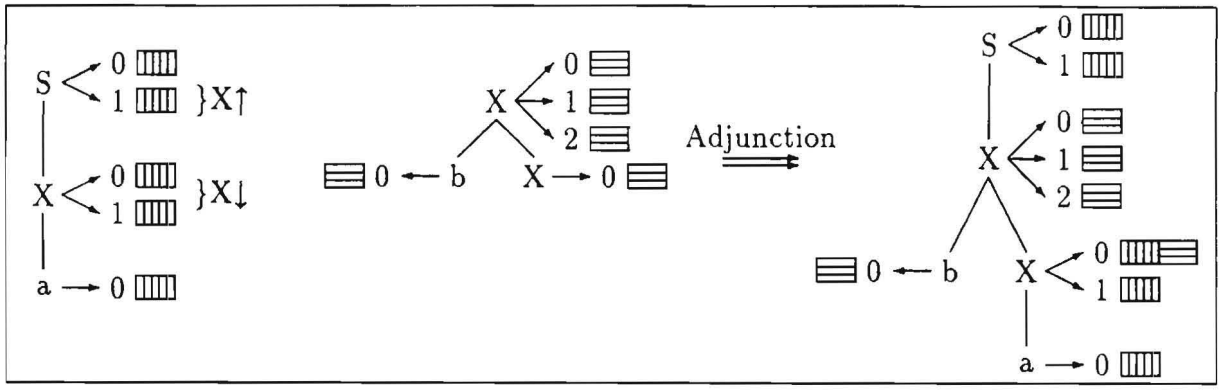


Figure 8: Adjunction with Unification

4. Since ↑X is associated with the father of X and the father becomes the father of the root node, the relation between these nodes is implicitly transferred.

An example for adjunction with unification is schematically represented in Figure 3.3. The resulting tree shows how the feature structures of the left tree (marked by vertical lines) and the feature structures of the auxiliary tree (marked by horizontal lines) are combined.

There are at least two ways to realize this definition of UTAGs (see [Schauder 92]). First, the feature structures of all nodes of each elementary tree are unified destructively by default (i.e., the i-substructure of a node is unified with the 0-substructure of its ith son). This leads to trees associated with global feature structures that allow for the direct inheritance of values by structure-sharing. But in case of adjunction in a node X, it is difficult to localize ↑X and ↓X. The destructive unification of the feature structure of X with the feature structures of the surrounding nodes makes it impossible to identify local structures. Therefore, adjunction with unification can only be realized on the basis of backtracking. The specification lists must be stored in addition to the feature structures at the nodes. During adjunction, the feature structures of both trees are thrown away and have to be rebuilt by compiling the specification lists into new feature structures and unifying their respective parts.

In a second kind of realization, the local feature structures of the nodes are kept separated in order to ease adjunction. The disadvantage of this approach is that there is no direct inheritance of values by structure-sharing. They must be accessed to by expensive reading operations over all local feature structures that should have been unified. Especially during incremental generation where it often cannot be guaranteed that no further adjunction will take place – then all feature structures could be unified – this reading operation is frequently used.

The decision which realization to use depends on the constraints of the actual system. This problem and a comparision with an alternative definition of Feature Structure based TAGs (FTAGS, see [Vijay-Shanker & Joshi 88]) are discussed in [Schauder 92].

A disadvantage of the combination of TAGs with unification is the increase of power for TAGs. But the simple encoding of complex syntactic features and rules is a strong motivation, so UTAGs are used in ISGT.

## 3.4 Local Dominance/Linear Precedence-TAGs (LD/LP-TAGs)

The trees of the previously discussed extensions of TAGs have the following disadvantage: Trees with the same structure but different orderings on their subtrees must be defined several times, the grammar becomes redundant. Furthermore, incremental generation demands a seperation of structures into hierarchical and positional ones (see Section 2.3). The aim of LD/LP-grammars is to distribute the positioning of subtrees from the pure hierarchical description.

An *LD/LP-TAG* is a 5-tupel (S, N, T, IS, AS). N represents the nonterminals, T the terminals ($N \cap T = \emptyset$), S is the start symbol from N. The tree sets are defined as follows:

**IS=** { $(\alpha, LP_\alpha)$ | $\alpha$ initial tree with root S without linear ordering (i.e., a mobile), $LP_\alpha$ set of <-relations on node numbers from nodes of $\alpha$ },

**AS=** { $(\beta, LP_\beta)$ | $\beta$ auxiliary tree without linear ordering (i.e., a mobile), $LP_\beta$ set of <-relations on node numbers from nodes of $\beta$ }.

Figure 9 shows two trees with associated LP rules. In this case, the LP rules define the order of the terminals totally, but the freer the word order of a language is, the fewer LP rules must be associated with a tree.

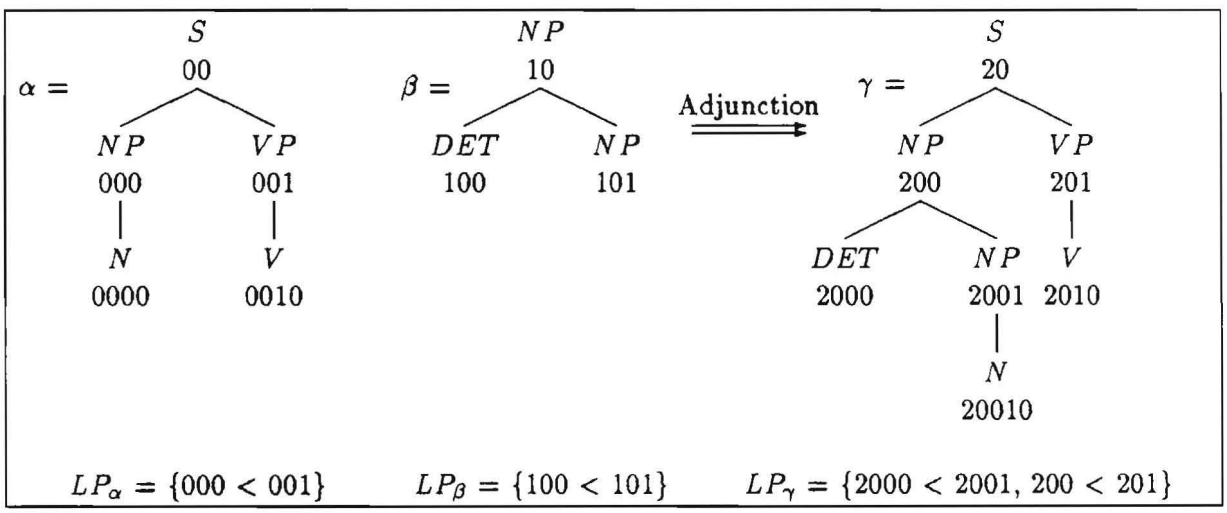


Figure 9: Elementary Structures and Adjunction in an LD/LP-TAG

LP rules are only allowed if they describe relations between disjunct subtrees of an elementary tree. None of the related nodes may be the predecessor of another. This is why every LP rule can be transformed into an aquivalent rule between sisters, namely the roots of the disjunct subtrees. This property can be used during incremental processing: During the top-down or bottom-up traversal of a tree, the <-relations are known for all subtrees. These relations help deciding which part of the tree can first start uttering its terminal string and which part has to wait for another one. If the decision is not possible

at the actual node, the question is handed over to its father (up to the root) and handled locally as before (for details see Section 5.2).

Adjunction for LD/LP-TAGs is defined as follows: Let $(\gamma, LP_\gamma)$ be an LD/LP-TAG structure (either elementary or modified), let $(\beta, LP_\beta)$ have root and foot node with label X and a corresponding node within $\gamma$. The result of an adjunction of $(\beta, LP_\beta)$ in X in $(\gamma, LP_\gamma)$ is

$(\overline{\gamma}, LP_{\overline{\gamma}})$ with: $\overline{\gamma}$ is the normal adjunction result (as mobile), $LP_{\overline{\gamma}} = LP_\gamma \cup LP_\beta$ using the newly defined node numbers.

It becomes clear, that the sets of LP rules are unified in the resulting tree. These rules cannot contradict because of their locality.

LD/LP-TAGs allow for a separation of the grammar into a hierarchical and a positional part which can be used on the respective levels of the generator (see Section 2.3). Each tree is taken as mobile and its definite form is described by the associated LP rules. Although LD/LP-TAGs in principle allow for the separation of hierarchical and positional descriptions, they don't seem to be adequate with respect to languages with relatively free word order. German phrases are often not arbitrarily movable but can be found in several permissible orders. Even the primitive sentences "Ich kaufe Äpfel" and "Äpfel kaufe ich" cannot be represented within one single structure by use of LD/LP-TAGs. It could be an interesting goal of further investigations to find alternative definitions for linear precedence relations.

Further disadvantages of Tree Adjoining Grammars have to do with the structure (especially the size) of their trees. Following the linguistic constraint, each tree must describe a complete phrase with all its obligatory parts. Furthermore, they must be expanded down to the preterminals. This results in the following problems:

1. The same subtrees can be found at different places in distinct trees, the grammar is redundant.

2. Trees must not be designed arbitrarily small (as a consequence of the linguistic constraint). That is why choosing a tree often means to decide about a larger structure than your knowledge about the input lexeme allows for. This contradicts to the demand that *the syntactic subtrees chosen for given lexemes may only reflect details of that phrase, whose head the lexeme is* (see Section 2.3).

3. In spite of that, the linguistic constraint has to be preserved, for it reflects an adequate use of the enlarged domains of locality. All obligatory parts in the subcategorization frame of a lexeme shall be represented in the chosen tree.

*Lexicalized TAGs* allow for this kind of *lexical guidance* that has been motivated in Section 2.3.

## 3.5 Lexicalized TAGs (LTAGs)

Following [Schabes et al. 88], a lexicalized grammar consists of

1. a finite number of structures, each associated with a lexeme which must be the head of the structure, and

2. combination operations for these structures.

3. There can be constraints specified over the set of structures which are local with respect to their lexical heads.

A grammar that is lexicalized in this way not only produces the same language as the original grammar but also derives the same structures. The idea can be applied to Tree Adjoining Grammars ([Schabes et al. 88]) and seems to create better presuppositions for incremental processing.

### Definition of LTAGs

The definition of trees is basically the same as in TAGs. In addition to that, the following is defined:

- Elementary trees may have (apart from the foot node of auxiliary trees) leaves labeled with nonterminals. These nonterminal leaves must be marked (X↓) in order to differentiate them from foot nodes. They are called *substitution nodes.*

- Each elementary tree must contain at least one terminal leaf representing the head of the linguistic structure described by the tree. For structures with just one terminal this must be the head; if there are several terminal leaves the head is linguistically defined, e.g., following the dependency theory. The head of the initial S-tree on the left side of Figure 10 is $V$.

- Apart from the initial S-type trees (initial trees with root $S$) there may be arbitrary initial X-type trees. Initial X-type-trees can replace substitution nodes with the same label, in order to create complete derivation structures. In Figure 10, the initial $NP$-type tree can replace the node NP↓.

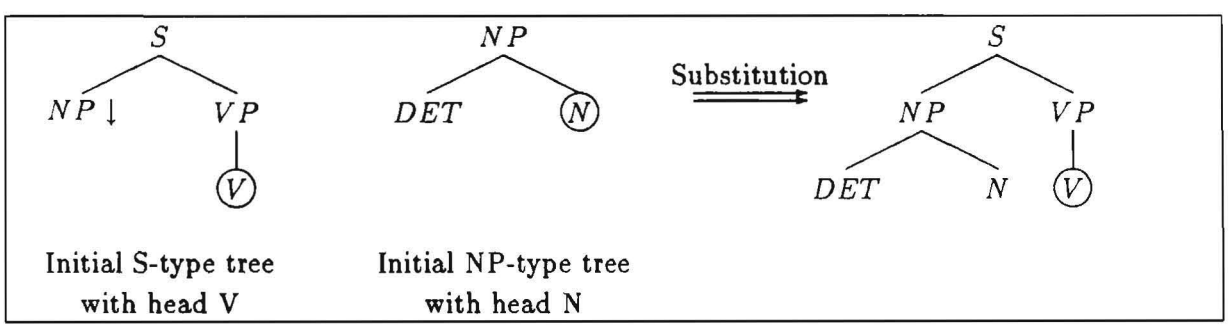


Figure 10: Initial Trees and Substitution for Lexikalized TAGs

Initial X-type trees cannot substitute arbitrary internal nodes of elementary trees, as auxiliary trees can. They have no foot node preserving the subtree of the substituted node. $X$-type trees always replace nonterminal leaves labeled with the same symbol $X$. This operation is called *substitution*:

- Substitution inserts an initial (or modified initial) $X$-type tree into an elementary tree by substituting a nonterminal leaf X↓ (this operation is used by context-free grammars). Figure 10 shows an example for substitution.

- Substitution nodes can be associated with constraints which are comparable with local adjunction constraints. But substitution is always obligatory, only the set of trees that may be substituted can be constrained.

- If substitution is marked at a node, adjunction is forbidden.

Substitution can be introduced to the TAG formalism without problems because it can be simulated by the more powerful operation of adjunction. The new operation enlarges the descriptive power of the formalism without changing its generative capacity. The advantage of substitution is its natural adequacy for lexical insertion and for syntactic constructions where the power of adjunction is not needed. A derivation structure of LTAGs is not complete before all substitution nodes are replaced.

The special rule that the head must be beyond the terminals of each structure of lexicalized TAGs has consequences for the lexicon:

- The category of each word is represented by a tree structure. Phrase structure rules and argument structures are not separately defined in grammar and lexicon but combined. More details about the linguistic interpretation of lexicalized TAG trees can be found in the next section.

- Lexical entries are duplicated if they refer to different argument structures. This corresponds to a semantic differentiation. In this way lexicalized TAG trees represent semantic and syntactic units.

### The Use of LTAGs

The possible applications of lexicalized TAGs in an incremental syntactic generator and the use of the different forms of trees are discussed in the following.

Recall the problem described in Section 2.3: The size of trees should correspond to the information associated with single input lexemes. The problem with Tree Adjoining Grammars is, that each tree can consist of several phrases (and this will indeed often be the case because of the linguistic constraint). Figure 11 shows the tree for an intransitive verb which not only includes the verbal phrase but also the nominal phrase of its subject.

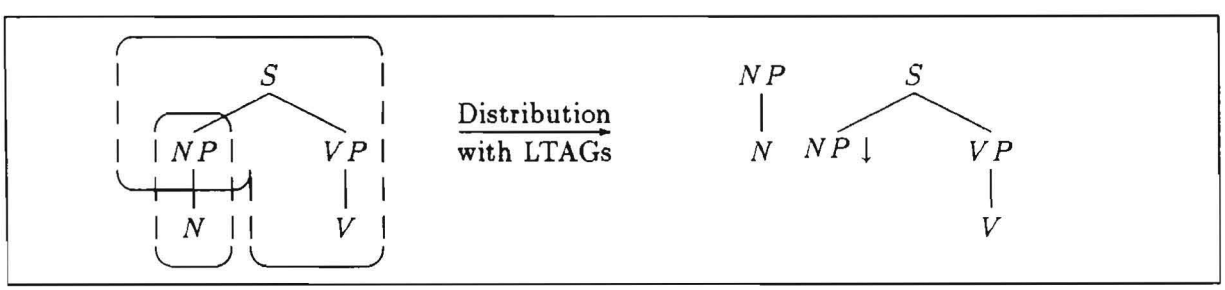


Figure 11: Distribution of Elementary Trees by Use of Substitution Nodes

If this tree is chosen on the basis of a verbal lexeme or even a noun, then decisions must be made about details of the other phrase. Lexicalized TAGs allow for a distribution of trees by using substitution nodes. The size of trees can be reduced so that each tree in fact describes the phrase of its head lexeme. The substitution nodes serve as stand-ins

for obligatory subtrees – which can later be filled in – and thereby preserve the linguistic constraint. The sentential tree on the right side of Figure 11 represents the subject as an obligatory part as before.

Lexicalized TAGs can be used for a more 'deterministic' choice of trees: There are three (instead of two) types of elementary trees that can be used for special linguistic reasons.

### Initial S-type Trees

An initial S-type tree is the kernel of each generated sentence. The head of the sentence often is the verb, but there can be specific predicative nouns or adjectives with the same function. The left tree in Figure 12 shows an initial S-type tree with a verbal head.

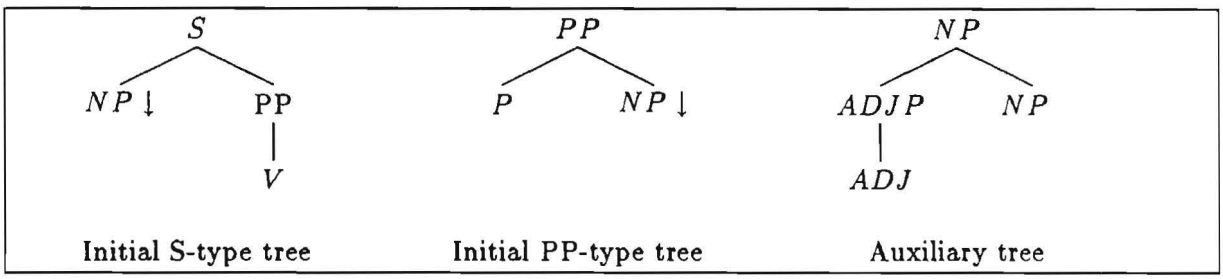


Figure 12: Linguistic Classification of Elementary Trees of LTAG

According to the linguistic constraint, each tree must contain all *complements* (see [Engel 77]) of its head. Verbs can define 'stand-ins' in the sentence that have to be filled by their complements in order to create a grammatically correct sentence. E.g., the verb "to praise" must be combined with an accusative object. The left tree in Figure 12 could, e.g., represent a verb like "to work" which only needs a subject as its complement.

Stand-ins are defined by substitution nodes which must be replaced by initial X-type trees.

### Initial X-type Trees

Initial X-type trees are partial trees that have to replace substitution nodes in order to create a complete derivation tree. They represent arguments or complements. The label of their root reflects the represented phrase, i.e., the projection of the category of their lexial head. An initial X-type tree always defines the maximal projection (i.e., again a complete argument structure) of this category. Substitution nodes can not only be defined in initial S-type trees but also in X-type trees and auxiliary trees. The tree in the middle of Figure 12 shows an initial PP-type tree representing a prepositional phrase.

### Auxiliary Trees

Auxiliary trees can be used to realize adjuncts. Adjuncts (see [Tesnière 59]) can depend on all elements of a word class and are in principle optional. The fact that adjuncts can modify all elements of a word class is reflected in the structure of auxiliary trees: Root and foot node define the category of the modifiable node. Adjuncts modify other phrases and can only be used and moved in connection with those. Auxiliary trees guarantee these effects: If they have replaced a node they follow all its movements (during linearization) in the tree. The right tree in Figure 12 realizes a modifier for a noun.

Lexicalized TAGs can realize all structures that can be realized by use of 'normal' TAGs. They support the representation of predicates, complements and adjuncts in such a way that incremental processing is possible. Nevertheless, another demand on the form of elementary trees must be formulated. In the examples shown above, the labels of the substitution nodes define the respective realization of all complements. In the left tree in Figure 12, NP↓ represents the subject, but this could also be realized by a subordinate clause or something else ("That you call, pleases me"). One simple solution consists in using the same label 'A' for all complements and specifying possible realizations by feature structures. The grammar can be made even less redundant if we allow for the underspecification of root nodes of substitution trees, adding the respective features as the incremental input specifies the relation of the tree to its regent. The left tree in Figure 13 describes a transitive verb as the head of a sentence. Both arguments are represented as substitution nodes with label A, they are differentiated by the associated feature 'func'. All phrases that serve as constituents for their regent are specified in an

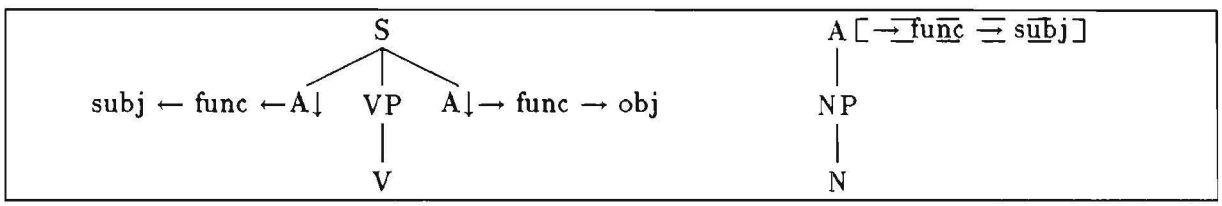


Figure 13: Functional Symbols in TAG Trees

A-type substitution tree (as the NP-tree in the right of Figure 13). After the decision about their exact functional relation to their regent is made this information is associated with the root node and serves as identifier for the substitution node.

This design concept is not only a means to reduce redundancy but also helps to improve the incremental choice of trees.

The concept of parallel computation that has been motivated in Section 2.4 requires a segmentation of the syntactic structures into parts that can be handled by single active objects as independently as possible. The three types of trees within lexicalized TAGs and their linguistic characterization eases the segmentation, as will be shown in the following.

### Segmentation with LTAGs

Lexicalized TAGs make available elementary structures and combination operations suitable for incremental natural language generation (see Section 3.5). Elementary trees of the form described above have an adequate size and can be chosen incrementally with less more than local information. Therefore, they are used as the segments of syntactic knowledge that are managed by single objects. This approach has a lot of advantages:

- The segments correspond to elements of the grammar. This way, existing computation procedures can be preserved.
- The size of the segments has structural advantages that are grounded in their domain of locality.

- The used definition of TAGs with unification allows for further kinds of control. The feature structures allow for direct tests during lexical insertion or adjunction.

- Because of its lexicalization, each tree is associated with its lexical head. This allows for incremental and parallel processing of the segments.

The disadvantages of this segmentation have to do with the size of trees. The grammar is redundant and the choice of a tree means the choice of a relatively big structure. But these problems seem to be less strong than for 'normal TAGs', as the trees of the lexicalized TAGs are kept small because of the substitution nodes.

The transfer from the hierarchical to the positional level must include the trees themselves because they are the basis for the definition of LP-rules and they represent the correct relations between the lexemes managed by distinct objects. This means a nearly complete transfer of structures that can be modified at the hierarchical level even after this transfer. These modifications would have to be simulated at the positional level, too. For this reason, hierarchical and positional level are not realized as modules but their distinction is defined inside the programs of the objects: If a subtree of an object at the hierarchical level is complete and integrated into the global syntactic structure, the object changes its state and tries to fulfill the tasks of the positional level. Although there is no distinction of data and program, the objects are called in the following 'objects at the hierarchical level' and 'objects at the positional level'. These terms refer to the specific state of the objects. The use of the same data helps avoiding redundancy.

With respect to the criticism of LD/LP-TAGs (see Section 3.4) the following has to be stated for the design of the positional level: The presented concepts are based on the possibilities given by Tree Adjoining Grammar and its extensions. A more flexible and efficient processing of word position could be realized by another definition of LP rules, e.g., by a real transformation of hierarchical into other positional structures corresponding more adequately to word order phenomena of natural language. An interesting approach is described by [DeSmedt 90].

## 3.6 The Relevance of TAGs for Generation

In order to compute a grammatically correct utterance, there must be a grammatical component somewhere in the generator. This grammatical component that designs syntactically correct sentences must systematically interact with the planner that decides what is to be said. The interface should allow for incremental generation. For this reason there can be a set of demands on the grammar concerning the efficient processing of the generator and a fast and incremental output.

Incremental generation on the syntactic level means that incoming elements are to be inserted into the syntactic representation structure as fast as possible, and that parts of this structure can be found which represent partial input for the next level. According to this demand, the size of grammar rules and the kinds of combination operations have to be discussed. [DeSmedt & Kempen 87] demand three kinds of syntactic expansion:

**Upward Expansion:** Upward expansion means to insert an existing partial tree into a bigger one, e.g., a nominal phrase as the subject of a sentence. Upward expansion can be realized with TAGs by adjunction of an auxiliary tree into the root of the

actual structure or by substituting the actual structure in a substitution node of the new tree.

**Downward Expansion:** Downward expansion of a node means its further specification on lower syntactic levels, i.e., its distribution into its components (e.g., a sentence into nominal and verbal phrase). Downward expansion can be realized by replacing a substitution node by a corresponding substitution tree.

**Insertion:** Insertion means to fit in new syntactic material between existing nodes. An example is the modification of a noun by a determiner. Insertions correspond to adjunctions at internal nodes in the TAG formalism.

Tree Adjoining Grammars allow for all three kinds of expansion. The incremental choice of conceptual structure and the incremental construction of the sentence can well be collated. It is essential that the syntactic structures encode linguistic knowledge in such a way that the incremental choice and processing of trees is supported.

The four most important properties of elementary TAG trees which are additional motivations for their use in a syntactic generator result from their extended domain of locality:

1. Many (linguistically relevant) relations between nodes can be defined locally.
2. Properties can be testet locally (e.g., agreement).
3. The argument structure can be defined locally (and is demanded as the linguistic constraint for the design of TAG trees).
4. The argument structure is preserved during adjunction.

These properties make TAG trees adequate structures for the construction of the syntactic representation of a sentence. They are defined as linguistically minimal units and are therefore usable for incremental processing.

The realization of this style of processing is explained in the next sections which deal with the two levels of computation.

# 4 The Hierarchical Level

This section describes the co-operation of objects at the hierarchical level. They construct a complete sentence tree and transfer partial structures to the positional level. The monitor plays a central part at the hierarchical level as it controls the co-operation of all objects during the incremental construction of the sentence tree. The next sections will deal with the different abilities of the objects.

## 4.1 The Creation of Objects

Each object at the hierarchical level is responsible for the processing of one lexeme. This is a consequence of the principle of *lexical guidance* and leads to a clear and unique distribution of tasks over the objects.

Incremental processing at the hierarchical level is primarily useful if the input to the level is also given in a piecemeal fashion. For each lexeme that is created by the component for lexical choice, a single object is created at the hierarchical level. It manages the further processing of the lexeme and needs some more information for doing so. Parts of this processing are the choice of a lexicalized TAG tree whose head the lexeme will be, and the choice of a feature structure that will be associated with the preterminal leaf of the tree, thereby representing the special features of the head.

The obligatory information that is used for the choice of feature structure and TAG tree consists of the lexeme, its category, some semantic information for the differentiation between readings, the function of the word in the sentence under construction, and some morphological information used for the inflection of the lexeme. Knowledge about the function is needed to choose an adequate TAG tree (see Section 3.5): Initial trees represent the predicate of a sentence or complements, auxiliary trees represent adjuncts. The result of this choice process is not a single tree but a set of trees representing equivalent structural alternatives for the realization of the input (e.g., a modifier can be realized by an adjective phrase or a relative clause). The set is called ATS (alternative tree set).

Apart from the information guiding the choice in lexicon and grammar, each object receives a unique identifier, that helps to initiate communication with other objects (see Section 2.4). Additionally, each object gets the identifier of its regent in the sentence which is (or will be) represented by another single object. This knowledge will be used during all activities of the object because of a basic principle, that will be motivated later (see Section 4.3): Each activity is initiated by the dependent that tries to combine its structure with the structure of its regent in order to complete the sentence tree.

The different parts of input information can be encoded in two groups: Information describing the entity itself is marked with the name 'entity'. A unique identifier must be given to refer to the object. It can be followed by a list of pairs, each specifying a part of information by a keyword and a value. An example is given on the left side in Figure 14. These pairs need not be given in one compact package, the object can be incrementally supplied with input.

Information describing the relation of an object to another one is marked with the name 'relation'. It must contain the two relevant identifiers, another unique identifier for the relation between them, and – in some cases – a further specification of this relation. This specification can again be given as keyword-value pairs. An example for a specified

```
(ENTITY VP-1    ((HEAD "put")          (RELATION VP-1 MOD-VP-1 NP-1)
                 (VALENCE V3)
                 (TENSE present)
                 (CAT V)
```

Figure 14: Input Entities for ISGT

relation is shown on the right side of Figure 14.

## 4.2 The Initialization Phase

The first thing an object does, is to carry out the choice process on the basis of the given obligatory information (as soon as it is complete). The result is a specification list for the feature structure and an ATS. The specification list is enlarged by the given morphological information.

In order to start its work at the hierarchical level, the object must choose one tree among the ATS. Up to now, this is done with respect to the possible combinations of trees at the hierarchical level. For further expansions it seems possible to directly influence this choice, e.g., if a specific syntactic style is to be realized.

### Demands on the Choice in an ATS

The most important decision points for the choice among the trees in an ATS are listed in form of questions:

1. *How are alternatives handled?* In the system ISGT, just one alternative is to be chosen and processed furtheron. A motivation for this approach is, that this form of processing is more efficient in the case of success than the processing of alternatives, and that redundant work is avoided.

2. *Which information should influence the choice?* The choice in an ATS should be made on the basis of *local* information of the object and *global* information that characterizes its relation to other objects. The local information includes, e.g.,

   - the syntactic alternatives themselves, as they have been chosen on the basis of some local informations, and
   - the history of previously chosen structures, in order not to loose these experiences.

   The global information guarantees, that the chosen tree fits in the actual structure. It is helpful to know

   - the structures belonging to other objects (most interesting are the structures of regent and dependents),
   - the number of repetitions of a specific syntactic form in order to avoid monotonuous formulations,

- the actual 'active' or 'used' form, for humans often use a specific syntactic form several times successively, and
- information about the degree of 'simplicity' of the syntactic alternatives, so that among the set of alternative trees the most simple one can directly be chosen (as senseful default).

3. *Where is this information stored?* Local information should be stored within the respective objects themselves. Furthermore, it is obvious that the number of repetitions or the determination of the actual form can best be controlled, if the different ATS structures are stored globally.

   Global information is stored within the monitor object. The monitor observes changes and enlargements of the structure tree and can be asked for information by the single objects. This approach is well suited for a first realization of the system, because it can be developped modularily.

The next section deals with the question which information the monitor demands from the single objects in order to describe the global state at the hierarchical level, and how this state is represented.

### Representation of the Global Structure

In order to represent the actual situation at the hierarchical level, the monitor has to be informed about existency, state, and relation of all objects. The following data must be part of the global structure:

1. **Address:** The monitor needs an object's address in order to communicate with it. The monitor transfers this address to all potential partners for communication.

2. **Unique Identifier:** The objects' unique identifiers are used to recognize communication partners as they are given as patterns with the input.

3. **Regent:** The monitor must know the identifier or address of an object's regent in order to correctly simulate its integration into the sentence structure.

4. **Tree:** The tree managed by an object is the central part of the global structure. The sentence structure is to be constructed incrementally from the different trees at the hierarchical level.

5. **Used Nodes:** The combination of the trees is represented by marking which nodes of the trees have been used for which combination operation, and which other object has been integrated.

6. **Goal Object:** One problem of control is the adjunction of several auxiliary trees into the same node (e.g., several modifiers for the same noun). Only one auxiliary tree can be adjoined directly into the node of adjunction, all the others must be adjoined into the respective part of the modified tree (an integrated auxiliary tree). The address of the object which has sent that tree is stored.

There are two important factors for the choice of trees by the monitor: Firstly, the action takes place during the initialization phase of the object and is in fact the first action at all. Secondly, the monitor must integrate the object into the global structure in order to get as much information as possible for the tree choice. That is why, the object communicates with the monitor before it chooses its tree.

### Construction of the Global Structure

For the first communication with the monitor, the object must hand over its address, its unique identifier, and the identifier of its regent. These data are stored in the global structure, where the object's address is used as the key. Furthermore, the given information is used as far as possible to mediate between the communication partners. The monitor uses two local variables to manage this task:

**search-regent** stores the unique identifier and the address of each object. It helps to find the address of an object when only its identifier is known (e.g., each time when an object looks for its regent).

**search-deps** associates identifiers with a list of addresses of objects looking for a regent with the respective identifier. Each object that cannot identify its regent during its registration at the monitor – because it was earlier created than the regent – is stored in this variable. For each new object the monitor tests, if its identifier can be found here and sends the respective messages to all waiting objects (the dependents of the new object).

The monitor also informs the regent when new dependents are created at the hierarchical level. The regents can use this information, e.g., to wait for delayed objects. It is important to know if the dependents manage an auxiliary or an initial tree, i.e., if they will be adjoined or substituted (for the motivation see Section 4.3). Address and type of the tree are stored in the variable *context* of the regent.

The following steps are made during the registration of a new object: First, an entry is created under the object's address. It is stored together with the identifier in *search-regent*. Then the monitor looks in *search-regent* for the unique identifier of the object's regent. If it can be found, it is integrated into the entry in the global structure. If the address could not be found, the object is stored in *search-deps*, together with the identifier of its regent. Finally, the monitor searches for dependents waiting for the actual object as their regent in *search-deps* and informs them in case of success.

The monitor returns to the registrated object a list of all known dependents and its regent. The tree choice is modelled as a distinct process because it can be made several times, whereas the registration of an object is made just once. It will be described in the next section.

### Tree Choice with the Monitor

Each object must send some local information to the monitor that tries to choose an adequate tree. Relevant data are the ATS and the history of previously tried trees (abbreviated as TTS for *tried tree set*) that is empty during the initialization phase. TTS can

be used to prevent the system from doing the same mistake twice. Also, a specification of the syntactic form is useful here.
The algorithm for the tree choice is roughly described in the following:

1. One tree is chosen according to the given local information of the object. It is called *temporary-tree*. It is a time consuming task to compute all possible trees that could be integrated into the overall structure. This is why, the monitor just chooses one tree – with very simple means – and tests, if this tree can be integrated. During this choice the monitor follows the constraints given by repetition number, active forms and so on (see Section 4.2).

2. If *temporary-tree* is found and is not an element of TTS, then go to Step 3, else to Step 4.

3. For auxiliary trees, it is roughly tested if the precomputed tree can be integrated into the global structure. If the object's regent is known, the monitor searches for a node in its tree where *temporary-tree* might sensefully be adjoined. Obviously, this test can be combined with the computation of the *goal node*, i.e., the node of adjunction for the tree. If there are several candidates for this role, it should be possible to use further information to restrict the possibilities, e.g., the state of linearization, or the current style. In our prototype, the monitor chooses the deepest node (the node that is as far away from the root as possible) as a default. It is presupposed that the grammar designer restricts all necessary structural combinations with help of constraints.

   The only problem with the computation of the goal node again is the case that two auxiliary trees need to be adjoined into the same node of their regent tree. One must be the first, the second can be adjoined into this one, and so on. These interlocking combinations must be synchronized. In order to keep the global structure consistent and to compute new combinations on the basis of previous ones, the monitor of ISGT fixes the chosen tree and its relation (goal node) to the regent, even before adjunction (or substitution) has really taken place. For interlocking adjunctions this has the consequence that the order in which the goals were computed by the monitor must be kept during the real combination. Otherwise, the goal nodes would have to be computed again. The basic principle for this approach can be formulated like this: *From a set of concurrent auxiliary trees (that need to be adjoined into the same node) that tree is adjoined first, which was first registered at the monitor, i.e., which corresponds to the earliest given input.*

   The test for possible integration of a tree into the global structure is subdivided into the following steps:

   a. If the object's regent is known, look in its tree for an adjoining or substitution node.

   b. If the regent is known but no goal node could be found, then go back to step 2.

   c. Insert *temporary-tree* into the entry of the actual object in the global structure.

   d. Return all found structures to the registered object and terminate the tree choice.

4. Return NIL.

It becomes clear that an object's initialization phase largely consists of communication with the monitor. In the best case, it receives the addresses of its regent and its dependents, the tree that is to realize the represented lexeme, and the goal node in the regent's tree, in which the local tree shall be adjoined. Each object stores these results in its local variables.

## 4.3 Construction of the Sentence Tree

The goal at the hierarchical level is the construction of a complex sentence tree from the single trees that are ruled by the objects. In order to reach that goal, the objects must co-operate and communicate. It seems convenient to define a certain 'direction of activity'. If it becomes a principle that either the regent initiates the communication with its dependents or the other way round, the distribution of tasks and responsibilities becomes clear. A 'free communication' would be much harder to handle.

In ISGT, each dependent is responsible for offering its local data to the regent at the appropriate time. If the regent was responsible for those activities, then it would have to query its dependents several times in order to find out if they are ready to integrate their partial structures into the global tree. The costs for communication would be higher. It remains to be described what the appropriate time to communicate with a regent is. According to [Neumann 89], the term 'completeness' is used here.

### The Completeness of Dependents

The test for completeness is made for trees that are to be combined with other ones by adjunction or substitution. Their completeness should prevent them from too early bindings, for otherwise they would have to hand some data to their regents later which in the worst case could lead to failure. What does completeness mean for initial and auxiliary trees?

1. The feature structure of the head must be complete in the sense that it contains no path which is not ended by a value. This condition is necessary to guarantee that the relevant information is complete when the object tries to integrate its tree into the global structure.

2. Initial X-type trees are integrated into the global structure by unifying the feature structure of their root node with the feature structure of the substitution node of their regent. Later adjunctions or substitutions in the dependent trees do not influence the regent tree, except when the feature structure of the root is changed. In this case, the newly constructed feature structure of the root must be sent to the regent tree. The change can be non-monotonic (see Section 3.3), i.e., the feature structure is not purely enlarged but modified in another way. In this case the transfer to the regent tree becomes more difficult, because the two 'interface-feature structures' can not simply be unified. It might be sensible not to declare a substitution tree complete before all its known dependents have been adjoined into it. Substitutions can be left aside, because they only lead to monotonic changes. Up to

how it could not be estimated, how often those late changes of feature structures of roots can take place. It will be tested by using several different completeness tests in the prototype.

3. In the actual version of the generator, auxiliary trees are structurally embedded in their regent trees – adjunction will be realized as a 'parallel' operation within the next extension of the prototype. A late adjunction or substitution in an auxiliary tree therefore means a structural change of their regent tree. There are different definitions for the completeness of auxiliary trees that have to be tested in the prototype (e.g., *all substitution nodes must be filled*, ...).

This definition is the presupposition for defining the combination operations themselves. As adjunction and substitution imply different demands and different actions, they are discussed in two distinct sections.

### Substitution at the Hierarchical Level

**Identification of the Substitution Node:** It is very easy to identify the respective substitution node for a given tree: Since most substitution nodes in the regent tree will be labeled with 'A', the feature structure of the root node of the substitution tree is compared with the feature structures of all substitution nodes. The first node whose feature structure is compatible with that of the root node is chosen as goal node. Both objects store the identified node in order to use it as interface for the further flow of information.

**Substitution:** The trees at the hierarchical level are distributed over different objects. They can only be combined by sending messages between the objects. It would be possible to send whole trees to the respective regent object, but this would concentrate the structure more and more in one object (the highest regent in the hierarchy of the sentence) and would reduce parallelism. How can the trees remain distributed over the objects and be combined all the same? This question is answered by the introduction of TAGs with substitution (see Section 2.4). The substitution node can be seen as an 'interface' between regent tree and substitution tree, and between the two objects as well. There is no structural change inside but rather at the edge of the regent tree. A continuous separation of the resulting tree means the duplication of the substitution node, but in the regent tree it is only a stand-in for the substitution tree: It is associated with the feature structure that results from the unification of (copies of) the feature structures from root and substitution node.

Additionally, it is associated with a pointer to the dependent object, thereby defining a quasi-structural relation that remains uncomplicated because adjunctions are forbidden in substitution nodes. All modifications of the substitution node must take place in the dependent object and are sent (if necessary) to the regent.

**Information Flow over the Substitution Node:** If the feature structure of the root node of a substitution tree is changed, the new information must be sent to the regent tree. Those changes can be caused by adjunctions or substitutions in the object, or by

an information flow through its own substitution nodes. If this happens, the object must communicate with all partners connected with it via substitution nodes (regents as well as dependents), if the respective 'interface feature structure' has been changed.

A newly sent feature structure can directly be unified with the feature structure of the goal node if either this is a substitution call or the new feature structure comes from an object where it has also been created by direct unification. This simple kind of processing is guaranteed only if the respective feature structure has been changed by a monotonic operation (unification). The non-monotonic adjunction (see Section 3.3) can modifiy feature structures in a way that they are no more compatible with those feature structures which existed before the adjunction took place. These kinds of feature structures cannot simply be unified with their partners, because this could lead to a failure. They must be 'set' instead of unified. This must happen after adjunction, but also if the feature structure comes from an object where another one is set and has thereby caused the change. This is why, the sending of new feature structures to or from substitution nodes must be associated with information about their genesis as can be seen in Figure 15

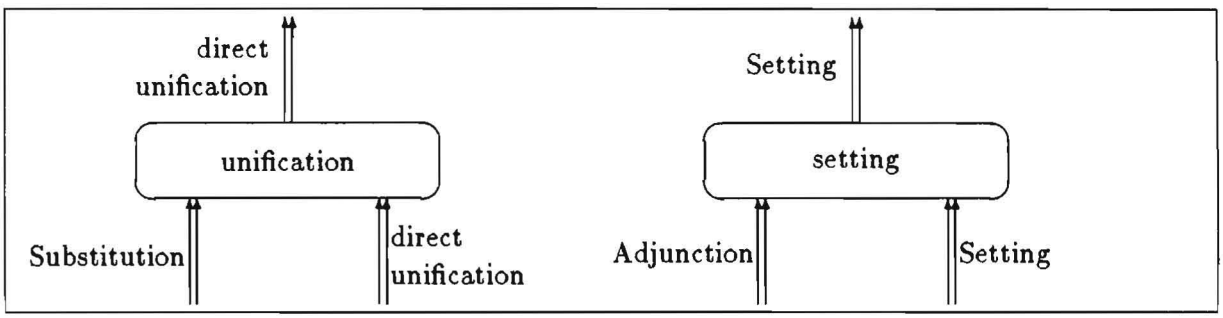


Figure 15: 'Types' of Information Flow

### Adjunction in the Hierarchical Level

**Identification of the node of adjunction:** Tree choice for an object is carried out either during its initialization phase or directly before its combination with the regent object. At the same time, a complex description of the goal node is computed which, e.g., says if the auxiliary tree is to be adjoined in a node of the original regent tree or in a node of another auxiliary tree that has been integrated into the regent before.

**Adjunction:** Substitution trees remain separated in spite of the unification of the feature structure of their root with the feature structure of the substitution node. This is possible, because the substitution tree is not inserted into the regent tree but associated with it at one edge. If auxiliary trees should also be kept separated, the flow of information through their root and foot nodes would have to be simulated – a very complex and costly task. Consequently, adjunction is handled in a different way than substitution. Auxiliary trees are fully handed over to the regent object. The dependent object starts to sleep because it has transferred all its responsibilities to its regent. It must not be terminated for two reasons: (a) In the case of backtracking it could be possible that the adjunction must be taken back and the original state must be created again, or (b) there could be

late messages from its own dependents. The number of these messages can be restricted by a more or less strong definition of the completeness condition (see Section 4.3). They concern substitutions, adjunctions, and the flow of information via substitution nodes.

In connection with adjunction and backtracking, the idea of a 'history of trees' in the objects has come up. The result of each adjunction will be integrated into this history. This allows for taking back adjunctions step by step during backtracking.

## 4.4 Transfer to the Positional Level

Recall the results from Section 2.4: The positional level is not realized as a distinct level with new objects. The objects of the hierarchical level change their state, if they fulfill some specific demands, which have to be defined in the following. Then they try to manage the tasks of the positional level. There is no 'real' transfer of TAG trees to structures defining positional relations. Instead, the already existing trees are associated with their LP rules.

The central question that will be discussed in this section is which conditions an object must meet, before it is allowed to change to the positional level. The conditions are called *linearization conditions.*

### Linearization Conditions for the Transfer to the Positional Level

Not all objects at the hierarchical level are transferred to the positional level but only those managing an initial tree. All objects with auxiliary trees have started to sleep sometime before – their regents are responsible for the further computation of their structures. This also holds for linearization.

The following conditions seem to be senseful preconditions for the transfer of objects with initial trees to the positional level:

1. Initial trees must have been substituted. As LP rules define relations (not absolute positions) for elements of TAG trees, the position of a word in the whole sentence can only be determined when looking at the global structure from its root. Therefore, each object must be able to reach the absolute regent of the sentence, at least its own direct regent. For further expansions of the system, you can think of a more variable management, e.g., default positions for subjects or other means (see Section 6).

   Substitution itself is bound by completeness conditions. They have to be tested again in order to decide if they delay linearization too much. This discussion is described in the next section. Anyhow, the completeness of the feature structure of the head remains the minimal presupposition for the integration of a substitution tree into the global structure.

2. All dependents managing an auxiliary tree that are known to the object must have been adjoined. The reason is, that they can influence the order of the terminals in an unpredictable way.

3. In contrast to this, the object does not have to wait for dependents with initial trees. An object whose tree contains unfilled substitution leaves may nevertheless change to the positional level. The positional relation of the substitution nodes to the rest

of the tree reflects the position of the whole subtree which is expected to substitute the node. This subtree needs not be integrated in order to linearize the supertree.

Since linearization and completeness conditions are closely related, they will be compared in the following section.

### The Relation of Linearization and Completeness Conditions

Although objects with auxiliary trees are not transferred to the positional level, the time of their completeness and their integration into their regent is relevant for the transfer of those regents. This is why, completeness conditions of both auxiliary and initial trees are compared in this section:

**Initial Trees:** The linearization conditions for initial trees contain their completeness conditions, but vary with the realized alternatives. If the completeness conditions did not include the test for adjunction of all known dependents with auxiliary trees, then it must be carried out additionally. This takes place after the substitution and – if necessary – after each further change of the object. If the test is part of the completeness conditions then the permission for substitution is at the same time a permission for the transfer to the positional level.

**Auxiliary Trees:** If an object with an initial tree knows that one of its dependents manages an auxiliary tree and has not yet been adjoined, then it must not linearize. That is why, completeness conditions of auxiliary trees can delay the linearization of initial trees (their regents) by delaying their adjunction. On the basis of these considerations, it seems more useful to define a less strong completion test for auxiliary trees: They should not be forced to wait for all substitutions before they are allowed to be adjoined in the regent tree.

# 5 The Positional Level

Objects changing from the hierarchical to the positional level exchange their goal for a new one: Their local structure must now be linearized, i.e., ordered according to the given LP rules, the lexemes must be inflected and uttered. Therefore, the change to the positional level can be understood as an object's readiness for the verbalization of its own lexemes. The objects need a minimum of information to do that. They have been introduced as linearization conditions which are tested before the transfer.

The mixing of objects of the hierarchical and the positional level, whose communication links are preserved, has the consequence that many calls which are specific for one of the two levels must be realized for both levels. An object at the positional level must be able to handle all information it receives from objects of the hierarchical level after its transfer. This concerns all late substitutions, the managing of information passing through 'interface nodes', and the adjunction of latecomers. These late structural modifications normally do not influence the fulfillment of the linearization conditions. Especially with respect to adjunction it must be noted that the linearization condition demands only the adjunction of all actually *known* dependents with auxiliary trees.

The objects at the hierarchical level must also be able to react to calls from the positional level. Each object that wants to utter its partial structure must know its position in the global sentence tree. This can be computed by stepwise calls up to the highest regent. These calls also concern objects which are still at the hierarchical level.

In the following, the methods specific to the positional level will be introduced. Linearization and inflection which must be managed by objects at the positional level can in principle be handled locally. The leaves of the managed tree must be ordered in a way that is allowed by the associated LP rules. After that, the lexemes are inflected and uttered. These processes become complicated, because the objects just deal with parts of a global structure, other parts can be created incrementally, new objects can change from the hierarchical to the positional level and can work in parallel.

Since all subtrees are distributed over distinct objects, this also holds for the associated LP rules. That is why, an object can not locally decide which position its subtree has in the global structure. It must get this information by communication with higher objects. This process is called *output call* here and is described in more detail in the next subsection. The output call is the first action of an object at the positional level: Without permission 'from above', the terminals of the local tree must not be uttered. The time that passes until the answer to that output call is received can be used for a *first linearization and inflection phase*. During this phase the leaves of the local tree are ordered as far as possible according to the given LP rules, the lexemes are inflected and stored, as they must not yet be uttered. The phase is called *first* linearization phase because the local tree possibly can only be traversed partially. Substitution nodes can act as 'breaks' if they lead to subtrees which have not yet been uttered. If the object receives a message from its regent allowing for output the prepared sequence of words is uttered and the output of the previously 'breaking' subtree is initiated. After this subtree has been completely uttered, there can be *further linearization phases*. They differ from the first linearization phase in the fact, that the reached words can now be inflected and uttered immediately, as the permission for output has already been given. Again, these

phases can be interrupted by substitution nodes.

The communication with other objects is not the only reason for the complexity of processing at the positional level. There must also be plans to handle exceptions like late adjunction in objects that already have linearized and/or uttered their trees, or backtracking at the hierarchical level which also influences objects at the positional level. Some of these cases can be integrated into the overall processing without too much difficulties, but for others it seems necessary to design new strategies.

## 5.1 The Output Call

Objects at the positional level want to linearize their local trees and inflect their leaves with the goal to utter them. The actual utterance must not start before the sentence part can be uttered within the global structure. Output phases naturally must not take place in different objects in parallel. Nevertheless, the possibility to work in parallel can be used to process linearization and inflection locally in the objects, even if output is not allowed.

All actions at the positional level – output call, linearization, inflection, and utterance – are not useful before at least one lexeme can locally be positioned in the first place. This condition is called *output condition.* Even if it is fulfilled, the output can not start before it is allowed by the regent. The time that passes while an object waits for the answer can be used to linearize and inflect the local tree as far as possible. This is why, the first action at the positional level is the sending of the output call (after the output condition is fulfilled) to the regent. After that, the first linearization phase can start.

### Storing the Output State

Each object at the positional level fulfilling the output condition asks its regent, whether it is allowed to utter its part of the sentence. The relation to the regent always consists of a substitution node. The regent tests its local structure in order to decide whether the dependent may start uttering at this time.

In order to correctly answer the output call, the regent must know, which dependents have already finished or just begun their output, and how the LP relations of the single nodes look like. The state of the subtrees must be stored within the respective objects for the following reasons:

- Objects must be able to answer an output call, i.e., to decide whether parts of their tree may be uttered.
- In order to know the current global output state (compare with the description of the problem of consistency in Section 5.1), all objects which send a positive answer to one of their dependents must store that the respective object now begins with its output.
- New states are also stored within those objects which are themselves still at the hierarchical level, for they must possibly deal with output calls. In these cases, the states are not used for the objects themselves but only for the calls from their communication partners.

The LD structures can very easily be associated with LP rules (see Section 3.4). There are several possibilities to realize this association. The definition can be transferred literally and then leads to <-relations with references to node numbers. This kind of definition has two disadvantages: During the traversal of a tree for its linearization, the rules must be interpreted globally. Furthermore, an adjunction leads to the combination of two sets of LP rules whose node numbers must be adapted to the new tree structure. A more favourable solution is the following realization of LP rules: A rule 'X < Y' may be interpreted as 'Y has to wait for X during linearization'. This suggests the idea to associate T with all nodes which have to be uttered before it. Since this only concerns sisters, the information remains locally and unchanged even during adjunction. In Figure 16, all sisters of Z which have to be uttered before it are associated with the edge from Z to its father. Figure 17 shows that these sets are not changed during adjunction.

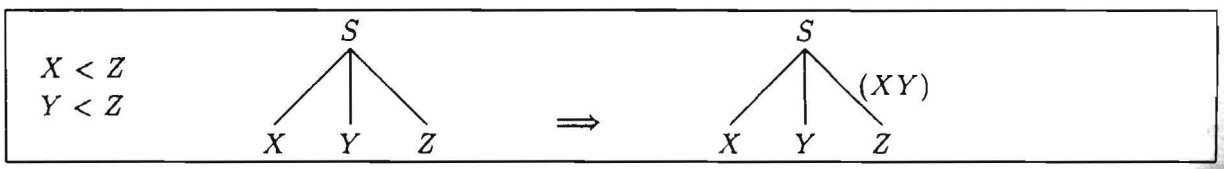


Figure 16: The Representation of LP Rules in the Objects

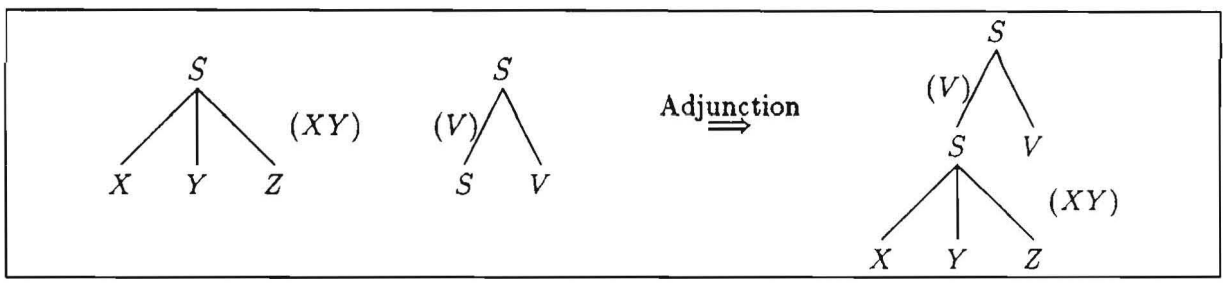


Figure 17: Consequences of Adjunction on the Internal Representation of LP Rules

In the next paragraph it will be examined, how it can be stored within the trees whether subtrees have started or finished their output.

**Representation of Complete Output:** It must be stored whether a lexeme has been uttered as well as whether a complete subtree has been uttered. In both cases, the respective father nodes are stored in the local variable *output-string*, together with the uttered words. The node numbers can then be compared with the sets on the next traversed edges. In the example in Figure 18, the words below X and Z as well as all leaves of the subtree of Y have already been uttered. This is marked by boxes around the terminals or sequences of terminals. The three nodes are stored in *output-string*. The same is done for W, for its complete subtree (consisting of X and Y) has been uttered. On the basis of the entries in *output-string*, it can now be computed that the word below A can be uttered next: All nodes in the set on the edge to A can be found in *output-string*.

The list of nodes associated with an edge of a tree only represents the *necessary* condition for linearization. Additionally, another condition must be fulfilled: Even if this list is empty at one edge, it is possible that linearization has just begun in a neighbouring

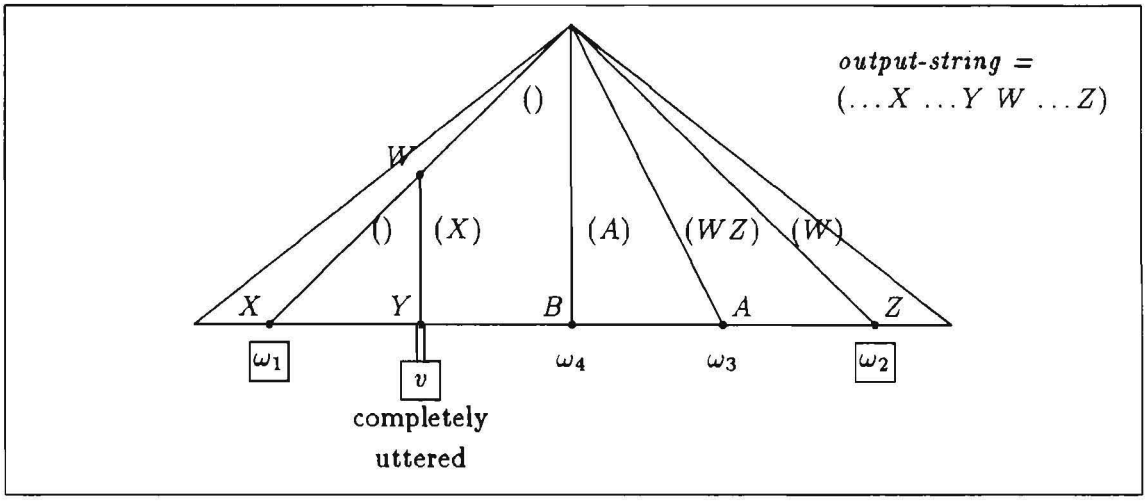


Figure 18: Representation of Complete Output

subtree and is not yet finished there. So the edge must not be traversed before the processing of the other subtree is completed.

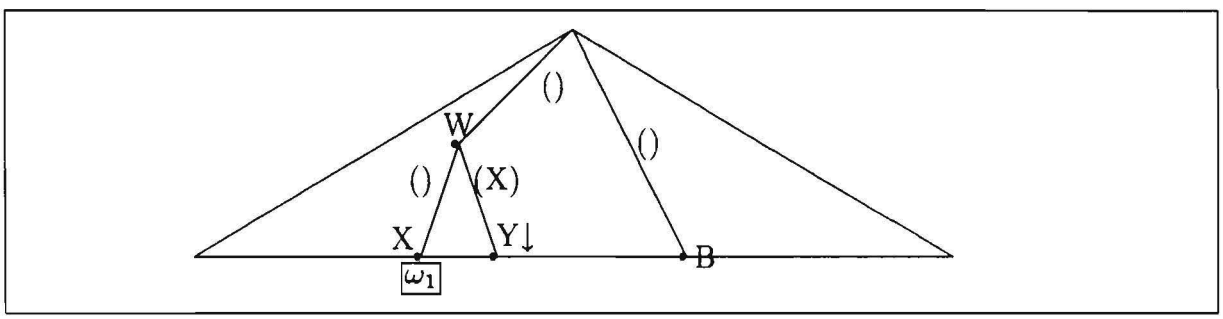


Figure 19: Linearization of Subtrees

Figure 19 shows that the edge to node B could theoretically be traversed because the associated list is empty. But output has already begun in the subtree under W (with the terminal $\omega_1$) and is not yet finished because of Y↓.

**Representation of 'Uttering' Subtrees:** If a regent receives the message that one of its dependents has begun with the output but not yet finished it, this information must be stored. The reason is, that other subtrees must not utter their terminal strings even if they would not have to wait for the first one with respect to the LP rules, because output must be sequentialized. The easiest solution for this problem is another local variable *output-by* containing information about the question whether the object itself, one of its dependents, or its regent is actually occupied with the output. The variable can contain the following values: NIL (neither the object itself nor one of its dependents are actually or have already been occupied with the output), the own address (the object itself has started uttering), the address of a dependent (the dependent has started uttering), or the address of the regent (the object and all its dependents have completed their output and reported this to the regent). The computation of an answer for an output call is based on the first three possible values of *output-by*, the fourth is used during corrections of

the output (see Section 5.4). As can be seen in the next section, an output call can be handed over by several objects until one is found which is able to compute an answer. On the way back to the original questioner, each object stores the respective dependent in *output-by* if the answer is positive. Figure 20 shows, how $O_1$ sends an output call (in the picture O-C) to $O_2$ that is passed to $O_3$. $O_3$ computes a positive answer (O-R stands for output result), so in a first step $O_2$ and then $O_1$ are stored within their regents as the object which (or whose dependent) is now occupied with the output.

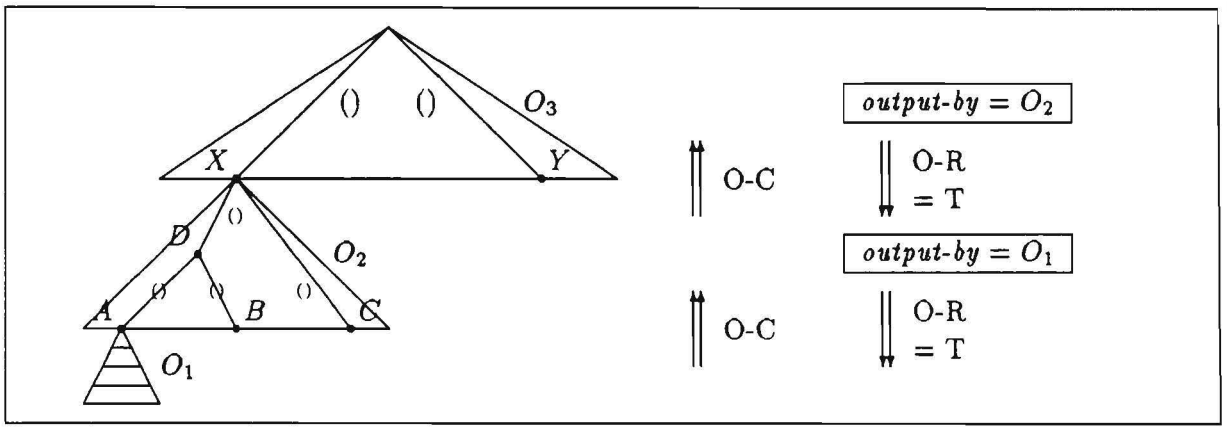


Figure 20: Representation of 'Uttering' Subtrees

The following section shows how an object answers an output call on the basis of its local information. If the local information is not sufficient, the object passes the call to its own regent.

### Answering the Output Call

Three situations can be identified in the called object which are represented in Figure 21. The position of the terminal string and the important substitution node always reflect

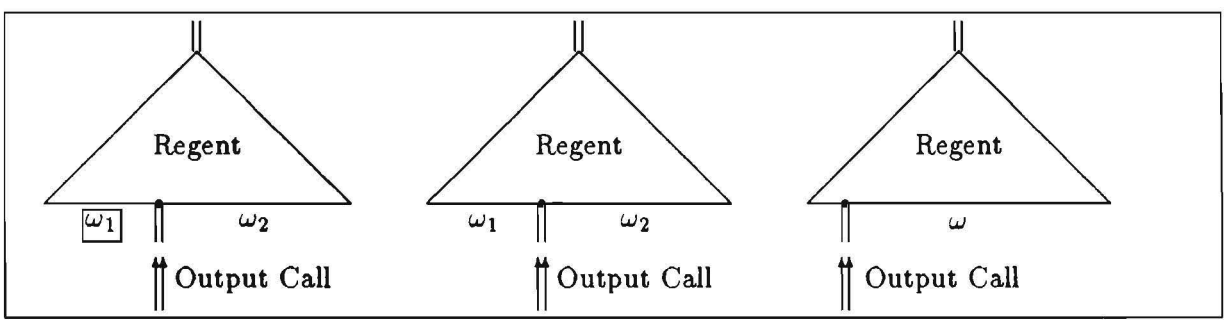


Figure 21: The Regent Answering the Output Call

that this node is placed as far left as possible with respect to the LP rules. The trees in this and the next figures always represent mobiles. Either they are shown in one allowed ordering for reasons of clarification, or are traversed according to these LP rules in another than the represented ordering. The reaction of the regent to the output call of its dependent depends on the state of its local tree:

1. 'To the left' of the respective substitution node is a terminal string $\omega_1$ that has already been uttered completely (see left tree in Figure 21). In this case, no further call to the regent's own regent is necessary: As the regent already has the permission for output, this is also valid for its subtrees and therefore for all its dependents. The answer now depends on the fact, which value is stored in *output-by*. If the value is not the regent's address, this means that another dependent is occupied with uttering and the answer must be 'NIL'. If the value is the address of the regent itself, it may allow for the output for the calling dependent (the answer is 'T'). At the same time, the address of the calling dependent is stored in *output-by*.

2. 'To the left' of the respective substitution node is a terminal string $\omega_1$ that has not yet been uttered (see tree in the middle of Figure 21). This situation can be found if a) the regent is still at the hierarchical level, or b) the regent is at the positional level but either does not fulfill its own output condition or waits for an answer to its own output call. This is reflected by the value 'NIL' in *output-by*. In both cases, the call of the dependent can immediately be answered with 'NIL'.

3. If the substitution node can be positioned in the first place of the sequence of leaves (see right tree in Figure 21), the output call is handled like this:

    a. The regent itself is the highest object of the regent-dependent hierarchy. Then it can decide about the output call. The decision depends on the value in *output-by* as in Point 1. If it is the address of a dependent, the actual question is refused. Otherwise, the answer is 'T' and the calling dependent is stored in *output-by*.

    b. The regent is not the highest object. If *output-by* is NIL (it is not known to the regent that another object is occupied with uttering), and the regent knows the address of its own regent, it forwards the output call. Otherwise, it cannot decide if the calling dependent may start the output and answers 'NIL'. Output will also not be allowed if another dependent is stored in *output-by*.

It becomes clear that an output call is probably sent over several regents before it can be uttered.

### Forwarding the Output Call

In order to return the answer to an output call to the originial questioner, the call must be associated with a list of 'callers'. The answer can then be handed over to the correct object by traversing this list in reverse direction.

During the downward transfer of the answer, it should have side effects if it is a 'NIL'. A dependent that wants to utter its local structure should not have to call its regent again and again, so it is stored within its regent in the variable *ready-for-output*. The same is done for all objects which have forwarded the call to their own regents. If an object is actually occupied with uttering and has traversed its local tree up to a substitution node, it looks for the respective dependent in *ready-for-output* and sends a permission, if it has found the address.

Figure 22 shows how two dependents of an object $R_1$ send output calls close on one another. The circled numbers define the order of actions. Think of a situation where the respective substitution nodes can be positioned so that both dependents could potentially utter their local string. If the regent can not locally decide if its dependents may utter, both output calls are forwarded. Thereby one of the two dependents must be handled first. As the call from $O_1$ has reached the highest object first (and $R_1$ is allowed to utter

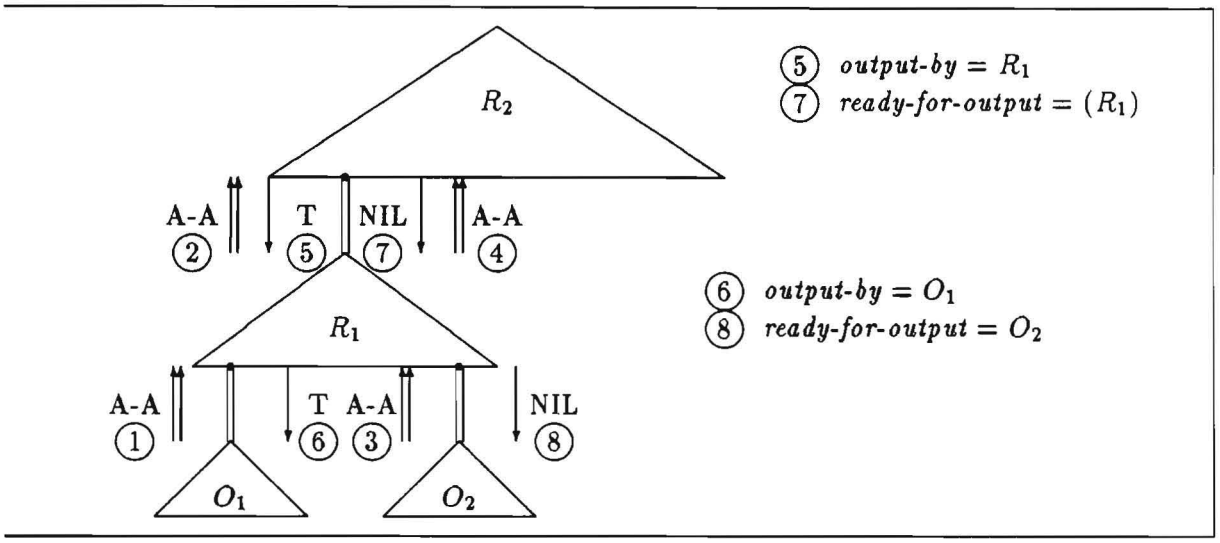


Figure 22: Forwarding the Answer to the Output Call

its partial tree), the answer is 'T' and $R_1$ is stored in *output-by*. $R_1$ forwards the answer to $O_1$ and stores $O_1$ in *output-by*. In contrast to this, the call from $O_2$ reaches $R_2$ at a time when another one of its dependents is occupied with uttering. This dependent is the same as the direct caller, but the list of callers differs in a deeper level. The output call is refused, but $R_2$ as well as $R_1$ store the respective dependent ($R_1$ or $O_2$) in *ready-for-output*. Obviously, it is redundant to store $R_1$ in the variables *output-by* and *ready-for-output* of $R_2$, but this redundancy does not disturb the processing and is maintained for reasons of simplicity and uniformity.

### Reacting to the Answer

If the answer is positive, the partial string that has been computed in the meantime is presented on a special part of the screen (see Section 5.2). If the answer is NIL, the object waits for the permission, because its readiness is now known to its regent. In spite of its complexity, the forwarding of the output call and the respective answer over several objects functions without deadlocks and in a finite number of steps. The reason is the concept of one direction of activity. If an object itself can not answer a call, it forwards it to its own regent. Only this one (or a higher object) can have enough information to decide about the positioning of the calling object. The forwarding of the call must end at the absolute regent of the hierarchy of objects. There are no crossings of calls, because their transfer is synchronized.

No object has to send its output call twice and its readiness is stored in the variable *ready-for-output* of the regent after the first call. So the time for handling each output

call stays linear (with the number of objects or the number of words of the sentence).

During the time that passes between the first output call of an object and the reception of the answer, the object tries to find a local connected partial terminal string and prepares it for output. It must be able to *linearize* and *inflect*.

## 5.2 Linearization and Inflection

### Linearization

As mentioned above, the trigger for the first linearization phase is sending the output call. This may only happen if the object fulfills the output condition, i.e., if there is a lexeme which can be uttered locally in the first position. The next goal is to look for this lexeme and as many directly connectable other lexemes as possible, and to store their order (as the output is not yet allowed). The substitution nodes possibly divide the linearization of the local tree into several steps, namely if there is no order of the terminal leaves which allows for a continuous sequence of words. The first linearization part is computed during the output call and the respective string is stored. All further connected sequences of words can be uttered directly during several output processes. The two different phases of linearization are described in more detail in the following paragraphs.

**Linearization during the Output Call:** The goal of the linearization process during the output call is to compute a local string of lexemes which is as long as possible. Therefore, parts of the tree must be linearized, the found lexemes must be inflected (see Section 5.2) and stored in the local variable *output-string*. If the answer to the output call is positive, this string may be uttered immediately.

The linearization process traverses the actual subtree from its root depth-first in order to find a path leading to a lexeme without contradicting any given LP rule. The LP condition sets are examined with respect to the elements which are already stored in *output-string*. In Figure 23, $X$ and $Y$ have been identified as possible start of the terminal

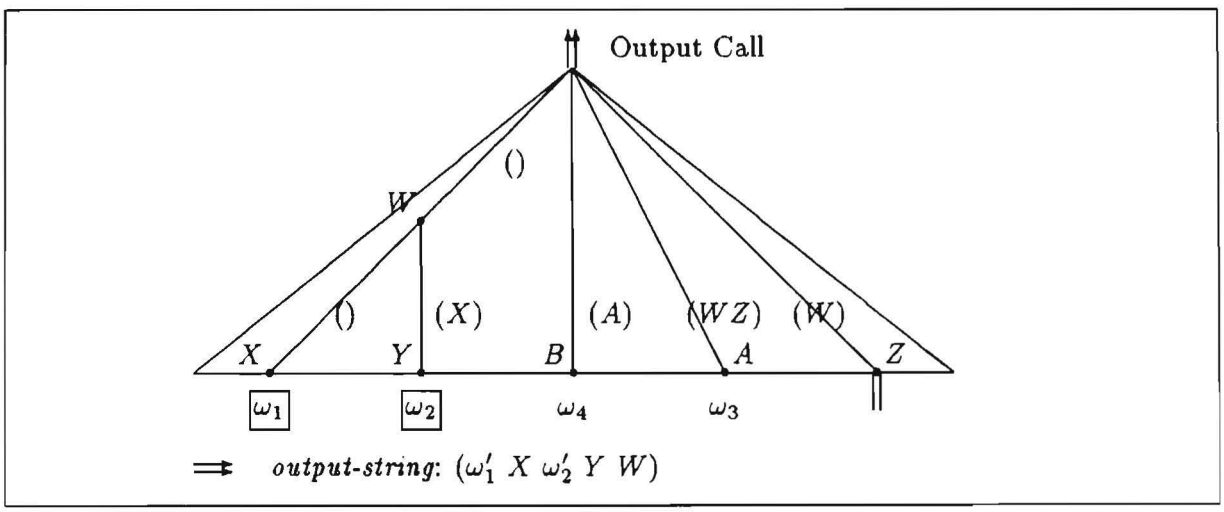


Figure 23: Computation of *output-string* during the First Linearization Phase

string. Their lexemes are inflected in this order (and called $\omega_1'$ and $\omega_2'$ for reasons of

distinction) and stored in *output-string*. This does not only finish the linearization of $X$ and $Y$ but also of the complete subtree under $W$. These three nodes are also stored in *output-string*. This variable is used during the further phases of linearization and is filled incrementally.

**Linearization and Output after Completion Messages from Subtrees:** Substitution nodes are 'breaks' for the first linearization phase. If all possible – according to the given LP rules – lexemes to the left of the first substitution node are uttered, further output can not start before the respective dependent objects have completed their own utterance. For this reason, each message from a dependent about the completion of its output triggers a new linearization phase. First, the uttered string and the respective node numbers are stored in *output-string*. In Figure 24, this is the string $v_1$ and the node $Z$. During the next phases, linearization and entries in *output-string* are processed as

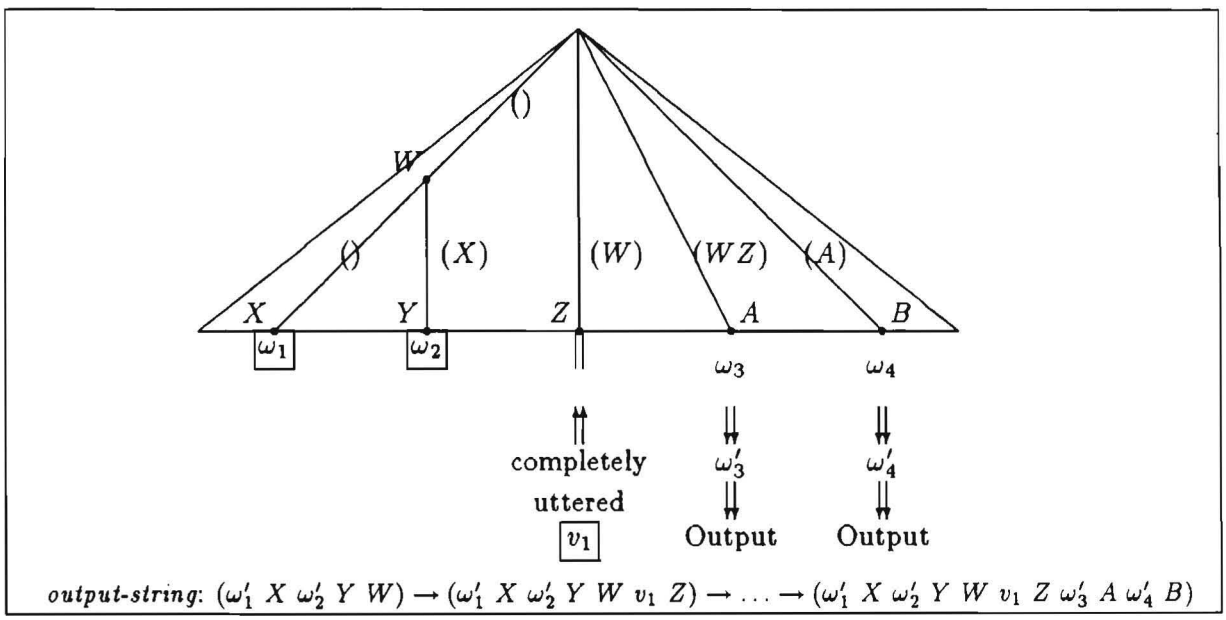


Figure 24: Linearization and Output after Completion Messages from Subtrees

before but – in contrast to the first linearization phase – each reached lexeme can directly be inflected and uttered. In Figure 24, these are the lexemes $\omega_3$ amd $\omega_4$ (inflected $\omega'_3$ and $\omega'_4$).

If another substitution node breaks a linearization phase, the readiness of the respective dependent can be found in *ready-for-output*. If its address is found, it is requested to linearize its subtree.

The above described depth-first search in the trees in several phases is a simple and basical principle of the positional level. It can easily be shown that this processing is disturbed by structural changes of the actual tree. The order of subtrees can be changed by late adjunctions, so that in this case linearization has to start again for the whole tree.

### Inflection

The lexemes are inflected with help of the morphological module MORPHIX (see [Finkler & Neumann 89]). MORPHIX allows for generating and analyzing word forms. The details can be found in the cited literature.

Linearized and inflected words are uttered from time to time, as has been shown in Section 5.2. Although the output is realized as simple as possible, it is briefly described in the next section.

## 5.3 Output at the Positional Level

The first output of an object – after its first linearization phase – is read from the variable *output-string*. Each further output is produced directly during the linearization of further subtrees. Each time, the respective roots of the subtrees are stored in *output-string*.

If the output is complete, a message is sent to the regent and the object starts 'sleeping'. The regent can store the respective substitution node and eventually some others in *output-string*. Another consequence of the message is that it means an output permission for the regent if it is at the positional level. Its variable *output-by* is set to its own address. In the worst case, the regent can not directly start to utter but has to wait for other dependents first.

Together with the message that the output is complete the output string is transferred to the regent (see Figure 24). It is stored in *output-string*. This duplication of terminal strings simplifies a local correction of the output in the case of late structural changes. This problem is handled in the following.

## 5.4 Late Adjunction in Objects at the Positional Level

### Two Cases of Late Adjunction

There are two different problems with late adjunctions in objects at the positional level that have to do with the actual situation of the object.

**After Sending the Output Call:** After an object has entered the positional level, fulfilled the output condition, and sent the output call, the output condition can later be changed by adjunctions. Figure 25 shows, that the object fulfilled the output condition because of the leaf $\omega_1$ that could be positioned at the left. After another adjunction, the object's state has changed. The first element of the output string must now be the terminal string of a dependent. So the output condition is no more fulfilled.

Corrections in incremental generation systems are a very complicated task. In the presented module, a very simple and uniform solution has been realized, which is on the other hand very costly. Better solutions can be imagined, e.g., by using communication between objects or by realizing a central component for control. The solution for this prototype is that the state of the object remains the same. In the worst case, the output is hereby delayed. If the answer to the output call is negative, the problem is less relevant. If it is positive, the object must not produce any output itself, but it can react fast (and

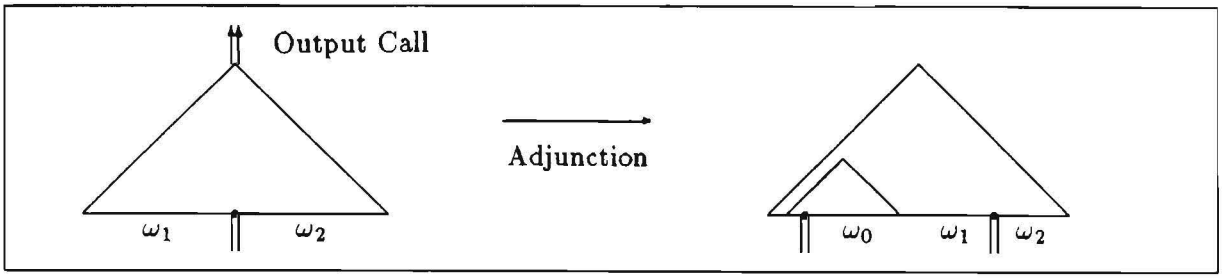


Figure 25: Late Adjunction after Sending the Output Call

positively) to an output call of the problematic dependent which hopefully is received soon.

There are other cases where the results of linearization (the entries in *output-string*) must be newly computed.

**During the Output Phase:** Late adjunctions can only become visible to the user if the respective object is in the output phase and if the structural change leads to a situation, where parts of the utterance must be drawn back. Figure 26 shows two situations that can come up after a late adjunction during the output phase. On the left side, the worse case is shown: $\omega_1$ and $\omega_2$ have already been uttered (which is marked by the boxes). The

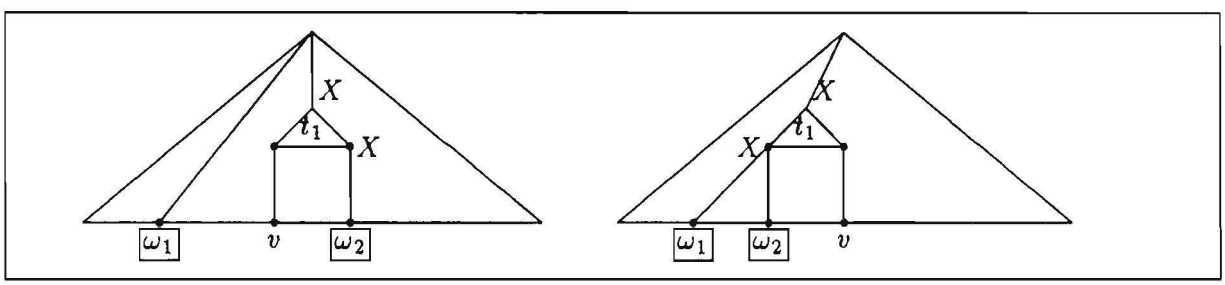


Figure 26: Situations with Late Adjunctions during the Output Phase

consequence of the adjunction of $t_1$ is, that the terminal string $v$ is inserted between the previously uttered parts.

There can also be adjunctions that do not change the output string but only add some words to the right (see the example on the right side of Figure 26). The situation becomes problematic if, e.g., the subtree below $X$ that now is placed below the foot node $X$, has already been marked as completely uttered. There must be a correction, too.

Again, a simple and uniform solution is realized. Overt revisions in the output phase should be prevented or at least limited. One possibility is to influence the hierarchical level with the aim to choose another auxiliary tree. If such a tree cannot be found, *output-string* is newly computed and compared with the previous state. According to the result, overt revisions are produced.

It has been motivated that in the case of a late adjunction the variable *output-string* must be adapted to the new state. This is done by a new computation. In the following, it will be explained how the produced utterance is overtly repaired on the screen.

### Repairing the Output

Either the concerned object is able to repair the output itself, or it must transfer the newly computed *output-string* to its regent. This depends on the question if the object has already finished its output phase.

**Output Phase not yet finished:** If the output phase of the object is not yet finished, either the object itself, or one of its dependents is actually occupied with the computation of its output string. This can be examined by a look at the variable output-by.

1. **output-by = own address**: The object itself is actually occupied with the computation of its output string and therefore can repair it locally. The kind of repair *depends on the result of the comparison of the old and the new value of output-string*. If the new one results from a simple addition to the old one, the utterance can be succeeded to the right. But if the terminals in the new string have another order than the terminals in the old one, the repair becomes visible. The most simple realization of repair on the screen is the jump into a new output line and the total repetition of the sentence. The example in Figure 27 shows how a late adjunction inserts terminals between the previously uttered lexemes $\omega_1$ and $\omega_2$. Furthermore,

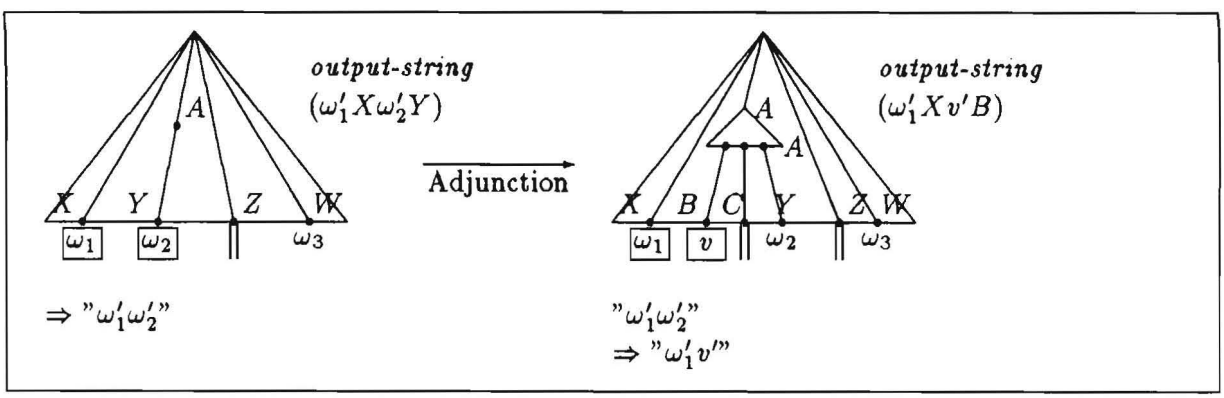


Figure 27: Local Repair

an additional substitution node is inserted between $X$ and $Y$. So not only the order of terminals in the new and the old value of *output-string* are different, $v'$ must also be added. The object starts a new line on the screen and utters the changed beginning of the sentence. This total repetition can only be locally computed if the object is the absolute regent within the hierarchy. Otherwise, it must hand over a respective message to its regent which is forwarded to the absolute regent.

Repair can be realized in a more flexible and adequate way, e.g., words could be crossed out or erased, new parts could be inserted into the sentence, and so on.

2. **output-by = address of a dependent**: The simplest solution in this case is: The object does not try to stop the dependent's output - which could lead to communication over several levels. Instead it waits for its message 'output complete'. At this time, the object itself can again become active and can handle as described for Point 1.

if the object has already finished its output phase (*output-by* = address of the regent), its regent must repair the output.

**Output Phase already finished:** As above, the object newly computes its *output-string*. It sends a message to its regent which is forwarded if necessary and leads to the interruption of output, no matter which object is actually occupied with it. When the object has received the confirmation and again has finished its local linearization, it sends a new terminal string to its regent which must care for the realization of the overt repair.

# 6 Conclusion and Outlook

## 6.1 Main Results of the Work

The goal of this work was to develop concepts for a syntactic generator that uses *Tree Adjoining Grammars* to generate natural language sentences in an *incremental* style. Incremental processing within the formulator suggests that input from the conceptualizer and output to the articulator should be incremental, too. The model of *cascades* allows for the transfer of partial results to the succeeding components, so that there may be parallelism at all levels.

ISGT is a *description-directed* generator (see Section 2.2). The idea of this approach is to insert several levels of explicit linguistic representation between message and sentence string representing the text on several levels of abstraction. Thereby, the data on the distinct levels guide the syntactic generation and allow for a more efficient processing. We use *Lexicalized TAGs with Unification and Constraints*, as this extension of the formalism fulfills the requirements upon a syntactic representation of natural language (see Sections 3.1, 3.2 and 3.3).

During the construction of the sentence structure, there should be as few as possible fixings with respect to the final position of subtrees. The order of input and the construction of subtrees should influence the order of output, thereby supporting an incremental style of processing. Hierarchy and position of constituents are therefore computed in distinct components. *LD/LP TAGs* allow for such a use of TAG trees (see Sections 2.3 and 3.4). The syntactic generator ISGT is separated into two components:

- At the *hierarchical level*, the syntactic structure of the sentence is built using the LD parts of TAG structures, i.e., mobiles (Section 4).
- At the *positional level*, word order is computed with the help of the LP rules, the lexemes are inflected and uttered (Section 5).

At the hierarchical as well as at the positional level, processing is incremental. The use of *parallel interactive communicating objects* allows for a further gain in efficiency. These objects manage subtrees of the overall structure (see Section 2.4).

The contributions of this work to the use of TAGs for incremental and parallel syntactic generation are

- a motivation for the use of lexicalized TAGs,
- the choice of an adequate segmentation of the representation structure,
- concepts for the incremental and parallel construction of syntactic structures at the hierarchical level and
- strategies for an incremental and parallel linearization at the poitional level.

The following sections will deal with these points.

### A Motivation for the Use of Lexicalized TAGs:

The system ISGT uses *conceptual and lexical guidance* to guide the choice of alternative trees and the construction of the sentence tree (see Section 2.3). The input for ISGT consists of lexemes and information about their functional relationship with the aim to make the choice of TAG trees as deterministic as possible.

Trees of basic TAGs contradict this demand. They should – this is a linguistic constraint – always present the predicate of the represented substructure together with all its obligatory arguments. As the structure of the regent and all substructures of its arguments must be expanded down to the terminal nodes, the choice of such a tree means the decision over structures which are not described by the input information associated with the regent. The choice component should therefore wait until all dependents of the respective regent have been specified by input information. This would cause a delay of the choice of trees and of the construction of the syntactic structure and directly leads to the demand for a further separation of TAG trees into smaller, more adequate parts. The formalism of lexicalized TAGs fulfills this demand: Each tree can be identified by the syntactic information of its regent and its functional relation to the surrounding structure.

### An Adequate Segmentation of the Representation Structure:

Since the tasks of the hierarchical and the positional level should be solved by interactive objects working in parallel, the representation structure must be adequately segmented in order to minimize the communication between the objects (see Section 2.4). The elementary trees of a lexicalized TAG serve as segments for ISGT. With the input of one lexeme and its functional relation to its regent, a tree can be chosen and an object can be created which manages its further processing.

Substitution nodes are an ideal interface between the regent object and the object that manages the substitution tree. Objects with auxiliary trees hand over all their local information during their adjunction into the regent. This saves costly management and communication. In the final state, the only active objects at the hierarchical and the positional level are objects with initial (and in the meantime integrated auxiliary) trees. In this way, the processing becomes uniform and simple. Furthermore, the resulting size of segments seems to be advantageous for an incremental and parallel computation on both levels of the formulator

### The Incremental and Parallel Construction of the Syntactic Structure at the Hierarchical Level:

Objects at the hierarchical level (see Section 4) are created incrementally on the basis of the input from the conceptualizer. Part of their local variables are instantiated with the input data. They are used during the *initialization phase* to choose an equivalent TAG tree, and additionally to identify other objects, with which the recent one can communicate and cooperate to build the overall structure. The communication partner for the initialization phase is the *monitor*. It receives information from each object that wants to be registered and builds a *global structure* on this basis giving an overview of the state of the hierarchical level. The monitor uses its global knowledge for the choice of trees and the computation of the addresses of communication partners. The other way

round, the monitor informs all objects about newly registered objects that are functionally related, thereby guaranteeing the construction of all necessary communication links.

After its initialization phase, each object tries to fit its tree into the existing syntactic structure. An efficient strategy to control this construction is the definition of a *direction of activity* from the dependent to the regent. One reason for this is, that the dependent has to fulfill specific *completeness conditions* in order to integrate its local tree into the syntactic structure. These criteria differ for objects with inital and objects with auxiliary trees. *Objects with auxiliary trees are incorporated during adjunction in the regent tree.* Therefore, they have to have solved all their tasks before this combination. During substitution, the feature structure of the substitution node and the feature structure of the root of the substitution tree are unified. *Substitution is realized as exchange of information between the dependent and the regent objects.* The hierarchical level converges to a state where only objects with initial trees are active, whose substitution and root nodes are ideal interfaces for a distributed sentence tree.

**Strategies for Incremental and Parallel Linearization at the Positional Level:**

Since in the final state there are only objects with initial trees at the hierarchical level, only these trees must be transferred to the positional level (see Section 5). The change is done within the work at the hierarchical level when an objects fulfills specific *linearization conditions* which allow for a further processing at the positional level. As the linear precedence rules which are relevant at the positional level refer to the hierarchical structures (mobiles) of the hierarchical level, these are not copied into new objects. *Instead, the objects of the hierarchical level change their state when fulfilling the linearization conditions and exchange their goal against a new one: The locally managed structures are to be linearized, the lexemes are to be inflected and uttered.*

As the LP rules of the TAG structures only allow for a local definition of positions of partial trees, one object alone cannot decide about the position of its subtree in the whole syntactic tree. This can only be done by the regent of the complete syntactic tree. Each object that fulfills the linearization condition, i.e., that has a leaf (not a substitution node) in the first local position, must ask its regent if output is allowed. The call is forwarded by several objects to their respective regent until the top object is found, or until one is asked which can give a unique answer because of its situation. Even if the answer is negative, the object needs not call again: It is stored at its regent and the regent will inform it if it may utter its terminal string.

The time that passes between sending the output call and getting the answer can be used for a *first linearization phase.* Coming from the lexeme that can be positioned at the leftmost place, the tree is traversed according to the LP rules so that a terminal string is computed that is as long as possible. The lexemes are inflected and stored. They can be uttered immediately, when output is allowed for the object.

The first linearization phase stops if a substitution node is found. Only if the respective subtree has been uttered (the dependent is called to do so), the output can be continued. Further linearization phases can again be interrupted by substitution nodes and differ from the first phase in that each reached lexeme can be inflected and uttered immediately.

Thereby, the output activity passes over from one object to another. It is always checked by output calls that the local tree may be uttered next within the global struc-

ture. The output activity is passed over from each object to its dependents or to its regent if the output is locally complete. In this way, the sentence tree is completely and efficiently traversed at the positional level.

The presented concepts for incremental syntactic generation with Tree Adjoining Grammars can be further developed at several points which are briefly discussed in the following.

## 6.2 Further Developments

In addition to pure practical extension – like the development of a large-sized grammar and a lexicon – there are several theoretical problems that have been mentioned within the respective sections of this work and that are interesting starting-points for a further development of the system. On one hand, they concern a more detailed specification of concepts, on the other hand alternative concepts especially questioning the role of Tree Adjoining Grammars within the formulator.

### Starting-Points for a more Detailed Conception:

Generally, the system must be tested not in connection with a simulated input, but with a real conceptualizer which hands over the input, and with a more complex articulator. If several levels are realized, it must be thought over if a heterarchical model (which, e.g., allows for direct influence of the conceptualizer to the articulator) should be preferred to the cascade.

Additionally, the system should be implemented on a machine with several processors. Measurings of performance could show possible speed ups by parallelism.

The concepts of the hierarchical and the positional level that are to be worked out in more detail are mentioned in the respective sections. They affect the choice of TAG trees during the initialization phase, the different completeness conditions, and a better control of the information flow over the interfaces represented by substitution nodes and root nodes. Up to now, revision is solved with minimal means. Much work can be done to develop concepts for a more sophisticated realization of revision on a psycholinguistical and computational basis. This is important especially because an incremental style of generation often causes conflicts and leads to the correction of previously uttered parts.

### The Role of TAGs in an Incremental Syntactic Generator:

There are several points of criticism to the used extensions of TAGs which have partially been mentioned in Section 3.

TAGs with unification are too powerful for the representation of natural language. A useful restriction of the formalism should be worked out to allow for a more adequate representation and a more efficient generation of natural language.

Lexicalized TAGs are used with the additional constraint that each tree must contain terminal leaves. A less restricted formalism could be used, e.g., to define structures without leaves, i.e., without connection to the lexicon. This approach could be similar to the formalism Segment Grammar (see [DeSmedt 90]): Trees, that have only substitution nodes instead of terminal nodes, can be used to define purely syntactic (functional) relations between partial structures. This helps to avoid redundancy in the definition of the

grammar and possibly improves the use of the trees within an incremental system.

The most problematic extension of TAGs is LD/LP-TAG. It is not very useful for a skillful formulation of word order rules of natural language (e.g., German). The concepts of this work have been developped with the presupposition of using an existing TAG extension, and not to define a new one. The definition of LD/LP-TAGs strongly influences the realization of the positional level. The presented approach is an applicable solution with respect to the given situation. But it does not allow for a flexible realization of all possible word positions of German. One idea inspired by [DeSmedt 90] consists in the use of another representation formalism at the positional level. For ISGT, this would mean not to use Tree Adjoining Grammars for the representation of positional relations between constituents. DeSmedt's idea is to turn away from the local and relative definitions of LP rules. He arrives at the definition of absolute positions in the sentence. It could be interesting to try to find a similar approach for an incremental syntactic generator based on TAGs. The transfer between hierarchical and positional level would then be a real transformation and the positional level would have to be totally redesigned.

Within this work, concepts for an incremental TAG-based syntactic natural language generator have been developped and basically implemented. The result is the module ISGT that has been presented and reviewed in this chapter.

# References


[Buschauer et al. 91] B. Buschauer, P. Poller, A. Schauder, K. Harbusch: *Parsing von TAGs mit Unifikation*, German Research Center for Artificial Intelligence (DFKI), Technical Memo TM-91-10, Saarbrücken, FRG, 1991.

[DeSmedt & Kempen 87] K. DeSmedt, G. Kempen: *Incremental Sentence Production, Self-Correction and Coordination*, in G. Kempen: Natural Language Generation, Martinus Nijhoff Publishers, 1987.

[DeSmedt 90] K. DeSmedt: *Incremental Sentence Generation: A Computer Model of Grammatical Encoding*, Ph.D.thesis, Department of Psychology, University of Nijmegen, 1990.

[Engel 77] U. Engel: *Syntax der deutschen Gegenwartssprache*, Erich Schmidt Verlag, Berlin, 1977.

[Finkler 89] W. Finkler: *POPEL-HOW: Ein verteiltes paralleles Modell zur inkrementellen Generierung natürlichsprachlicher Sätze aus konzeptuellen Einheiten: Teil 1*, Master's thesis, Department of Computer Science, University of the Saarland, Saarbrücken, FRG, 1989.

[Finkler & Neumann 89] W. Finkler, G. Neumann: *MORPHIX: A Fast Realization of a Classification-Based Approach to Morphology*, Proceedings of the Workshop 'Wissensbasierte Sprachverarbeitung', Berlin: Springer, August 1989.

[Joshi et al. 75] A. Joshi, L. Levy, M. Takahashi: *Tree Adjunct Grammars*, Journal of the Computer and Systems Science, Vol 10, No 1, 136-163, 1975.

[Joshi 85] A. Joshi: *An Introduction to TAGs*, Technical Report MS-CIS-86-64, LINC-LAB-31, Department of Computer and Information Science, Moore School, University of Pennsylvania, 1985.

[Kempen 77] G. Kempen: *Conceptualizing and Formulating in Sentence Production*, in S. Rosenberg: Sentence Production: Developments in Research and Theory, Hillsdale, N.J.: Erlbaum, 1977.

[Kempen & Hoenkamp 82] G. Kempen, E. Hoenkamp: *Incremental Sentence Generation*, Proceedings of COLING-82, pp. 151-156, 1982.

[McDonald & Pustejovsky 85] D. McDonald, J. Pustejovsky: *Description-Directed Natural Language Generation*, Proceedings of IJCAI-85, W. Kaufmann Inc., Los Altos, CA, 1985.

[McDonald et al. 87a] D. McDonald, M. Vaughan, J. Pustejovsky : *Factors Contributing to Efficiency in Natural Language Generation*, in G. Kempen : Natural Language Generation, Dordrecht, Boston, Lancaster: Martinus Nijhoff Publishers, 1987.

[McDonald 87b] D. McDonald: *Natural Language Generation: Complexities and Techniques*, in S. Nirenburg : Machine Translation, Cambridge: Cambridge University Press, 1987.

[Neumann 89] G. Neumann: *POPEL-HOW: Parallele, inkrementelle Generierung natürlichsprachlicher Sätze aus konzeptuellen Einheiten: Teil 2*, Master's thesis, Department of Computer Science, University of the Saarland, Saarbrücken, FRG, 1989.

[Reithinger 88] N. Reithinger: *POPEL: A Parallel and Incremental Natural Language Generation System*, Paper presented at the 4th IWG, Santa Catalina Island, 1988.

[Schabes et al. 88] Y. Schabes, A. Abeillé, A. Joshi: *Parsing Strategies with Lexicalized Grammars: Application to Tree Adjoining Grammar*, Proc. of COLING 88, Budapest, 1988.

[Schauder 90] A. Schauder: *Inkrementelle syntaktische Generierung natürlicher Sprache mit Tree Adjoining Grammars*, Master's thesis, Department of Computer Science, University of the Saarland, Saarbrücken, FRG, 1990.

[Schauder 92] A. Schauder: *Realization of Tree Adjoining Grammars with Unification*, Technical Report, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, FRG, 1992. to appear.

[Shieber 86] S. Shieber: *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Note, No. 4, Stanford University, Stanford, California, 1986.

[Steele 90] G. Steele: *Common LISP: The Language*, Digital Press, Bedford, Massachusetts, 1990.

[Symbolics 89] Symbolics Handbuch Nr. 8: *Internals*, Symbolics Inc., Burlington, MA, 1989.

[Tesnière 59] L. Tesnière : *Éléments de syntaxe structurale*, Klincksieck, Paris, 1959.

[Vijay-Shanker & Joshi 88] K. Vijay-Shanker, A. Joshi: *Feature Structure Based Tree Adjoining Grammars*, Proc. of COLING 88, Budapest, 1988.

[Wahlster 81] W. Wahlster: *Natürlichsprachliche KI-Systeme: Entwicklungsstand und Forschungsperspektive*, GWAI, 1981.

[Wahlster 82] W. Wahlster: *Natürlichsprachliche Systeme: Eine Einführung in die sprachorientierte KI-Forschung*, in W. Bibel, J. Siekmann: Künstliche Intelligenz, Frühjahrsschule, Teisendorf, 1982.

[Wahlster et al. 88] W. Wahlster, E. André, M. Hecking, T. Rist : *WIP: Wissensbasierte Informationspräsentation*, Description of the Project, DFKI Saarbrücken, 1988.

[Yonezawa & Tokoro 87] A. Yonezawa, M. Tokoro: *Object-Oriented Concurrent Programming: An Introduction*, in A. Yonezawa, M. Tokoro: *Object-Oriented Concurrent Programming*, Cambridge, Massachusetts: MIT Press, 1987.

Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

DFKI
-Bibliothek-
PF 2080
D-6750 Kaiserslautern
FRG

## DFKI Publikationen

Die folgenden DFKI Veröffentlichungen sowie die aktuelle Liste von allen bisher erschienenen Publikationen können von der oben angegebenen Adresse bezogen werden.
Die Berichte werden, wenn nicht anders gekennzeichnet, kostenlos abgegeben.

## DFKI Publications

The following DFKI publications or the list of all publisched papers so far can be ordered from the above address.
The reports are distributed free of charge except if otherwise indicated.

### DFKI Research Reports

**RR-91-08**
*Wolfgang Wahlster, Elisabeth André, Som Bandyopadhyay, Winfried Graf, Thomas Rist:* WIP: The Coordinated Generation of Multimodal Presentations from a Common Representation
23 pages

**RR-91-09**
*Hans-Jürgen Bürckert, Jürgen Müller, Achim Schupeta:* RATMAN and its Relation to Other Multi-Agent Testbeds
31 pages

**RR-91-10**
*Franz Baader, Philipp Hanschke:* A Scheme for Integrating Concrete Domains into Concept Languages
31 pages

**RR-91-11**
*Bernhard Nebel:* Belief Revision and Default Reasoning: Syntax-Based Approaches
37 pages

**RR-91-12**
*J.Mark Gawron, John Nerbonne, Stanley Peters:* The Absorption Principle and E-Type Anaphora
33 pages

**RR-91-13**
*Gert Smolka:* Residuation and Guarded Rules for Constraint Logic Programming
17 pages

**RR-91-14**
*Peter Breuer, Jürgen Müller:* A Two Level Representation for Spatial Relations, Part I
27 pages

**RR-91-15**
*Bernhard Nebel, Gert Smolka:* Attributive Description Formalisms ... and the Rest of the World
20 pages

**RR-91-16**
*Stephan Busemann:* Using Pattern-Action Rules for the Generation of GPSG Structures from Separate Semantic Representations
18 pages

**RR-91-17**
*Andreas Dengel, Nelson M. Mattos:* The Use of Abstraction Concepts for Representing and Structuring Documents
17 pages

**RR-91-18**
*John Nerbonne, Klaus Netter, Abdel Kader Diagne, Ludwig Dickmann, Judith Klein:* A Diagnostic Tool for German Syntax
20 pages

**RR-91-19**
*Munindar P. Singh:* On the Commitments and Precommitments of Limited Agents
15 pages

**RR-91-20**
*Christoph Klauck, Ansgar Bernardi, Ralf Legleitner* FEAT-Rep: Representing Features in CAD/CAM
48 pages

**RR-91-21**
*Klaus Netter:* Clause Union and Verb Raising Phenomena in German
38 pages

**RR-91-22**
*Andreas Dengel:* Self-Adapting Structuring and Representation of Space
27 pages

**RR-91-23**
*Michael Richter, Ansgar Bernardi, Christoph Klauck, Ralf Legleitner:* Akquisition und Repräsentation von technischem Wissen für Planungsaufgaben im Bereich der Fertigungstechnik
24 Seiten

**RR-91-24**
*Jochen Heinsohn:* A Hybrid Approach for Modeling Uncertainty in Terminological Logics
22 pages

**RR-91-25**
*Karin Harbusch, Wolfgang Finkler, Anne Schauder:* Incremental Syntax Generation with Tree Adjoining Grammars
16 pages

**RR-91-26**
*M. Bauer, S. Biundo, D. Dengler, M. Hecking, J. Koehler, G. Merziger:* Integrated Plan Generation and Recognition - A Logic-Based Approach -
17 pages

**RR-91-27**
*A. Bernardi, H. Boley, Ph. Hanschke, K. Hinkelmann, Ch. Klauck, O. Kühn, R. Legleitner, M. Meyer, M. M. Richter, F. Schmalhofer, G. Schmidt, W. Sommer:* ARC-TEC: Acquisition, Representation and Compilation of Technical Knowledge
18 pages

**RR-91-28**
*Rolf Backofen, Harald Trost, Hans Uszkoreit:* Linking Typed Feature Formalisms and Terminological Knowledge Representation Languages in Natural Language Front-Ends
11 pages

**RR-91-29**
*Hans Uszkoreit:* Strategies for Adding Control Information to Declarative Grammars
17 pages

**RR-91-30**
*Dan Flickinger, John Nerbonne:* Inheritance and Complementation: A Case Study of *Easy* Adjectives and Related Nouns
39 pages

**RR-91-31**
*H.-U. Krieger, J. Nerbonne:* Feature-Based Inheritance Networks for Computational Lexicons
11 pages

**RR-91-32**
*Rolf Backofen, Lutz Euler, Günther Görz:* Towards the Integration of Functions, Relations and Types in an AI Programming Language
14 pages

**RR-91-33**
*Franz Baader, Klaus Schulz:* Unification in the Union of Disjoint Equational Theories: Combining Decision Procedures
33 pages

**RR-91-34**
*Bernhard Nebel, Christer Bäckström:* On the Computational Complexity of Temporal Projection and some related Problems
35 pages

**RR-91-35**
*Winfried Graf, Wolfgang Maaß:* Constraint-basierte Verarbeitung graphischen Wissens
14 Seiten

**RR-92-01**
*Werner Nutt:* Unification in Monoidal Theories is Solving Linear Equations over Semirings
57 pages

**RR-92-02**
*Andreas Dengel, Rainer Bleisinger, Rainer Hoch, Frank Hönes, Frank Fein, Michael Malburg:* $\Pi_{ODA}$: The Paper Interface to ODA
53 pages

**RR-92-03**
*Harold Boley:* Extended Logic-plus-Functional Programming
28 pages

**RR-92-04**
*John Nerbonne:* Feature-Based Lexicons: An Example and a Comparison to DATR
15 pages

**RR-92-05**
*Ansgar Bernardi, Christoph Klauck, Ralf Legleitner, Michael Schulte, Rainer Stark:* Feature based Integration of CAD and CAPP
19 pages

**RR-92-07**
*Michael Beetz:* Decision-theoretic Transformational Planning
22 pages

**RR-92-08**
*Gabriele Merziger:* Approaches to Abductive Reasoning - An Overview -
46 pages

**RR-92-09**
*Winfried Graf, Markus A. Thies:* Perspektiven zur Kombination von automatischem Animationsdesign und planbasierter Hilfe
15 Seiten

**RR-92-11**
*Susane Biundo, Dietmar Dengler, Jana Koehler:*
Deductive Planning and Plan Reuse in a Command Language Environment
13 pages

**RR-92-13**
*Markus A. Thies, Frank Berger:*
Planbasierte graphische Hilfe in objektorientierten Benutzungsoberflächen
13 Seiten

**RR-92-14**
Intelligent User Support in Graphical User Interfaces:
1. InCome: A System to Navigate through Interactions and Plans
   *Thomas Fehrle, Markus A. Thies*
2. Plan-Based Graphical Help in Object-Oriented User Interfaces
   *Markus A. Thies, Frank Berger*

22 pages

**RR-92-15**
*Winfried Graf:* Constraint-Based Graphical Layout of Multimodal Presentations
23 pages

**RR-92-17**
*Hassan Aït-Kaci, Andreas Podelski, Gert Smolka:*
A Feature-based Constraint System for Logic Programming with Entailment
23 pages

**RR-92-18**
*John Nerbonne:* Constraint-Based Semantics
21 pages

---

**DFKI Technical Memos**

**TM-91-01**
*Jana Köhler:* Approaches to the Reuse of Plan Schemata in Planning Formalisms
52 pages

**TM-91-02**
*Knut Hinkelmann:* Bidirectional Reasoning of Horn Clause Programs: Transformation and Compilation
20 pages

**TM-91-03**
*Otto Kühn, Marc Linster, Gabriele Schmidt:*
Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model
20 pages

**TM-91-04**
*Harold Boley (Ed.):*
A sampler of Relational/Functional Definitions
12 pages

**TM-91-05**
*Jay C. Weber, Andreas Dengel , Rainer Bleisinger:*
Theoretical Consideration of Goal Recognition Aspects for Understanding Information in Business Letters
10 pages

**TM-91-06**
*Johannes Stein:* Aspects of Cooperating Agents
22 pages

**TM-91-08**
*Munindar P. Singh:* Social and Psychological Commitments in Multiagent Systems
11 pages

**TM-91-09**
*Munindar P. Singh:* On the Semantics of Protocols Among Distributed Intelligent Agents
18 pages

**TM-91-10**
*Béla Buschauer, Peter Poller, Anne Schauder, Karin Harbusch:* Tree Adjoining Grammars mit Unifikation
149 pages

**TM-91-11**
*Peter Wazinski:* Generating Spatial Descriptions for Cross-modal References
21 pages

**TM-91-12**
*Klaus Becker, Christoph Klauck, Johannes Schwagereit:* FEAT-PATR: Eine Erweiterung des D-PATR zur Feature-Erkennung in CAD/CAM
33 Seiten

**TM-91-13**
*Knut Hinkelmann:*
Forward Logic Evaluation: Developing a Compiler from a Partially Evaluated Meta Interpreter
16 pages

**TM-91-14**
*Rainer Bleisinger, Rainer Hoch, Andreas Dengel:*
ODA-based modeling for document analysis
14 pages

**TM-91-15**
*Stefan Bussmann:* Prototypical Concept Formation
An Alternative Approach to Knowledge Representation
28 pages

**TM-92-01**
*Lijuan Zhang:*
Entwurf und Implementierung eines Compilers zur Transformation von Werkstückrepräsentationen
34 Seiten

**DFKI Documents**

**D-91-04**
DFKI Wissenschaftlich-Technischer Jahresbericht 1990
93 Seiten

**D-91-06**
*Gerd Kamp:* Entwurf, vergleichende Beschreibung und Integration eines Arbeitsplanerstellungssystems für Drehteile
130 Seiten

**D-91-07**
*Ansgar Bernardi, Christoph Klauck, Ralf Legleitner*
TEC-REP: Repräsentation von Geometrie- und Technologieinformationen
70 Seiten

**D-91-08**
*Thomas Krause:* Globale Datenflußanalyse und horizontale Compilation der relational-funktionalen Sprache RELFUN
137 Seiten

**D-91-09**
*David Powers, Lary Reeker (Eds.):*
Proceedings MLNLO'91 - Machine Learning of Natural Language and Ontology
211 pages
**Note:** This document is available only for a nominal charge of 25 DM (or 15 US-$).

**D-91-10**
*Donald R. Steiner, Jürgen Müller (Eds.):*
MAAMAW'91: Pre-Proceedings of the 3rd European Workshop on „Modeling Autonomous Agents and Multi-Agent Worlds"
246 pages
**Note:** This document is available only for a nominal charge of 25 DM (or 15 US-$).

**D-91-11**
*Thilo C. Horstmann:*Distributed Truth Maintenance
61 pages

**D-91-12**
*Bernd Bachmann:*
$H^{iera}C_{on}$ - a Knowledge Representation System with Typed Hierarchies and Constraints
75 pages

**D-91-13**
International Workshop on Terminological Logics
*Organizers: Bernhard Nebel, Christof Peltason, Kai von Luck*
131 pages

**D-91-14**
*Erich Achilles, Bernhard Hollunder, Armin Laux, Jörg-Peter Mohren: KRIS:* Knowledge Representation and Inference System
- Benutzerhandbuch -
28 Seiten

**D-91-15**
*Harold Boley, Philipp Hanschke, Martin Harm, Knut Hinkelmann, Thomas Labisch, Manfred Meyer, Jörg Müller, Thomas Oltzen, Michael Sintek, Werner Stein, Frank Steinle:*
µCAD2NC: A Declarative Lathe-Worplanning Model Transforming CAD-like Geometries into Abstract NC Programs
100 pages

**D-91-16**
*Jörg Thoben, Franz Schmalhofer, Thomas Reinartz:*
Wiederholungs-, Varianten- und Neuplanung bei der Fertigung rotationssymmetrischer Drehteile
134 Seiten

**D-91-17**
*Andreas Becker:*
Analyse der Planungsverfahren der KI im Hinblick auf ihre Eignung für die Abeitsplanung
86 Seiten

**D-91-18**
*Thomas Reinartz:* Definition von Problemklassen im Maschinenbau als eine Begriffsbildungsaufgabe
107 Seiten

**D-91-19**
*Peter Wazinski:* Objektlokalisation in graphischen Darstellungen
110 Seiten

**D-92-01**
*Stefan Bussmann:* Simulation Environment for Multi-Agent Worlds - Benutzeranleitung
50 Seiten

**D-92-08**
*Jochen Heinsohn, Bernhard Hollunder (Eds.):*
DFKI Workshop on Taxonomic Reasoning Proceedings
56 pages

**D-92-09**
*Gernod P. Laufkötter:* Implementierungsmöglichkeiten der integrativen Wissensakquisitionsmethode des ARC-TEC-Projektes
86 Seiten

**D-92-21**
*Anne Schauder:* Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars
57 pages

# Incremental Syntactic Generation of Natural Language with Tree Adjoining Grammars

Anne Schauder

D-92-21
Document