UNIVERSITY "POLITEHNICA" OF TIMISOARA
FACULTY OF AUTOMATIC CONTROL AND COMPUTER
SCIENCE AND ENGINEERING

A COMPUTATIONAL METHODOLOGY AS AN ARTIFICIAL LANGUAGE ABOUT NATURAL LANGUAGE RULES

PhD - Thesis

PhD coordinator:
Prof. Univ. dr. ing. Stefan HOLBAN

Author
Konstantinos Fouskakis

Timisoara – 2005

CONTENTS

1. Introduction

Natural language processing is a field that has concerned both *artificial intelligence* (Winston, 1992) (Dumitrescu, 2002) in the terms of its broader cognitive fields and the *computational linguistics* (Cecile, 1991) (Gilbert, 1991) (Grishman, 1989) (Noble, 1988) especially during the 60's and after.

Generally, the modern theoretical *linguistics* (Babiniotis, 1980) (Philippaki, 1987, 1992) (Lyons, 1981) is concerned with the scientific study of *language*. Namely, the object of the linguistics is the language. With the term *language* we are limited only to that natural communication system that is used by man and that is based on parole. Also *linguistics* is not limited only to one particular *language* (Fodor, 1964) and neither sets as a goal the study of each one of these languages individually. *Linguistics* studies the language as a phenomenon and its purpose is to define the general universal characteristics, of this phenomenon.

Linguistics has the following branches:

- *Phonetics* and *phonology*

  Phonetics is concerned on how the words of a language are pronounced (Malikouti, 1988), both individually and in combination among them within sentences or phrases. For the description of the words/ pronunciation, that is called phonetic description, we use special symbols that are called *phonetic symbols* (Halle, 1984). These *phonetic symbols* belong to the *International Phonetic Alphabet*. The science of phonetics analyses and describes the linguistic sounds, the phonemes (Abercrombie, 1967).

  The study of the phonemes' function within a particular linguistic system is called phonology and is differentiated from phonetics. Phonology studies the allocation of the phonemes and their contribution to communication and the phonological phenomena. The allocation of a phoneme is the linguistic environment in which it exists, that is, the elements that exist before and after the particular phoneme that we study, for example, we have the words "πίνω " (drink) and "τείνω" (tend) that with the use of phonetic symbols are described as [pino] and [tino] respectively. As we observe, these two words are differentiated only by the first phoneme, p and t. Therefore, the colloquists understand from these words two different meanings when they hear them, because the first phoneme is different.

An introductory general book about phonetics is the book of (Abercrombie, 1967). Also, a book about the useful sources of the genetic phonology is the book of (Chomsky, 1968). The books of (Durand, 1990) and (Roca, 1999) presents a more modern introduction.

- *Morphology*

Morphology is concerned with the internal form of the word. The word constitutes the basic unit both in the syntactic and the lexical level, while the words are not the minimum units of this level. In many cases the words are composite units and their elements have a specific meaning and thus they function as units in the syntactic level. These word elements are also observed in other environment, either alone or with other elements (Philippaki, 1976). Also, there are elements that cannot be analyzed any further.

A book with the more contemporary speculations around the field of morphology is the book of (Spencer, 1991). Another similar book is the book of (Selkirk, 1982). Also the book of (Haspelmath, 2002) presents a broad range of morphological phenomena from a wide variety of languages.

- *Syntax*

The syntax is concerned with the rules under which the words are combined in bigger structures, like the phrases, the sentences and the utterances. With the term utterance we mean the sentences within a text.

A first report in the term grammar of the phrasal structure is made in the first book of Chomsky Syntax Structures in 1957. The book is for the standard theory (Chomsky, 1965), while for the extended standard theory is the book (Chomsky, 1970), where we have for the first time the theory of the x-bar. The government and binding was originally developed in the book (Chomsky, 1981), while in the books (Theofanopoulou, 1989a, 1994) and (Philippaki, 1992) we have a generic consideration around the transformational syntax.

- *Semantics*

Semantics (Babiniotis, 1985) is concerned with the meaning of the words within the sentence and also with the meaning of the sentences.

A very useful introductory book on semantics is that of (Lyons, 1981) and on formal semantics (Cann, 1993).

- *Pragmatics*

Pragmatics is concerned with issues regarding the meaning of the sentences, as these are interpreted in a specific place and time, as well as in the terms of certain application fields.

A general introductory book which concerns pragmatics in particular is that of (Leech, 1983), while the book of (Philippaki, 1992) is about linguistics in general with important reports on pragmatics (Wirth, 1985). Also, another book that combines meaning with context is (Cruse, 2004).

The standard linguistic theory that has influenced linguistics since 1957 up today, is the one that mainly Chomsky has developed in 1957 and still does up today. This theory is known as *generative transformational grammar*.

The following can be considered as the basic stages in the evolution of this theory:

1) the standard of the *Syntactic Structures* in 1957 (Chomsky, 1957), that constituted the base for any further development.

2) the *Standard Theory* standard (Chomsky, 1965)

3) the *Extended Standard Theory* with its allocated realizations, that constitutes an evolution of the *generative transformational grammar* during the 70's (Chomsky, 1970) (Chomsky, 1972) (Chomsky, 1976).

4) The *Government and Binding* standard in 1981, that has been developed in the terms of the *Extended Standard Theory* (Chomsky, 1981, 1982) and for the Greek language (Theofanopoulou, 1994) (Philippaki, 1992).

5) Since the beginning of the 90's, we observe once again a new tendency to modify the standard, having as a starting point more general questioning about the role that the *principles of economy* play (Chomsky, 1988, 1995, 2000) (Samuel, 1999) (Belletti, 2002).

The evolution of the generative transformational grammar theory (Radford, 1981, 1988, 1997), resulted in a substantial turn for a rewriting rules system (Jacobs, 1970) towards a generalized system of universal principles that, with their coordinated co-operation define the organization of the language elements in every level.

Two are the main points that resulted in the evolution of the generative transformational grammar:

a) The attempt of the researchers for a simpler theoretical pattern based on a small number of generalized principles. Their target is the definition of abstract logic principles that are so general, that they cannot always be found in a direct response to the empiric data of the several languages. This took place only after the difference between the *internalized language* and the various language realizations was verified. This tendency leads to the formation of the *Universal Grammar* theory (UG) with the parametric diversities according to the various language realizations.

b) The parallel finding that many phenomena that up to now had been considered as different (e.g. the transformational rules and the binding rules) have been proved the result of common principles function. More specifically, the binding of elements, like the reflexive pronouns with their reference point and the binding of the movable nominal components with the trace that can be found in the place from which they were moved are controlled by common principles.

So there was the alteration of the transformational theory fundamental senses and, as a result, its modification from the standard of transformational interrelation of its deep-surface structure phrase markers to the abstract total of principles and parameters that constitute what is generally called *Universal Grammar*.

According to Chomsky, the existence of the Universal Grammar, that is, common linguistic schemes in all the languages of the world is based on the following:

a) the existence of common *abstract principles* (general limitations, generalized structure patters) in the systems of the several languages, despite the superficial diversity that the several languages seem to have.

b) the data of the language acquisition, where the following have been observed:

> 1) Language learning presents a unified form to all people. From experiments that have been conducted to the most uneven languages it has been proved that the stages through which every man passes during the learning of the maternal language in all its levels are basically the same for every language. The uniformity of the *language acquisition* is defined by the existence of universal linguistic elements that may refer to the way that the perceptive mechanisms, that the child used to analyze his/her language, function. Also, the child learns the language of the community in which he/she belongs naturally and effortlessly and this happens regardless its intelligence.
> 2) The perfection and speed of the language learning also count for its inherent character. It is amazing how man manages in a rather short period of time (within the 3 or 4 first years of his life) to conquer the basic system of his language.

According to Chomsky's conclusions (Chomsky, 1986a), the following applies:

> "The universal grammar is a theory of the linguistic ability initial state, before any linguistic experience"

The more exponent representative of the *universal grammar* is the theory of *government and binding*. This theory pays special attention to two basic principles, the *government principle* that describes the syntactic dependencies between the various lexical elements within the sentence, and the *binding principle*, that explains how the several elements are inter-connected in the sentence.

This theory has developed a set of several Principles. These principles include the following allocated theories:

a) *the government theory*
b) *the binding theory*
c) *the bounding theory*
d) *the Θ-theory*
e) *the case theory*
f) *the control theory*

The *government theory* defines the principles concerning the relation between the head of a structure and the terms that depend on it. The principles also concern the case of the empty categories and the problems that derive. While the *binding theory* is concerned with the conditions that control the way of *binding* an anaphoric element with its reference point in a natural language tree. Another theory is the *bounding theory* that sets the terms that bound locally the transfers, defining which nodes are constrained in their transfers and under which conditions. The *thematic roles theory* includes principles that define the semiotic function (thematic role) of a name phrase (NP) (if this phrase declares the action taker, the receiver, the theme, the instrument, the place, etc). This theory refers to the terms that control the determination of a thematic role: level, kind of position, as well as the cases where this is impossible. The *case theory* includes the principles that define the "abstract" case in a NP (when a NP is characterized as nominative, accusative or possessive) and the conditions that must exist in order for this theory to be fulfilled. While finally, the *control theory* defines the terms that control the presence of the empty PRO category, a fact that has been in question in the Greek language. It is namely concerned with the empty positions in the tree that are not created by the elements' transfer.

Since the beginning of the 90's and after, we observe another new tendency to alter the standard, having as a staring point general speculations about the role that the *economy principles* play in the formulation of the theoretical principles and the description of the language structure. The standard is simplified, including now a pair of two levels regarding the phonological and logic structure. The syntactic sectors are limited to the lexical and calculating department of production, while at the same time transformations function, that control the alternation of the phrase markers, the trees.

The structure of the phase marker components and the form of the rules are according to the convention of the x-bar (Theofanopoulou, 1989a) (Theofanopoulou, 1994) (Philippaki, 1992). This convention goes back to (Chomsky, 1970) and it was shaped with the works of (Jackendoff, 1977) and today constitutes the established method of the structural depiction, making the phrasal structure rules of the older standards unnecessary.

The general figure is as follows:

a) X'' → Spec X'
b) X' → X Compl

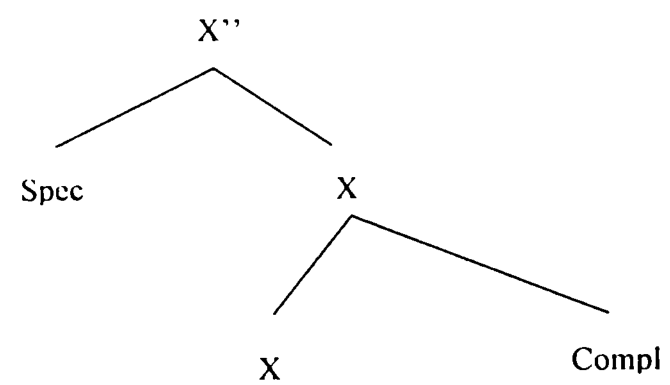
The X represents one of the main lexical categories such as the noun, the verb, the preposition and the adjective. The tone represents the level and we have the X'', X'

and X. The X'' is the *biggest projection* (phrase) for every lexical category, the X' is the *intermediate head* and the X is the *lexical head*.
Every biggest projection is analyzed in a *specifier* (Spec) and in the *intermediate category*. Every *intermediate category* is analyzed in the *lexical head* and in the *complement* (Compl). There is also the possibility to repeat X' under the following rule:

c) X' → X' Compl

The *complement* and specifier are of X'' category, they derive from the general form of the x-bar. Also, the *specifier* can be a node of the X category with the terminal that is connected to it.

The *X-bar tree* is the following:



In the recent studies the basic figure has been simplified (Haegeman, 1995) and replaced by the following rules:

a) XP → Spec X'
b) X' → X Compl

The above are some of the basic elements of the generative transformational grammar that we shall analyze in detail in the chapter about the X-bar theory that follows.
Also, there are parsers (Shaban, 1994) (Fong, 2000) that implement the Chomsky's government and binding theory and the minimalist program.

Finally, since language is a basic element in a series of human manifestations, linguistics co-operate with other sciences so that several branches of linguistics have derived. Next it follows a short description of the content of some of the main branches of linguistics (Babiniotis, 1980) (Philippaki, 1982, 1992).

Psycholinguistics studies the relation between the language behavior and the psychological mechanisms.

Sociolinguistics studies the ways in which language is affected by the social differences among the members of a linguistics society.

Stylistic linguistics examines the selections of a litterateur in certain texts.

Mathematic linguistics examines the mathematical properties of the language.

Computational linguistics studies the language with the use of computers aiming to confront a series of subjects such as the automatic translation, the information retrieval or the general development of the artificial intelligence.

Clinical linguistics uses the linguistic theory mainly in order to study problems in pronunciation or writing.

The main applications in the *computational linguistics* are:

- *Machine translation*

The automatic translation (Ananiadou, 1990) (Efthimiou, 1991) with the use of a computer is an application with great interest since the era of the cold war. A characteristic program is the Translearn/LRE that had as a target the development of an automatic translation tool that would not give high quality translations. The basic purpose of this system is to relief the translator from the recurrent parts in his job, mainly in special technical texts, as well as to raise the quality of the final product, by helping the translator, providing him with alternatives for every text (Gabriilidou, 1990). This system is based on extremely developed techniques that use linguistic and statistic information in order to define the bigger related text that has already been translated and stored accordingly in the system's text base. The text part that will be translated is given to the user for the appropriate corrections that he would wish to make, as well as for the confirmation and acceptance of the final result. Another characteristic and well-known program is the EUROTRA (Alshawi, 1992) (Schnelle, 1992) that has as a purpose the development of a machine translation system among the languages of the European Union member states. Another system for automatic translation that the European Community has developed for its internal needs is the SYSTRAN. This system provides translation services in 16 language pairs of languages. Generally, SYSTRAN can be seen as a tool for a first translation and is particularly quick since it can provide up to 2000 pages per minute.

- *Informational retrieval*

The informational retrieval from natural language texts is another extremely interesting application of the computational linguistics. The reason is that since the biggest part of the information lies in books, magazines and references, it is necessary to retrieve it from them. A program developed currently by the Greek Institute of Parole Processing is concerned with the collection of Greek multiform texts. This system creates a text base that is used from publishing organizations and linguists researchers for their studies. The body of the texts is accompanied also by computational tools that give the possibility to draw information from them as well as to process them linguistically. Another information retrieval program is the RENOS/LRE that had as a purpose the development of methods and tools in order to improve the performance of a full text recall system through the addition of linguistic information.

- *Man-machine interface*

The natural language is also the best case in order for man to communicate with a certain computational system, especially for people that do not have special knowledge on how to communicate with a computer. In 1983, Filgueiras presented a core of a general communication system between man and a computer through natural language.

Also, computational linguistics, while trying to achieve the development of systems that would perform a complete translation, proceeded in the study of several scientific domains that had not been investigated. For example, they were concerned with computational models that imitate the human reaction in the understanding of sentences and they were also concerned with computational models that represent knowledge.

Comparing the domains that the linguists and the computational linguists are concerned with, we observe that their interests are different. Also, we see the usefulness of the results that derive from the research of the theoretical linguists (Kosma, 1988)(Mackridge, 1985) on the problems of the computational linguistics.
Computational linguistics tries to find solutions that would cover the categories of the sentences we are concerned with for every application, while theoretical linguistics is concerned with issues such as:

- How people accept certain sentences as grammatically correct and others as incorrect
- The principles of grammar that can be applied in every natural language.
- The mechanisms with which people are able to learn and use the natural languages.

Independently of the differences between theoretical and computational linguistics, the theoretical linguistics (many stydies there are about the greek language (Philippaki, 1970, 1971, 1973, 1975, 1985) (Photopoulou, 1990) (Ralli, 1990a, 1990b)) is very useful for the computational linguistics (Ralli, 1992). The existence of a certain constraint, for example, that defines the grammatical correctness of a sentence is very useful because it will give us the ability to select among different syntactic analyses the correct one. Also, the abilities of the sentences transformation enable us to deal with a total of sentences that will present similar tree structure.

After the grammar of the phrase structure was developed in 1957 by Chomsky (Chomsky, 1957) it was defined that in order to produce sentences in a natural language, rewriting rules must be set. For example, for the formation of a simple sentence with subject and object we can set the following rules:

$S \rightarrow NP\ VP$
$NP \rightarrow A\ N$
$VP \rightarrow V\ NP$

Where S is the sentence, NP is the nominal phrase, VP is the verbal phrase, A is the article and N is the Noun.
The rules that have been used by the computational systems (Gilbert, 1991) examined also the environment of an element, e.g. of a NP that would be replaced by the application of a rule.

In posterior publications of the generative transformational grammar, the standard of the grammar was becoming more and more abstract. The tree of a phrase or a sentence derives now from the x-bar figure and the rules that theory sets have a general form and describe the laws that such a tree should fulfill. The generality of the description of the generative transformational grammar rules and principles has as a consequence the non-utility of the theory in computational systems used to process the natural language, because it made their description in some systematic and standard way very difficult.

Thus, the grammars used for the computational process of natural language use mainly rewriting rules in order to describe all the rules, either these are rules for the production/processing of the sentences or rules for the transformation of the sentences(Pedersen, 2000). Also, systems with network construction have appeared for the processing of sentences or phrases. Some well-known networks are the RTN and ATN. A detailed presentation of these methodologies can be found in the books of (Gilbert, 1991) and (Noble, 1988).

In the present doctorate dissertation (thesis) there was an attempt to develop a new systematic methodology that gives us the ability to define typically the rules of the generative transformational grammar in general (x-bar theory) and more gerenal other linguistic rules. The methodology leads to the development of the respective software. The result of this attempt is that the aspects and the research conclusions can be used and applied directly. It is a research effort that leads to a new artificial language that integrates the ideas of other theories in a more general and abstract way by presenting some new ideas.

Until today, most of the natural language processing systems used the rewriting rules that Chomsky had proposed in 1957. These rules are also used to describe the typical languages. Thus, by using this methodology, they were trying to solve all the problems of the natural language processing. However, with the evolution of the linguistic theories, a new basic scheme was developed, and all the trees of the sentences or phrases of any natural language derive from that scheme. The basic scheme is the x-bar scheme that we saw above but we shall analyze in the next chapter. This gives us the ability to deal uniformly with all the trees of a natural language, since all derive from the same basic scheme, an ability that we didn't have with the rewriting rules or other grammars and methodologies (Fouskakis, 2004b, 2005b) used to process a natural language in a computer. The linguists set also rules and a series of subtheories was developed, regarding the structure and content of the natural language trees that derive from the x-bar schema.

The above show the great value of a systematic methodology for the definition of the linguists' rules (Fouskakis, 2000, 2004a, 2005a) that could be applied on natural language trees that derive from the x-bar scheme. It must permit the definition of fewer and more general rules that are applicable in many trees since they are derivations of the same tree.

This methodology differs from the classic use of the rewriting rules for the development of natural language processing systems that use the linguists' conclusions and aspects.

The methodology that was developed is open to the changes in the linguists' theory and enables us to set the rules that are necessary each time. These rules are set in a simple way, while they are also more descriptive. Also, we are enabled to deal in a general and uniform way the several issues of the natural language trees.

The respective system that was developed is a very useful tool in the linguist's hands in order for him to study several rules and sub-theories in practice, applying them on trees that derive from the x-bar.
Also, this system can be used as a sub-system in a natural language processing system, since can describe also rules that haven't been formulated by the linguists in their theory.

The rules that one can set in the present system belong in two categories, the principles and the transformations. The principles study the structure and the content of the natural language trees that derive from the basic x-bar schema, while the transformations modify the structure of the trees and the contents of their nodes. Both the principles and the transformations apply on sub-trees of the natural languages sentences or phrase trees. These sub-trees are described in the declaration of the principles and the transformations rules. This fact enables us to describe more accurately the rules and the cases where each of these rules is applied in a way corresponding to the respective rules of the theory.

Also, the methodology enables us to describe sub-theories. Each sub-theory uses certain principles and certain transformations that we have already defined. Each sub-theory uses these rules according to the sequence and the conditions that have been setted in this sub-theory. A sub-theory can also use some other sub-theories that have been defined.
Also, the present methodology enables us to define which of the rules and in what sequence are going to be applied on the natural language trees that are under processing.
The methodology that was developed enables us to easily expand, modify and reuse the defined rules according to the situation, without the requirement for big and complex changes in the total of these rules.

2. The stage of knowledge

2.1 The X-bar theory

2.1.1 The School of Structuralism

The last forty years, since 1957, there is a special attempt to study the syntax, as well as a big turn regarding the older *school of structuralism* in the study of the language that appears mainly in the beginning of our century by Saussure.

Later on, we shall present some basic principles and positions of Saussure (Babiniotis, 1980).

The first important turn of Saussure, was to see the language as a communication instrument among the members of a language society and since through language communication is achieved, both in written and in its spoken form, it should comprise a system. The elements and symbols that uses function regularly and systematically and so language has a *structure* that the linguistic theory has to discover and describe.

Also, according to Saussure, there are two possibilities to describe the language system, the *synchronic* and *diachronic*. Using the synchronic description of language, the linguist describes language in a given moment in time, as this is presented in a language community. Using the diachronic description, the change of language is described from a previous to a later stage of the same language, since language, through time, goes through changes. According to Saussure the synchronic study of the language is more important, because this is also the condition for the correct diachronic study. The synchronic study of the language is a reaction towards the traditional grammar that ignored the modern language form and studied the previous form of the classic languages. Traditional grammar attempted to teach a language form that had been idealized for various social or esthetic reasons, while Saussure points out that the linguist should not act regulatory but he should describe objectively.

Another essential distinction of the *language* is in *langue* and *parole*. *Langue* is the abstract language system that all members of a language community possess in common and this system enables them to communicate among each other. *Parole* is the specific application and exploitation of the language system by every person of the community in communicating to the others. This distinction between langue and parole has a great methodological value and stresses that a linguist that wishes to approach langue proceeds ablatively, based on the parole data.

Also, according to Saussure, the language or rather the langue comprises of a sign system. The sign is a connection between two things: the meaning (the signifier) and the acoustic image (the signified).

2.1.2 The Standard of the Syntactic Structures

The big "revolution" regarding the school of structuralism and Saussure, was made by Chomsky in his book "Syntactic Structures" (Chomsky, 1957).

The structural standard of the language description and analysis is characterized by the absolute attachment to a subtotal of data, the application of a strict hierarchy in the analysis of the levels of grammar and by the use of finding procedures for the definition of the minimum units in every level (phonemes, morphemes, syntactic categories). Unlike the above theses of the structuralists, Chomsky regards grammar as a mechanism that produces an infinite number of only grammatically correct sentences. (Chomsky, 1957) (Theofanopoulou, 1989a, 1994) (Philippaki, 1982, 1992).

Chomsky's standard theory in 1957 is that the language is considered as a total of sentences that each of them has a finite size and is structured by a finite total of elements. The basic target of the linguistic analysis is the differentiation of the grammatical sequences that are language sentences from the un-grammatical ones, as well as the study of the grammatical sentences structure.

Through out the whole work of Chomsky, the basic question is how can we know if the each time proposed standard of grammatical description is adequate or not.

Thus we have several levels of adequacy that we shall learn below.

The grammar of a language is *observatorily adequate* if it can predict correctly which sentences are formed correctly or not regarding the syntactic, the semantic and the phonological level.

The grammar of a language is *descriptively adequate* if, apart from the above, it can also describe correctly the syntactic, semantic and phonological structure of the language sentences, in such a way that it can correspond theoretically to the intuition that the natural colloquist of this language has for its structure.

The grammar of a language is *interpreteraly adequate* if it, apart from the above, the description is based on general theoretical principles that are simple to describe, limited in number and universal. These principles represent psychological and intellectual human principles that depict the way in which a child can learn effortlessly, naturally and in a short period of time the language of his/her community, based on the fragmentary data to which he/she is exposed.

A standard of grammar, that fulfills the conditions for an interpretative adequacy, is based on the existence of universal characteristics. The universal characteristics define the way in which the language acquisition is being made.

Chomsky's contribution in the theory of syntax does not lie only to the reconsideration of the language theory purposes and to the foundation of the principles of a general descriptive standard that is subjected to certain adequacy conditions. Already, in his work "Syntactic Structures" (Chomsky, 1957), aims at the standardization of the principles that produce sentences, by using the methodology and the symbolism applied in the analysis of the typical languages.

Let us see an example with which he standardized the description of language production.

Suppose that we have the following natural language sentences:

a) the child reads the book
b) the teacher drives the car
c) the farmer ploughs the field

During the first phase of the transformational grammar (Chomsky, 1957), in order to describe the above sentences, we should set the following rules:

S -> NP VP
NP -> A N
VP -> V NP
A -> the
N -> farmer, teacher, child
V -> reads, drives, ploughs

Where:

S	Sentence
NP	Noun Phrase
VP	Verb Phrase
A	Article
N	Noun
V	Verb

These symbols are called *non-terminal*, while the "the", "child", "teacher", "farmer", "reads", "drives" and "ploughs" are called *terminal*.

This standard of description corresponds to the analysis of the sentence in direct components that are arranged in hierarchy and this standard that had been adopted by certain structuralists. The basic characteristic of the above rules is that they are applied in a certain sequence. All the rules, apart from the first one, are applied at the end of the last rule, where every symbols is replaced (rewritten) by the application of one of these rules. Also, all the rewriting rules do not take under consideration the neighboring symbols of what is being rewritten. Therefore, we have a grammar that is context free.

Each one of the above rewriting rules has the following general form:

$$X \rightarrow Y$$

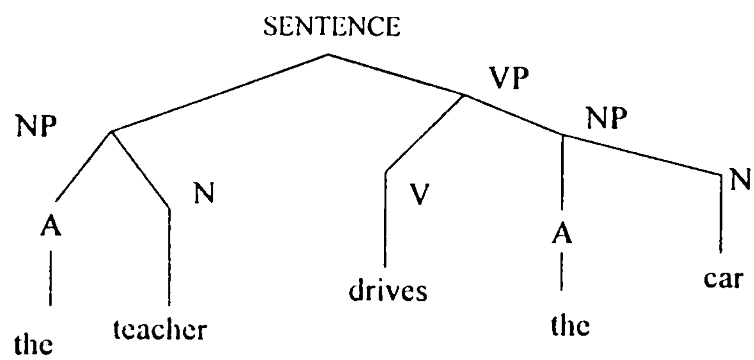
where X represents a sign (symbol) and Y can be one or more signs (symbols).

The sentence production procedure with the rewriting rules creates a tree. This tree is called also *phrase marker*. The *phrase marker* contains explicit information about the hierarchical construction of the sentence components, while senses like the subject are defined structurally.

Suppose we have the sentence:

The teacher drives the car

The respective phrase marker is the following:



The standard of the grammatical structure that Chomsky presented in 1957 in his book "Syntactic Structures" approaches more the structural way of description.

This standard includes the three following levels:

1) the *phrasal structure* level
2) the *transformational* level
3) the *morphophonological* level

with the respective rules for each one of them.

The *phrasal structure rules* have the following general form:

$$X \rightarrow Y$$

which we have already presented in detail above.

These rules function independently of the environment and their function produces a finite total of terminal elements, if of course no rules of *recursion* exist.

The *transformational rules*, which transform the structural level of the sentence that has been produced by the phrasal structure rules. A characteristic transformation is the transformation of the passive voice. The transformational rules are divided in *obligatory* and *optional*. An obligatory transformation is for example, in the English language, the use of the auxiliary DO in a question or negation. An optional transformation is the transformation of passive voice.

The *morphophonological rules* perform morphophonological changes.
For example: tie + past tense → tied

2.1.3 The Standard Theory

After the introduction of the *Standard Theory* (Chomsky, 1965) the articulation of grammar changed. This new standard became the reference point of the later evolutions of the grammar theory.

The changes that the *Standard Theory* introduced regarding the previous standard were the following:

1) The extension of a syntactic field that is now distinguished in the *deep structure*, the *transformations* and the *surface structure*. These perform the production of the sentence.
2) The consideration of the *recursion* as a part of the phrasal marker and not as a part of the transformations as it used to be in the standard of the syntactic structures (Chomsky, 1957).
3) The addition of the semantic *domain* in grammar that defines the semantic interpretation of the sentence.

According to this standard grammar comprises of the following components:

1) the *syntactic* component
2) the *phonological* component
3) the *semantic* component

The syntactic component, that is the basic component of grammar, can be divided in the following parts:

A) The base
The base includes the phrasal structure rules and the dictionary. The phrasal structure rules correspond to those applied in the Syntactic Structure rules (Chomsky, 1957).

The lexicon contains a list of the language morphemes as well as special information referring to their phonological disposition and their syntactic function. There are three kinds of lexical features. The first kind is the category, such as Noun, Article, Verb. The second one refers to the category environment of the word. For example, the verb through contains the information [+NP], meaning that the verb is used as transitional, having as a complement a NP (nominal phrase). Finally, the third category of features is the selection features that are related to the general frame in which the word can exist. These features can give semantic information. For example, the feature [+animate] that describes the action taker (subject) in a verb means that the action taker is an animated being.

B) The deep structure

The sentence produced by the function of the phrasal structure rules and the addition of the dictionary morphemes, constitutes the deep structure of the sentence. According to the standard theory, the semantic interpretation of the sentence lies in the deep structure, where the functions of the several terms of the sentence are defined structurally.

C) The transformational scheme

The transformational scheme of grammar with the transformational rules, deletes, adds or transfers elements in the deep structure and thus the surface structure derives.

D) The surface structure

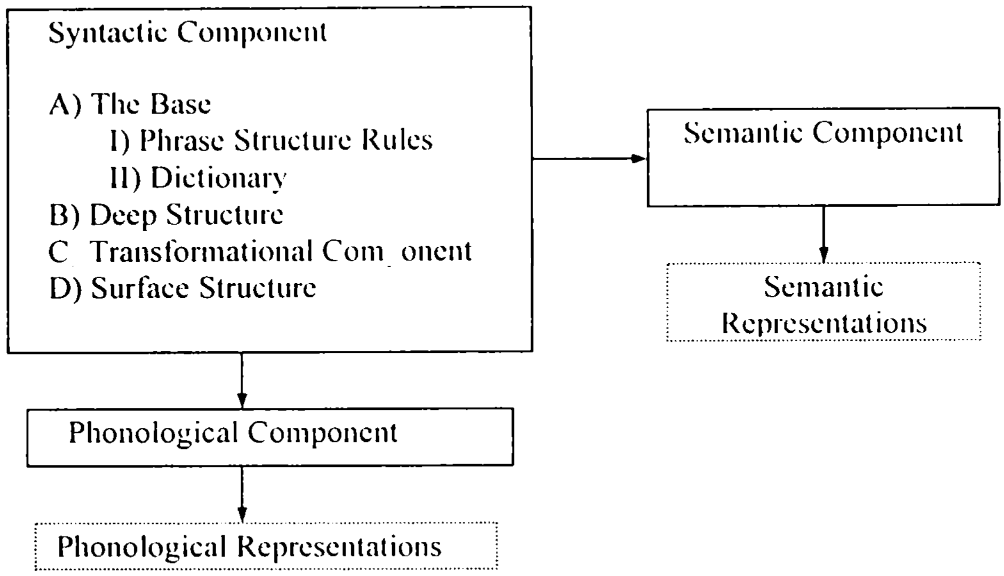
The surface structure is the result after the application of the transformational rules.

Finally, according to this standard, the phonological and the semantic component have an interpretative character.

The semantic component includes a lexicon that doesn't include only the syntactic features and the frames in which each word exists, but also all semantic characteristics as well as all the rules with which the meaning of the sentence is defined, according to the meaning of each word.

The phonological component defines the phonological form of the sentence that has derived from the syntactic component, according to the elements that exist for the words of this sentence.

The schema of the standard theory (Chomsky, 1965) is the following:



2.1.4 The Extended Standard Theory

Following the *Standard Theory* (Chomsky, 1965), we have the *Extended Standard Theory* (Chomsky, 1970), where we observe gradual changes that depict the tendency for greater generalization and apheresis (subtraction).

The changes in the *Extended Standard Theory* (Chomsky, 1970), are the following:

1) The phrasal structure rules subject to the X-bar (Jackendoff, 1977).

2) The transformational component of grammar is limited to only one generalized transformation (Move a). In this lie the transformational rules that are known from the previous stages of the theory, such as passivisation and question.

3) The function of the transformational rule (Move a) and the function of the sentences interpretation rules are regulated by constraints that are general and universal. If these constraints are violated, they lead to the formation of ungrammatical sentences.

4) The transformations, during their function, leave traces that result during the transfer of the tree elements and remain in the place where the transferring element was occupied. The trace and the transferring elements are connected to each other.

5) In all the formation levels, empty component may arise, namely components without any phonological content. These empty elements create problems and several studies are conducted regarding these cases.

6) The interpretation of the sentence is not conducted any more in the deep structure but in the surface structure. The surface structure is more complete because the place of the transferring elements can be seen from their traces.

7) Finally, the lexicon that now is enriched and extended is very important. It also includes, in relation to the previous phases of the theory, rules with which the composite and derivative words are formulated.

Therefore, between the Standard Theory (Chomsky, 1965) and the Extended Standard Theory (Chomsky, 1970) there are differences regarding the structure of the original phrase marker, the form, the kind and the function of the transformational rules. Also, the content of the lexicon has changed. Regardless though of these changes, the purpose remains the definition of the structural correspondence of the phrase markers between, e.g. interrogative-affirmative or active-passive sentences. The definition of the structural correspondence is conducted with the definition of the transformational rules with which the two levels of the sentence are connected, that is, the deep and the surface structure, as well as with the parallel examination of the way these rules function and interact, that is, the cycle and the sequence of their function.

The construction of the phrase marker components and the form of the rules are according to the convention of the X-bar (Theofanopoulou, 1989a) (Theofanopoulou, 1994) (Philippaki, 1992). This convention that goes back to (Chomsky 1970) and was formed with the works of (Jackendoff, 1977), constitutes today the established way of the natural depiction of the several categories, making the phrasal structure rules of the older standards unnecessary.

The general pattern has as follows:

a) X'' → Spec X'

b) X' → X Compl

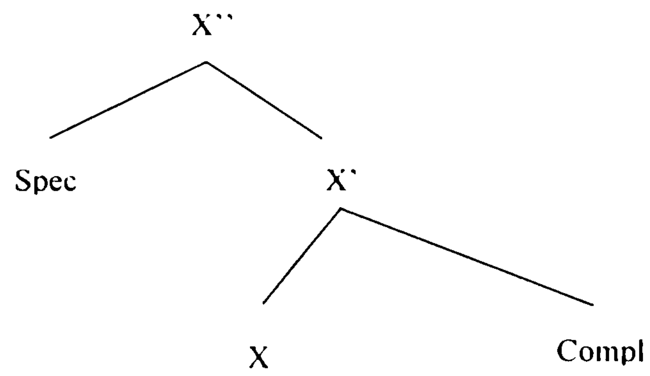
Where X denotes one of the main lexical categories such as the noun, the verb, the preposition and the adjective. Also, X may state one of the functional categories. As functional categories we regard the inflection and the supplementary marker. These functional categories are not the only ones, but continual research leads to new ones.

The tone denotes the level and we have the X'', X' and X. The X'' is the *biggest projection* (phrase) of every lexical category, the X' is the *intermediate head* and the X is the *lexical head*. Each biggest projection is analyzed in a *specifier* (Spec) and in the *intermediate category*. Each *intermediate category* is analyzed in the *lexical head* and the *complement* (Compl).

There is also the possibility to repeat the X' following the rule below:

c) X' → X' Compl

The *complement* and the *specifier* are of the X'' category, meaning that they derive from the general pattern of the x-bar. Also, the *specifier* can be a node of the X category with the terminal connected to it.

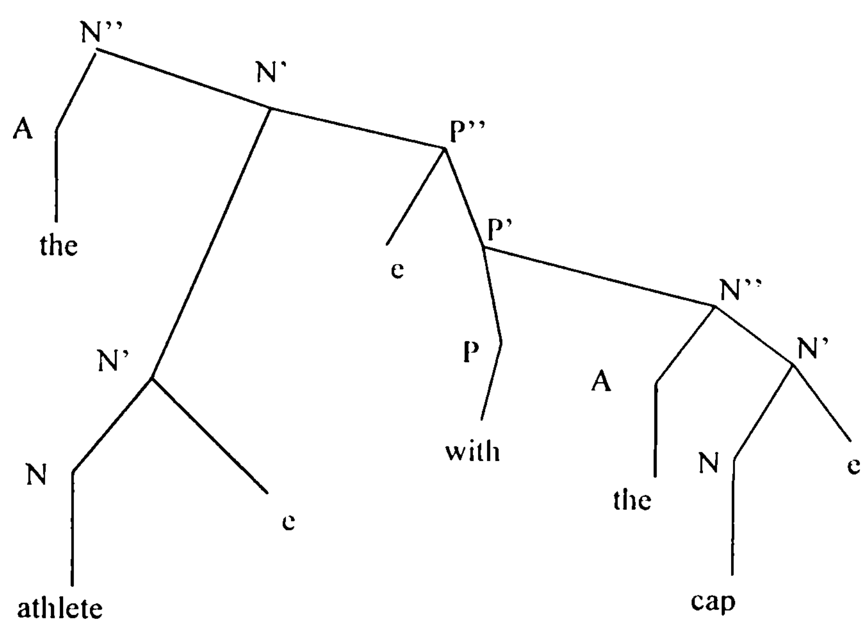


Next we shall give a series of examples.

Example 1

We have the sentence:

The athlete with the cap



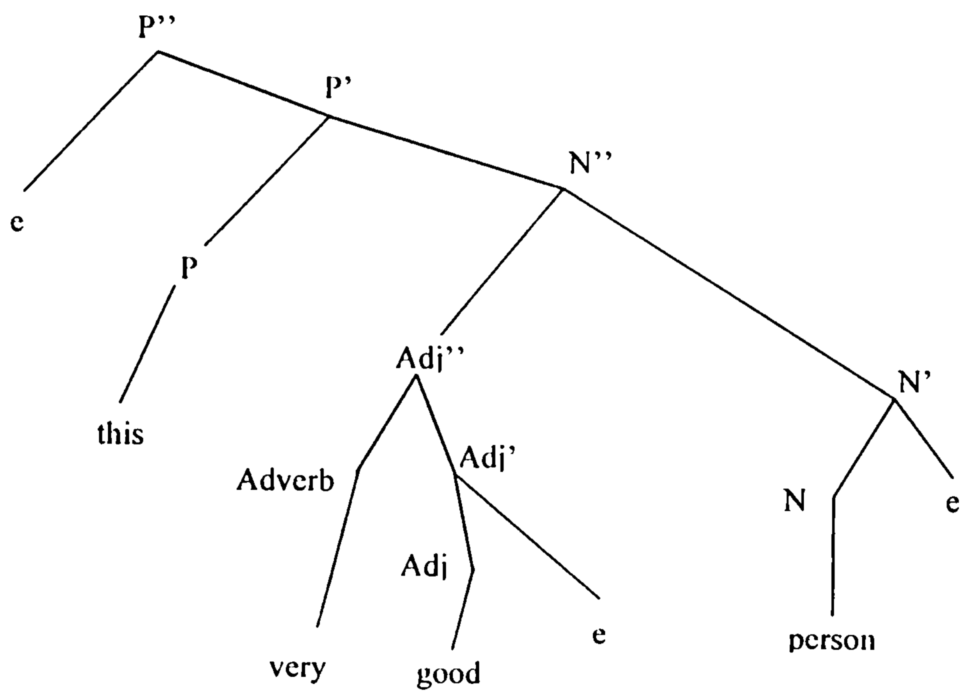
This nominal phrase has the word athlete, which has a complement the phrase with the cap. This complement specifies the athlete that he is an athlete who is wearing a cap. Therefore, we assign a feature to this particular athlete. This feature is complement of the athlete's nominal phrase.

Observing the tree, we can see that its top is in N'', therefore this tree is a nominal phrase. The left subtree is the article "the" and the right one is of the N' category. We also observe that since there is a complement, the phrase "with the cap", we see a second repetition of the node N' on the above tree. The first N' node has as a right sub-tree a sub-tree with the P'' as a top node, this is the prepositional phrase "with the cap". This phrase can be analyzed in the tree that has the P'' as a top, a left sub-tree the empty space and a right sub-tree the one with the Prep' as a top. The latter has as a left subtree the preposition "with" and as a right subtree the nominal phrase "the cap". (The cion the trees shows that there is no element in the corresponding place of the tree.)

Example 2

We have the phrase:

this very good person

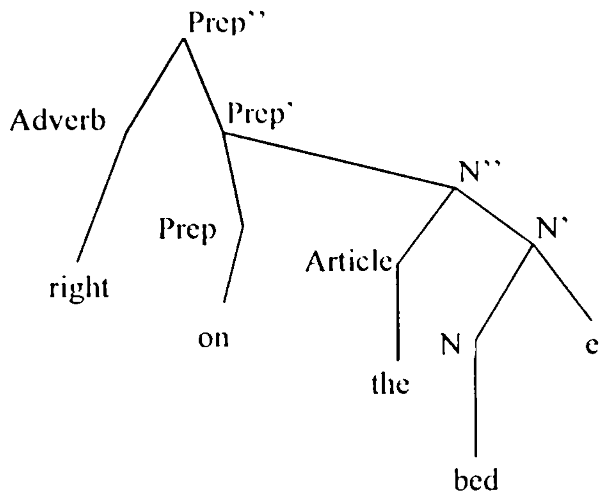


This tree has a top with the P'', that has a left subtree the empty specifier and right sub-tree of type P' with left subtree the P and right subtree N'' with left subtree for the phrase 'very good' and right subtree a N' for the word 'person'.

Example 3

We have the phrase:

right on the bed



This tree has as a top the Prep''. The adverb right is the left subtree, while the Prep' is the right subtree, with "on" as the left element and the tree for the noun phrase "the bed". This phrase is a prepositional phrase that has as a head the word "on" and as a complement the nominal phrase "the bed".

2.1.5 The Government and Binding theory

Two are the main parts in the development of the generative transformational grammar theory. These parts contributed to the decisive turn towards a generalized grammar standard.

1) The finding that the binding phenomena and the phenomena concerning the moving of the terms are controlled by common principles. A binding example is the connection of the reflexive pronoun to the reference point, meaning the respective word to which it refers. Another example of the moving conditions is the connection of the interrogative pronouns to the trace that can be observed in the place from which they were moved.

2) The definition of generalized constraints, not in the particular rules more, like for example the constraint in passivisation or the move of an anaphoric element, but also in structural schemes. These are very general principles that control the relation of interdependent conditions, like the relation of an anaphoric element with its reference point and the terms in which the bindings are performed.

These two developments resulted in the modification of the fundamental senses of the transformational theory. The transformational theory ceases to be a standard of transformational interrelation of the deep and surface structure phrase markers. It was altered in a generalized theory of allowable bindings to their reference point with the parallel definition of universal constraints that exclude such a connection, regardless of the way that the sentence was formed.

This grammar, known also as the *Government and Binding* theory, was introduced by Chomsky in his work "Lectures on government and binding" (Chomsky, 1981)(Haegeman, 1990). According to this standard, grammar includes two system categories: a *system of rules* and *systems of principles*.

The system of rules includes rules that function in the various levels and they generally correspond to those of the previous stages (phrasal structure rules, transformational rules, interpretative rules etc).

The systems of principles include sets of theoretical principles that refer to allocated structures of grammar and are interdependent both to each other and to the theoretical frame. This theory, although it constitutes a further phase of the generative transformational grammar development, presents an important difference regarding the previous phases of the theory's development. This difference is that for the first time an attempt is being made to define an abstract, generalized and universal system of principles which describes the language structure in general. These principles are so general that apply to all languages. But in order to solve the special issues in every language, there is need for another set of complementary principles (parameters). Thus we have a differentiation between the universal grammar and the several parameters needed for every language.

The title of this theory shows that it pays much attention to two basic principles, the *government principle* that describes the syntactic dependencies among the various word elements in the sentence, and the *binding principle*, which explains how two different elements in the sentence are bound.

This theory has developed a set of allocated principles that belong to the respective sub-systems, which include the sub-theories that we shall present next.

2.1.5.1 The government theory

The government theory defines the principles that concern the relation between the head of an x-bar structure and the conditions depended on it. The principles also concern the case of the empty categories and the problems that derive.

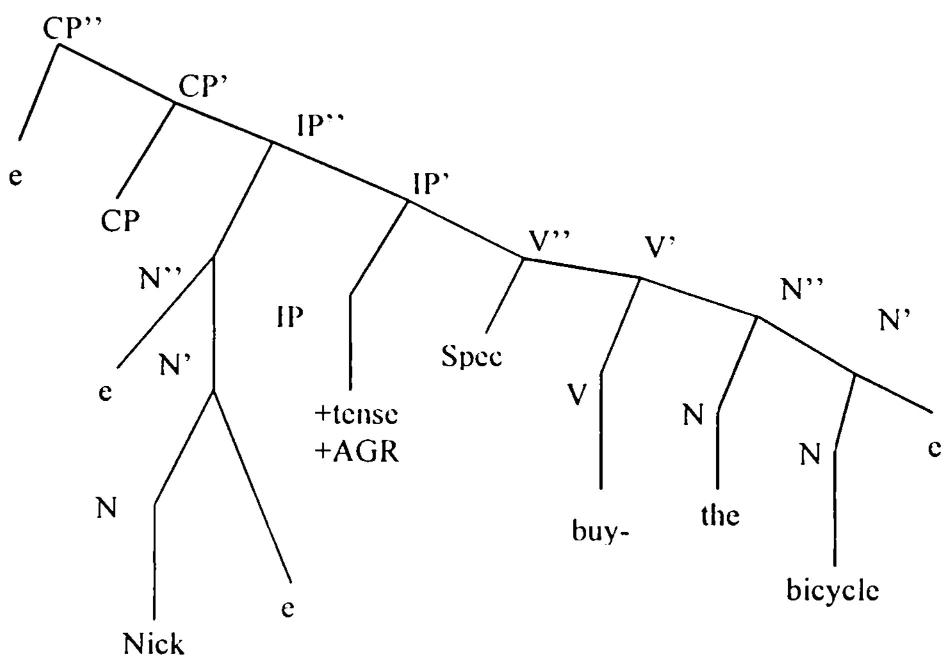
In the Government and Binding theory (Chomsky, 1981) the sense of government is very important and its definition is the following:

An X element governs a Y element if the first node of the biggest projection that dominates the X dominates also the Y and neither of these two elements dominates each other. If there is more than one governor we chose the one that is closer to the governable element.

On the above definition we used the sense of *domination*. According to this sense, a node *dominates* the nodes of its subtrees, that is, all the nodes that lie below this one on its subtree, where this node is the top.

Next we will see a government example in the following sentence:

Nick bought the bicycle



With the definition of government we observe that IP governs N'' in the qualifier of IP'', because the biggest IP'' projection that dominates IP, dominates also the nominal phrase of the qualifier and the IP is the closest governor.

2.1.5.2 The binding theory

This theory covers the area called *binding*. The binding theory refers to the conditions that control the way an anaphor is bound to its reference point.

This theory classifies the NP according to their anaphoric properties. The NP categories are the following:

a) *Compulsory Anaphors*, where the binding must be within the same structure. For example, we have in a sentence the binding of a reflexive pronoun with the NP. In the sentence "John admires himself" we have binding between "John" and the reflexive pronoun "himself".

b) *Pronouns*, that can be bound with a NP inside the same structure or obtain a reference point out of the sentence.
For example, in the sentence "George says that he loves Evaggelia" the pronoun he can correspond to George or to someone else.

c) *Independent anaphor*, where we have lexically expressed NP that each one has an independent anaphor. The heads of the NP refer to particular persons and things of the real and imaginative world.

The conditions that Chomsky established for the bindings that are also called *Binding Conditions* are the following:

A) An anaphor must be bound to its governing category.
B) A pronoun must be free within its governing category
C) A lexical NP must be always free everywhere.

These principles correspond to the three NP categories that we mentioned above.

Also, in Chomsky's establishment of principles, we used the term *governing category* that has the following definition:

> *Governing category* for an A element is the minimum nominal phrase or sentence that contains A, a governor for A and a subject, while this subject should be structurally higher towards A.

2.1.5.3 The bounding theory

The bounding theory sets terms that limit movements, defining which nodes are restrictive and under what conditions. The nodes on a subtree of a phrase or sentence may, in certain conditions, allow the movement of the subtree's elements in another place in the tree of this phrase or sentence.

A characteristic rule is the constraint of the *subjective bounding category*.

No rule can extract an element out of more than one bounding categories (Sentence or Nominal Phrase). Also there is a study about greek language (Horrocks, 1987) (Staurou, 1987).

2.1.5.4 The thematic role theory

The Θ-theory or theory of thematic roles includes principles that define the semantic function (thematic role) of a NP (i.e. the action taker, the receiver, the issue, the instrument, the place). This theory refers to the terms that control the apodosis of a thematic role: level, position, and the cases where this is impossible.

The main principle of the theory is the *Thematic Criterion or the Θ-criterion.*

Each term brings only one thematic role and every thematic role is attributed to only one term within the sentence.

2.1.5.5 The case theory

The case theory includes the principles that define the case of a nominal phrase (when a NP is characterized as nominal, accusative or genitive) and the conditions that must exist in order to be realized.

The case theory is based on the *case filter*
The *case filter* is the following:

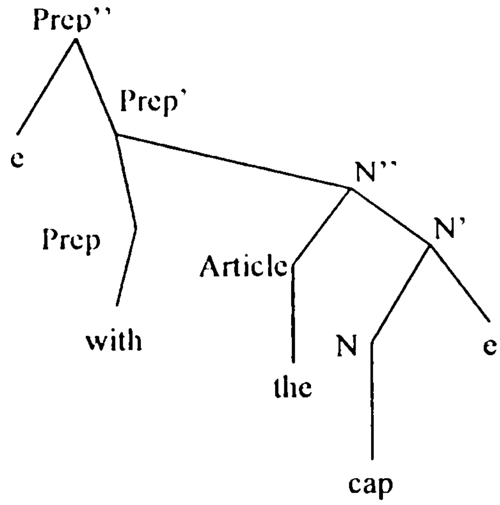
Case Filter

No Noun phrase (NP) can stand in a structure unless it bears a case.

The syntactic cases are of two kinds, the *structural* and the *inherent.*

The *structural* case is assigned in a NP by the elements that have the property to be *case assigners* under the governing conditions. Therefore, in this occasion, the case depends on the element that governs the element that will bear the case.

For example, we have the phrase "with the cap"



We observe that the Prep node governs the N'', assigning it an accusative structural case. That means that the NP "the cap" is in the accusative case, because of the preposition with.

Unlike the *structural case*, the *inherent* case is connected directly and exclusively to the thematic roles provided by the verb and not so much to the structural features of the tree. That means that the inherent case is mainly connected to the semantic features of the elements and not to the tree's geometry, like the structural case.

In Modern Greek, the genitive case of the indirect object is inherent. The genitive case is assigned when a verb can stand with an indirect object, supporting the thematic role of the indirect receiver. Also, the indirect object can be positioned on the tree or right after the verb or after the direct object or in several other places. The fact that a verb supports the thematic role of the indirect receiver, and that the indirect object cab be positioned in several places, makes the genitive of the indirect object an inherent and not a structural case.

2.1.5.6 The control theory

The control theory defines the conditions that control the presence of the empty category PRO. During the construction of a sentence or a phrase tree, some terminal spaces, without any content, remain on the final tree. These empty elements have not been created by the shifting of the elements with the application of certain transformations, but they existed since the creation of the tree (Philippaki, 1985, 1987, 1989, 1990). The empty category PRO appears mostly in the English language.

The verbs of the main clauses that contain an infinitival supplement with the empty category PRO contain in their lexical representation a feature that shows whether they are *subject control* verbs or *object control* verb.

In the *subject control* verbs, the PRO reference point is the same with the reference point of the subject of the verb in the main clause therefore we have the same as the reference point of the subject of the verb in the main clause.
Such a verb in English is *promise* that is a control subject verb.

In the *object control* verbs, the PRO reference point is identical with the reference point of the object of the verb in the main clause therefore we have the same with the reference point of the object of the verb in the main clause.
Such a verb in English is *persuade* that is a control object verb.

Finally there are cases where PRO is not in a control position but in a supplementary sentence after the verbs that are not control verbs; then its reference point is free or arbitrary.

2.2 The unification based approach

2.2.1 The context free grammars

An example of a CFG can be the following:

- S-> NP VP
- VP -> V NP
- VP -> V
- NP -> D N
- NP -> PRON
- NP -> PROPER_NOUN

- D -> the | a | every
- N -> car | bicycle | boat | bus
- V -> drives | repairs | drive | repair | rides | ride
- RPON -> I | you | he | she | they | us | them
- PROPER_NOUN -> ANN | GEORGE | NICK

- Terminal symbols: the, a, every, car, bicycle, boat, bus, drives, repairs, drive, repair, rides, ride, I, you, he, she, they, us, them, ANN, GEORGE, NICK
- No terminal symbols: S, VP, NP, V, N, D

This grammar produces a set of grammatically and semantically correct and incorrect sentences.

Some examples of sentences that are produced and are not grammatically or semantically correct are the following:

- them repairs bicycle
- bicycle drives car
- Ann drive George
- the bus repair Nick

The context-free grammars have the following problems:

- The phrase structure is the only syntactic relationship.
- The terminal and non-terminal symbols are atomic with out any properties.
- The information that encoded in the grammar is based only on production rules and any attempt to encode semantic information requires additional mechanism.

The CFG mechanism must be stronger in order to be able to fulfill the linguistic requirements:

- Features structures
- Generalized phrase structures

- Unification grammars

2.2.2 The feature structures and the unification

The CFG can be extended by associated features structures with the terminal and no terminal symbols of a CFG. The features structures are known and as AVM (attribute value matrixes).

The words in the lexicon can be enhanced with additional information by using the features:

Two examples are the following:

$$\text{Word: Car} \quad \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix}$$

$$\text{Word: I} \quad \begin{bmatrix} \text{NUM: singular} \\ \text{PER: first} \end{bmatrix}$$

Except the simple atomic values of the features NUM and PERSON in the above examples, it is possible to have as value of the features other features structures. An example of a verb and its feature AGR is the following:

$$\text{Word: Runs} \quad \begin{bmatrix} \text{AGR:} & \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix} \end{bmatrix}$$

Also, it is possible to use variables with name e.g. X or with number e.g. [1] as in the following example (Fouskakis, 2005a):

$$\begin{bmatrix} \text{AGR:} \quad X & \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix} \end{bmatrix} \quad \begin{bmatrix} \text{AGR:} \quad [1] & \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix} \end{bmatrix}$$

The variables are used in order to determine that two elements of an AVM have the same values.

The general format of an AVM is the following:

$$A = [i_0]\begin{bmatrix} F1: & [i_1]\ A1 \\ & \\ Fn: & [i_n]\ An \end{bmatrix} \qquad \begin{array}{l} dom(A) \\ F_i \neq F_i \\ val(A,F_i) = A_i \end{array}$$

According to this, the previous example has:

- dom(a) = {ARG}
- $val(A,ARG) = \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix}$

Also, there is the notion of path π. At the same example the value of the path:

- val(A,<AGR, NUM>) = singular
- val(A,<AGR, PER>) = third
- but the val(A,<PER, AGR>) = undefined

Between two different features structures we can define the relation of subsumption.

If A and B are two AVMs the A subsumes B (A≤B):

- A is an atomic AVM and B is an atomic AVM with the same atom
- For every F that belongs in dom(A) then and F belongs in dom(B) and val(A,F) subsumes val(B,F).
- If two paths are re-entrant in A they are also re-entrant in B.

An example is:

$$\begin{bmatrix} \text{NUM: singular} \end{bmatrix} \leq \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix}$$

An operation between two features structures A and B is the unification. An example of unification:

$$A = \begin{bmatrix} \text{NUM: singular} \end{bmatrix} \qquad B = \begin{bmatrix} \text{PER: third} \end{bmatrix}$$

and after the unification we have the:

$$\begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix}$$

If variables exist in the A and B features structures:

$$A = \begin{bmatrix} \text{AGR:} & [1] \begin{bmatrix} \text{NUM: singular} \end{bmatrix} \end{bmatrix}$$

$$B = \begin{bmatrix} \text{AGR:} & [2] \begin{bmatrix} \text{PER: third} \end{bmatrix} \end{bmatrix}$$

$$\text{After the unification} \quad \begin{bmatrix} \text{AGR:} & [1][2] \begin{bmatrix} \text{NUM: singular} \\ \text{PER: third} \end{bmatrix} \end{bmatrix}$$

We can add features in the rules except the words of the lexicon. An example is one of the rules that are described above.

$$\underset{\text{NP}}{\begin{bmatrix} \text{NUM: X} \end{bmatrix}} \text{------>} \underset{\text{D}}{\begin{bmatrix} \text{NUM: X} \end{bmatrix}} \underset{\text{N}}{\begin{bmatrix} \text{NUM: X} \end{bmatrix}}$$

In this example the scope of the variable X is inside the rule and means that the noun phrase (NP), determiner (D) and noun (N) have the same number. Also, if we want to control the case we can add a second feature the CASE and the result is the following:

$$\underset{\text{NP}}{\begin{bmatrix} \text{NUM: X} \\ \text{CASE: Y} \end{bmatrix}} \text{------->} \underset{\text{D}}{\begin{bmatrix} \text{NUM: X} \end{bmatrix}} \underset{\text{N}}{\begin{bmatrix} \text{NUM: X} \\ \text{CASE: Y} \end{bmatrix}}$$

The rule for the verb phrase (VP) depends from the type of the verb. There are transitive and no-transitive verbs that they do not have a noun phrase as complement. In this case we have two rules with their corresponding features structures.

$$\underset{\text{VP}}{\begin{bmatrix} \text{NUM: X} \end{bmatrix}} \text{------->} \underset{\text{V}}{\begin{bmatrix} \text{NUM: X} \\ \text{SUBCAT: intransitive} \end{bmatrix}}$$

$$\underset{\text{VP}}{\begin{bmatrix} \text{NUM: X} \end{bmatrix}} \text{------->} \underset{\text{V}}{\begin{bmatrix} \text{NUM: X} \\ \text{SUBCAT: transitive} \end{bmatrix}} \underset{\text{NP}}{\begin{bmatrix} \text{NUM: Y} \end{bmatrix}}$$

In the above examples it was used the CFG rules associated by the features structures. It is possible to include the no-terminals as values of a CATEGORY feature. An example is the following:

$$\overset{\text{NP}}{\begin{bmatrix}\text{NUM: X}\end{bmatrix}} \text{ -------> } \overset{\text{D}}{\begin{bmatrix}\text{NUM: X}\end{bmatrix}} \overset{\text{N}}{\begin{bmatrix}\text{NUM: X}\end{bmatrix}}$$

which can be as:

$$\begin{bmatrix}\text{CAP: NP}\\ \\ \text{NUM: X}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT: D}\\ \\ \text{NUM: X}\end{bmatrix}\begin{bmatrix}\text{CAT:N}\\ \\ \text{NUM: X}\end{bmatrix}$$

In order to have complete sub categorization information we can enter in the lexicon the complete list of complements and the subject. It is possible to add additional features like the CASE that is determined for the subject of the verb take in the following example:

$$\begin{bmatrix}\text{CAT: verb} & \\ \text{SUBCATEGORIZATION: < [CAT:NP], [CAT:NP] >} & \\ \text{SUBJECT:} \begin{bmatrix}\text{CAT: noun phrase}\\ \text{CASE: nominative}\end{bmatrix} & \\ \text{NUM: singular} & \end{bmatrix}$$

According to the above if we want to express the initial rule of the CFG:

S - > NP VP

with the use of features structures it will be as:

$$\begin{bmatrix}\text{CAT: S}\end{bmatrix} \rightarrow [1]\begin{bmatrix}\text{CAT: NP}\\ \text{CASE:nominative}\\ \text{NUM: [2]}\end{bmatrix}\begin{bmatrix}\text{CAT: VP}\\ \text{NUM: [2]}\\ \text{SUBJECT:[1]}\end{bmatrix}$$

2.2.3 The HPSG grammar

All the above examples and different cases describe the main notions and mechanisms of the unification based grammars. The different grammar formalisms (FUG, PART-II, LFG, CUG, FTAG) use the features structures that have been described in the previous section but the current formalism that is used very much with big research effort is the HPSG (Tatar 2001, 2003). None from the above does not be designed to be used on the Chomsky's x-bar scheme.
The HPSG is a declarative approach, it provides a model of what linguistic enitics are possible. It is seen as a later development of GPSGs (Gazdar, 1985) and makes more specific claims about universals and variation than the more conservative GPSG. It was designed as a synthetic grammar model. It combines the advantages of different grammatical theories Generalized Phrase Structure Grammars (GPSG), Categorical Unification Grammars (CUG) and Lexical Function Grammars (LFG). The TAGs are defined as a tree rewriting system. The TAG grammars use elementary trees which can be of any depth, in contrast to rewrite rules which have only two levels (left and right part of rule) and these trees are separated in initials and auxiliaries. A auxiliary tree has a noterminal as root node and exact one noterminal as foot node that must be the same noterminal. This is presented only in TAG grammars. Also, The HPSG has the dominance paradigm (expessed by the head feature principle: the HEAD value of the a headed phrase is identified with that of its head-daughter) that it was presented in the government and binding theory. This approach is not a transformational approach, like chomsky's theories, but it is based on the main mechanisms of the unifications grammars and supports features structures (AVM). It does not support rewriting rules in the general sence and there is no notion of deriving one structure from the another. It supports the sign structure with very detailed information from the lexicon. The sign has a format like the feature structure of the verb that is described below. It is said to be surface oriented because it provides a direct characterization of the surface order of elements in a sentence. The information about the specifiers and the complements is present in the argument structure attribute (ARG-ST). The value of this argument is an ordered list of the arguments that are required by the sign. The order is very important for every possible phrase in every language. The variables have a very important role in these grammars. They declare that two elements in a sign have the same values. The HPSG is based in all the mechanisms of the unification grammars as they have described in the above section.

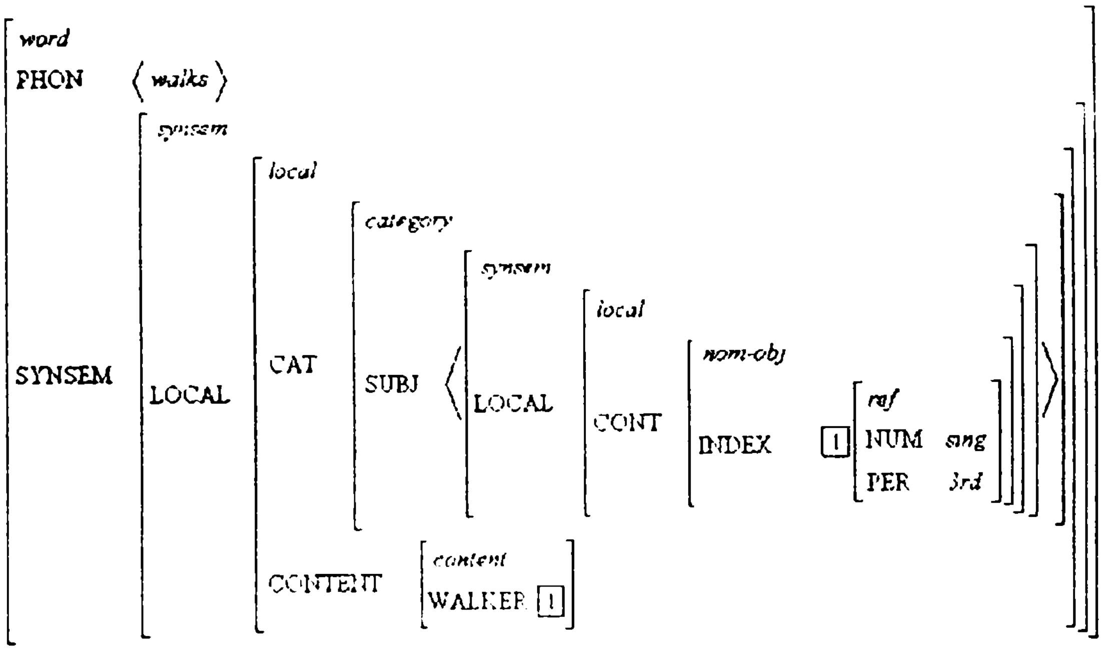
The HPSG puts a lot of emphasis on the precise mathematical modelling of linguistic entities. Because of the focus on precision, a lot of linguistic computer implementations are based in HPSG. It is a declarative approach and the combination of the declared information is depending from the corresponding software system (Copestake, 2002) that permits the declaration of the HPSGs. The number of these signs is increased enormously in order of every different case in a language to be described. Also, there are problems in translation systems because the sign of the source and destination language are not possible to be determined directly and it neseccary for another semantic mechanism to be used (Copestake, 2002).

The sign has a collection of phonological, syntactic and semantic constraints that are included in hierarchical features structures (attribute-value-matrixes AVM).

Sings have the attributes like:

- word or phrase status
- phonology (PHON)
- syntax/semantics (SYNSEM)

The structure of the last attribute can contain other attributes (AVM) that may contain other attributes (AVM) in any depth and structure. An example is the verb *walks* that have the following general format.



In general terms a HPSG has the following parts:

- A sign that describes specific attributes and types. A grammar that is complex enough, is characterized by a impressively complex sign.
- A inheritance hierarchy of types and a agreement specification about their attributes.
- A lexicon and a small list of rewriting rules that named schemes.
- A list of some general principles.

2.2.4 The PATR grammar

The PART (Tatar 2001) has its initials from the words parse and translate. It is one of the oldest unification based approach (after the FUG) and it supports grammar (CFG) rules that consists of a mother category and zero or more daughter categories with a list of feature equations. A category is a set of feature-value pairs. A feature is an atom and a value can be an atom, a variable or a category. The feature equations on a rule set constraints on their values. The lexical items are viewed as rules without daughter categories.

A rule of this type of grammars can be as:

XS → XNP XVP
<XS cat>= s <XNP cat>=np <XVP cat>=vp
<XS head>=<XVP head>
<XS head subject> = <XNP head>

The PATR formalism is reasonably expressive. But, it doesn't have some desirable properties, like disjunction and negation of a set or list of value features. It declarative, monotonic and reversible. Also, it is turing equivalent and if a PATR contains only atom-valued features it is as CFG of $Q(n^3)$.

As conclusion, the main characteristics of the PATR grammars are:

- CFG
- unification
- paths in equations

2.2.5 The FUG grammar

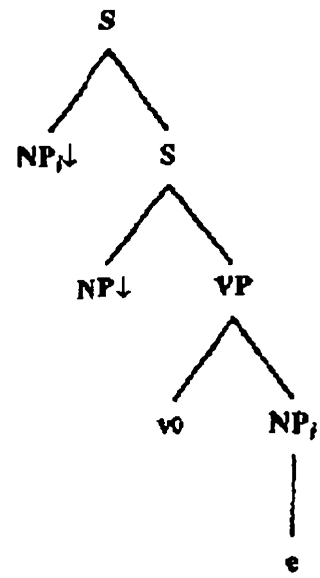
The FUG has the initials of the words functional unification grammars. It was presented before the PATR grammars and in many ways is similar. The context free part in the PATR grammars is replaced by two features the *cset* and the *pattern*. They declare which items are the daughters of a category and at which order they appear. Multiple CFG 'rules' about one category are declared by disjunction. Finally, there is the feature value *any* that declares the requirement that it is obligatory of a feature to exist. This possibility adds the non-monotonic in the unification based approach.

2.2.6 The TAG grammar

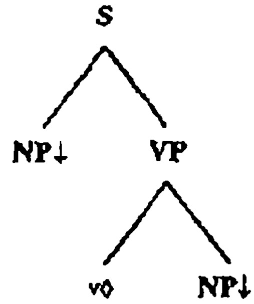
The Tree Adjoining Grammar (TAG) is defined as a tree rewriting system (Joshi, 1975). In the definition given traditionally, TAG is defined by a finite set of trees and an operation called adjoining to compose trees. It represents a extension of the basic rule rewriting scheme that underlying other modern grammatical formalisms. Unlike these string rewriting formalisms that write recursion into the rules that generate the phrase structure, a TAG captures recursion and dependencies (agreement, subcategorization filler-gap connections) into a finite set of elementary trees. The TAGs have provided a theoretical framework for linguistic description and natural language processing that has been shown to be superior to simply using rules of a context free grammar (CFG) due in large part to the extended context or "domain of locality" that TAG provides(Babko-Malaya, 2004).

There are three kinds of elementary trees: **initial trees, auxiliary** and **lexical** trees. The initial trees were defined to correspond to minimal sentential structures. Therefore, the root of an initial tree was required to be labeled by the symbol S. The following scheme has two elementary trees of this kind.

α_2 :

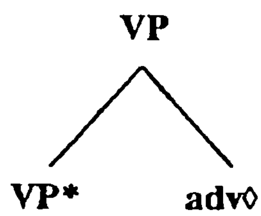


α_1 :

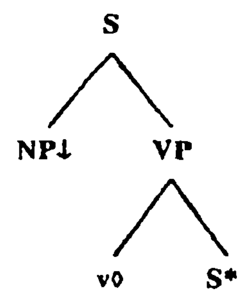


The second kind of elementary trees is the auxiliary trees. They have as root node any nonterminal symbol. The lowest nodes have only terminal symbols except for exact one nonterminal (foot node) that is the same as the nonterminal of the root node. The following are two examples of auxiliary trees:

β_1 :



β_2 :

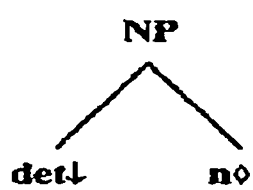


The pairs of nonterminals are (VP, VP*) in the first β_1 and (S, S*) in the second β_2 tree.

Later, a new category of trees have be introduced, named lexical trees (Schabes, 1988). They associated with particular words in the lexicon. They have as root node any nonterminal. In a lexicalized TAG, frontier nodes labeled by nonterminals (like the NP nodes in the above examples), with the exception of foot nodes, are marked for substitution (specified by ↓) and are not elaborated any more. Their elaboration is done by the lexical trees.

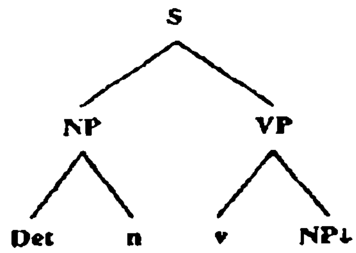
An example of a lexical tree is the following:

α_3 :



An example of substitution is the tree γ_3 as result of α_1 and α_3.

γ_3 :



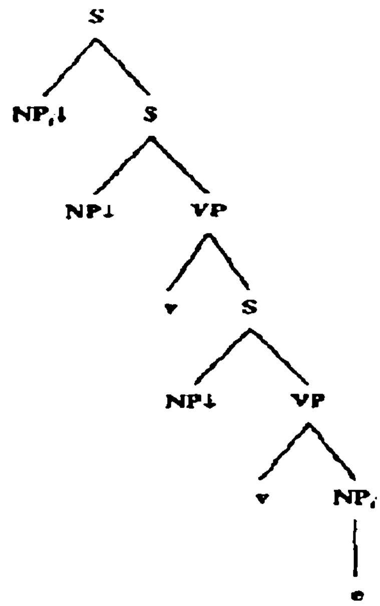
In deriving tree structures top-down from the grammar the usual operation of substitutution of a mother by its daughters has been augmented by the *adjoining* operation about composing trees. An auxiliary tree, whose root and its foot node are labeled X, can be adjoined at a node that is also labeled X. Adjoining may be described as follows: the subtree below the node of adjunction is excised; the auxiliary tree is inserted in its place; and the excised subtree is substituted at the foot node of the inserted auxiliary tree.

Two examples of trees adjunction:

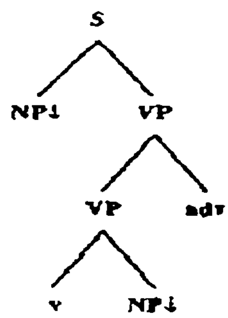
- γ_1 results from the α_1 and β_1
- γ_2 results from the α_2 and β_2

At the second example, it is observed that the co-indexed nodes (NP_i) remain and after adjunction.

γ_2 :



γ_1 :

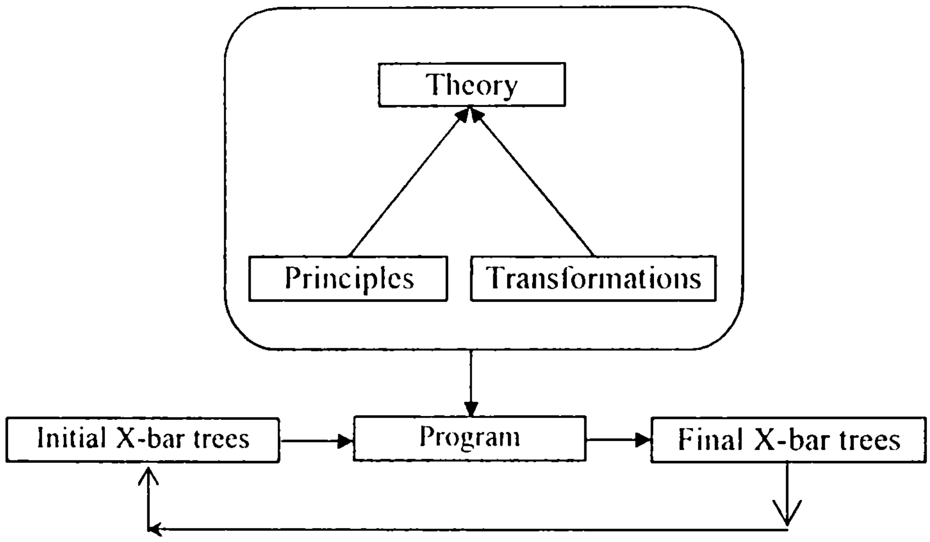


The TAG has been embedded in a feature structure based unification system and the resulting formalism is the FTAG (Vijay-Shanker, 1988). At each node has ossociated both a top and a bottom feature structure. If a djunction operation is taking place at a node the top feature structure unifies with that of the root node of the auxiliary tree and the bottom feature structure unifies with that of the foot node. If there is not adjunction at a node then its top and bottom feature structures must be unified. It functions equivantly with the PATR. The TAGs have extended domain of locality and provide greater expressive power. The formalism is fully declarative, reversible and monotonic. Different variations have been published that permit more flexible manipulation of long distance dependences and word order variations (Millett, 2004).

3. The personal contribution

3.1 Description of the linguistic system's structure of the presented methodology

The linguistic knowledge of this methodology has a structure which is presented in the following figure (Fouskakis, 2004c, 2005a). It is artificial language for linguistic rules, different that the classical approaches of grammar declaration and a parser that implements the corresponding grammar.



This structure represents the system of the linguistic knowledge.

Let define:

- LS: the system of the linguistic knowledge
- PR: the set of rules in the Principles
- TR: the set of rules in the Transformations
- GR: the set of rules in the Theory
- SR: the linguistic program
    - SR is subset of the concatenation of the sets GR, PR and TR
- IT: the set of initial X-bar trees
- OT: the set of final X-bar trees

$$LS=(PR,TR,GR,SR,IT,OT)$$

- <u>The initial X-bar trees</u>

It contains trees that derive from the X-bar scheme. These trees will be used by the methodology, in order to apply on them the rules. Their format is given in the corresponding section below and it is according to the X-bar theory.

- <u>Principles</u>

It contains all the principles that have defined so far. The principles check an X-bar structure if it accomplishes certain structural requirements as a whole or at its parts. Also, they can check if nodes, features of nodes, anaphors, even terminals are according to certain linguistic requirements.

- <u>Transformations</u>

It contains all the transformations that have defined so far. The transformations additionally, transform the X-bar structures and produce one or more new X-bar trees with different structure, nodes, features of nodes, anaphors or even terminals.

- <u>The Linguistic Theory</u>

It contains rules that express the linguistic theory that one wishes to develop. These rules are expressed as sequences of principles and transformations. We can also have a conditional application of the rules by using expressions if-then-else and change the X-bar trees that are used by the next rules. The abilities that these rules have will be described in detail in the next sections.

- <u>The Linguistic Program</u>

It is the part of the linguistic system which declares the rules of the theory, principles, transformations sets that are applied on the initial X-bar structure and their order.

- <u>The final X-bar trees</u>

It contains the generated X-bar structures according to the linguistic program.

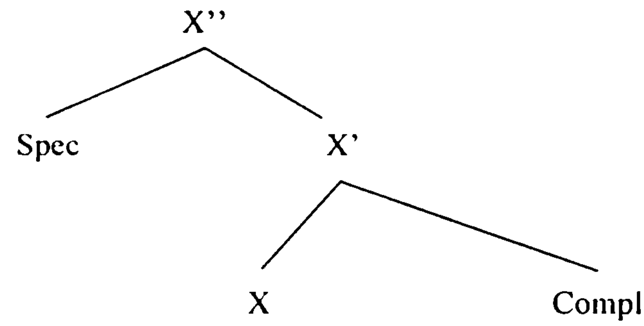
3.1.1 The Structures

The structures processed by the methodology are trees that derive from the basic scheme of the x-bar standard theory. The choosing of this binary scheme (Fouskakis, 2004b, 2005b) is based on its computational simplicity by permiting the declaration of more general rules on the produced trees.

These trees are described by one or more of the following rules:

X'' → Spec X''	X'' → Spec X'
X' → X' Complement	X' → X Complement
Spec → X''	Spec → X
Complement → X''	X → Terminal

As it is noticed from the above rules the general x-bar scheme is improved in the presented work with the possibility of repetition of the node X'' which facilitates in free order languages and in the case that we have many specifiers with adverbs, adjectives and quantifiers. Usually, the specifier is about the articles or the quantifiers of the nouns and the complement is for their complementary phrases or their adjectives. Similary, at verb phrases, the complement is about adverbs and complementary phrases (objects). The exact representation depends from the application and the language.



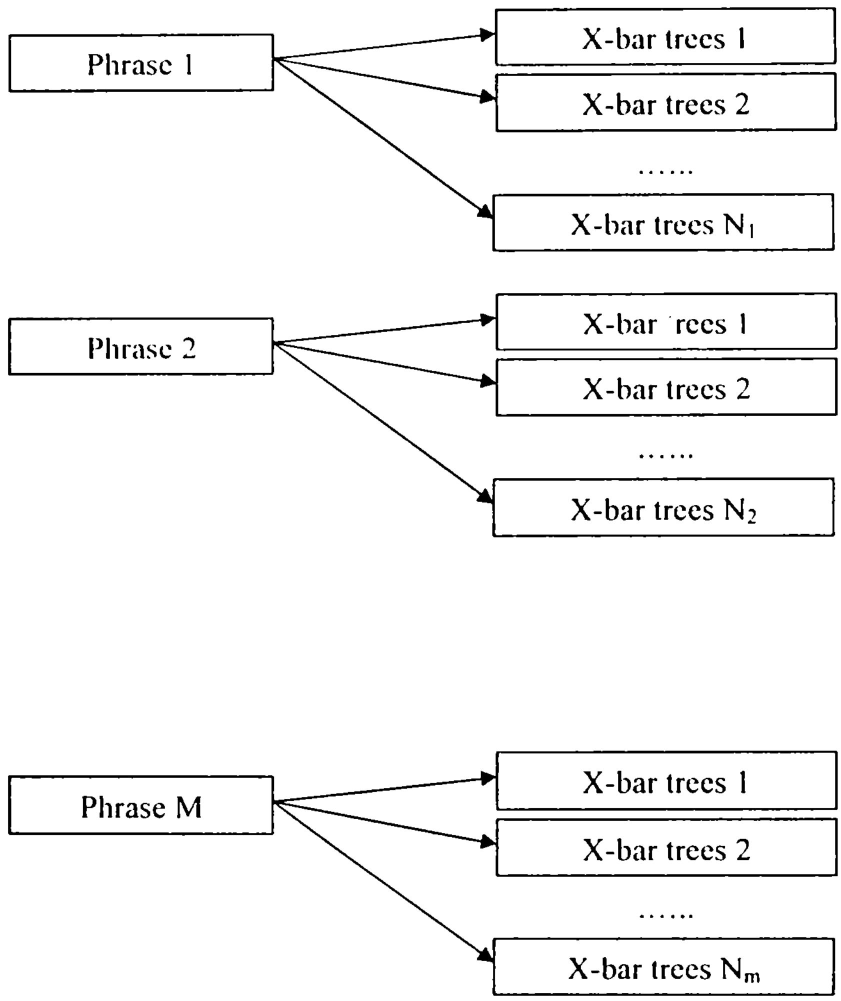
The basic schema of the X-bar theory

In the present methodology the X-bar structures are expressed with the use of parentheses as follows:

(X'' (*Spec*) (X' (X) Y''))
(X'' (*Spec*) (X' (X' (...) Y''))
(X'' (*Spec*) (X'' (*Spec*)...

In these structures the Y'' is the complement that is an X'' category tree, an x-bar tree. The specifier (spec) can be either an X'' category tree or an X category node with a terminal connected to it (see EBNF form of the X-bar trees below).

Every phrase, sentence or utterance can be represented by more than one X-bar trees. This is the reason that their trees can be represented as an table where every position has a list with the possible different X-bar trees of the corresponding phrase (Fouskakis, 2004c), sentence or utterance. The next scheme shows this broblem.



Next we will describe in detail the abilities of the methodology regarding the X-bar structures, that is, the trees that derive from the above basic scheme. These trees correspond natural language sentences or phrases. Also, due to the generality of the X-bar trees, it can be used in other branches of the linguistic research, in which there is an attempt to use the X-bar scheme, as in morphology, in order to describe the structure of a word.

The X-bar trees have nodes, terminal elements, anaphors and nodes features.

- <u>The nodes of the trees</u>

In order to describe a node on a tree we use its name that is a symbol followed by the node's category. The node name is described as a prolog atom. It is a sequence of letters and numbers, that its first character is a lower-case letter or, if this letter is capital we should use quotes. For each node category we have the following cases:

1. we enter the X'' category node as x **barii**
2. we enter the X' category node as x **bari**
3. we enter the X category node as x **bar**

Every node, apart from its category and its name, can also have features that are entered with the operator **features**. The features of the nodes give grammatical, syntactical, semantic and pragmatic information about the node and the subtree that has this node as a top. The features of the node are enclosed in [and] and separated by commas. The sequence of the node's features is irrelevant.
The features of the node are notated as follows:

1. *+name of the feature*
2. *name of the feature*
3. *name of the feature*
4. *name of the featureX = name of the featureY*
5. *[name of the feature1, ..., name of the featureN]= name of the featureX*

Their semantics depend from our interpretation. In the forth and fifth case the order of the features is important and these cases are not supported by the X-bar theory of Chomsky. They permit better well expressed additional descriptions.
For example, we can say that a node may have the following features:

- +animate
- -inanimate
- person
- [+live_being,+thing]=complements
- phrase_time=t3, focus=v1

Therefore, the nodes of an X'', X', X tree are described in the following general way:

node *name of the node type of the node*: **features** *characteristics of the node*

For example, the node:
having **A''** as a name and the characteristics **singular, nominative**
is expressed as follows:
node 'A' **barii** : **features** [singular, nominative]

- <u>The terminal elements</u>

Apart from the nodes, a tree has also the terminal elements that can be connected to other terminal elements of the tree or to whole subtree with anaphors. In order to denote that the specific element of the tree is terminal we use the operator **terminal**, while for the anaphors we use the operator **anaphor**.

Therefore, if we have the terminal element "the" and the anaphor "i1" that connects the terminal element to another element of the tree, we can describe this as follows:

terminal the : **anaphor** i1

If the terminal element of the tree didn't have an anaphor, then we have:

terminal the

Therefore the terminal element of a tree is generally described as follows:

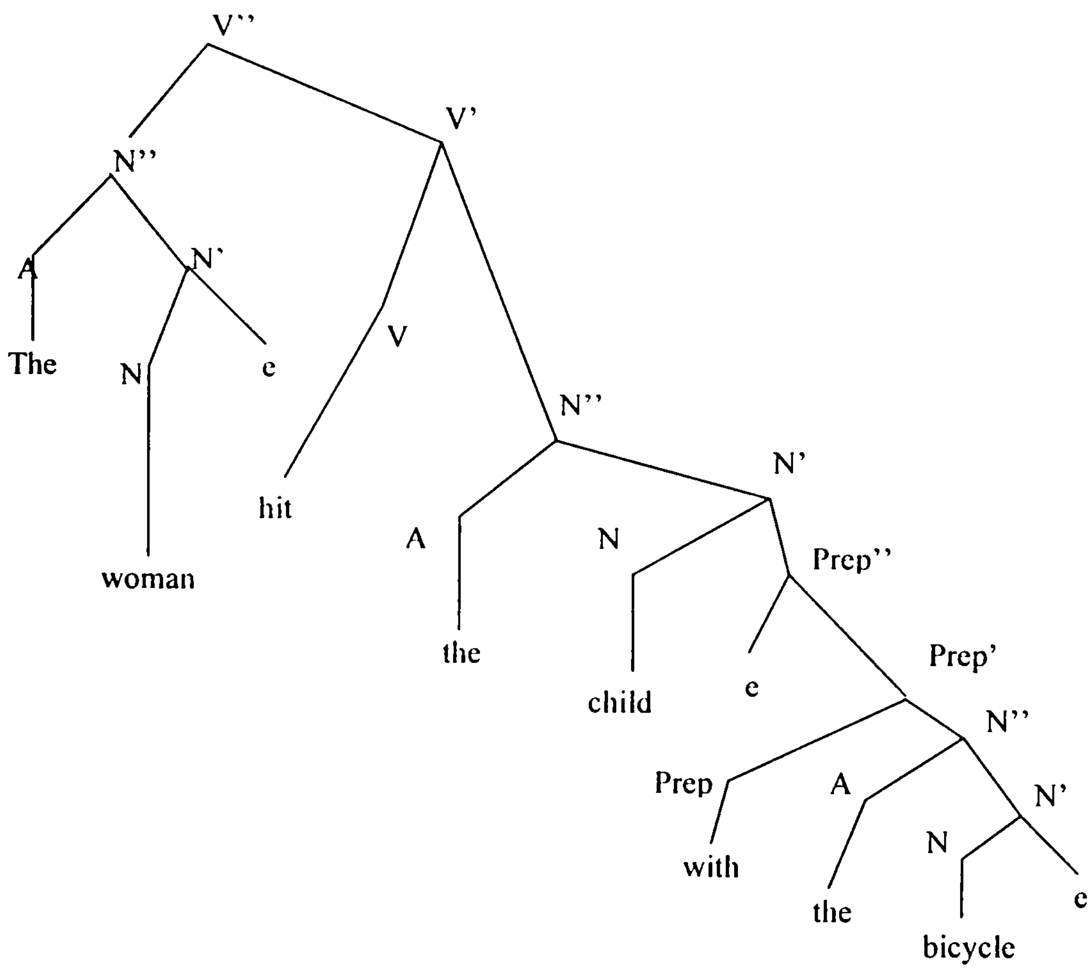
terminal *terminal element* : **anaphor** *name of the anaphor*

Next we shall present the tree of the sentence below that derives from the x-bar scheme:

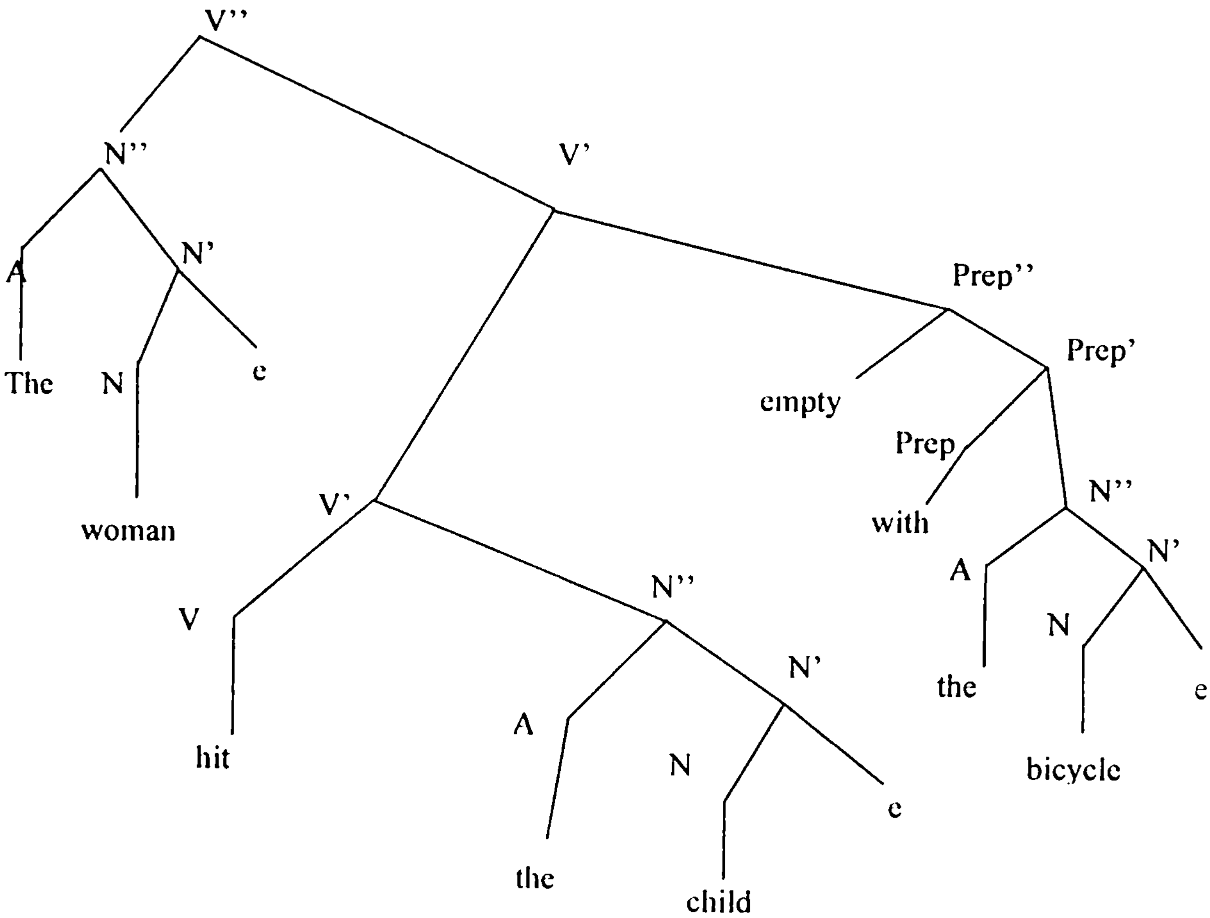
The woman hit the child with the bicycle

The above sentence can be represented by one more than one trees, depending on the sentence's meaning.

A) If in the sentence above the prepositional phrase "with the bicycle" is a complement to the nominal phrase "the child", it specifies the child we are talking about, meaning the child with the bicycle and not another child, then the tree for the above sentence is the following:



B) If in the sentence above the prepositional phrase "with the bicycle" is not a complement to the nominal phrase "the child" but a complement of the verb "hit" and specifies the instrument with which the woman hit the child, then the tree for the above sentence is the following:



These two different trees denote two different interpretations of the sentence

The woman hit the child with the bicycle

Therefore we can describe these two trees in this methodology as:

a) The first tree is:

(node v **barii,(node** n **barii,**
(node a **bar,terminal** the),

```
                    (node n bari, (node n bar, terminal woman),empty)
                  ),
                  (node v bari,
                      (node v bar, terminal hit),
                      (node n barii,
                          (node a bar, terminal the),
                          (node n bari,
                              (node n bar, terminal child),
                              (node prep barii,
                                  empty,
                                  (node prep bari,
                                      (node prep bar, terminal with),
                                      (node n barii,
                                          (node a bar, terminal the),
                                          (node n bari,
                                              (node n bar, terminal bicycle),
                                              empty)
                                      )
                                  )
                              )
                          )
                      )
                  )
)
```

b) The second tree is:

```
(node v barii,
    (node n barii,
        (node a bar, terminal the),
        (node n bari, (node n bar, terminal woman),empty)
    ),
    (node v bari,
        (node v bari,
            (node v bar, terminal hit),
            (node n barii,
                (node a bar,terminal the),
                (node n bari,
                    (node n bar, terminal child),
                    empty)
            )
        ),
        (node prep barii,
            empty,
            (node prep bari,
                (node prep bar, terminal with),
                (node n barii,
                    (node a bar, terminal the),
```

```
(node n bari,
    (node n bar, terminal bicycle),
    empty)
    )
   )
  )
 )
)
```

We observe that the names of the tree nodes are in lower case letters and we also use the word **empty**. As we observe in the respective trees, **empty** corresponds to an empty subtree, meaning that the respective branch of the tree is empty.

On the above trees the nominal phrase "The woman" is described in the following tree:

```
(node n barii,
    (node a bar, terminal the),
    (node n bari, (node n bar, terminal woman),empty)
)
```

We observe that this is a phrase of the N'' category that has as a left subtree the article and a right subtree of the N' category, with the noun "woman" as a left subtree and an empty tree as a right subtree, that is expressed with the code word **empty**.

Also, the verbal phrase "hit the child with the bicycle" is expressed in two trees, depending on the meaning of the sentence.

a) the first tree is :

```
(node v bari,
    (node v bar, terminal hit),
    (node n barii,
        (node a bar, terminal the),
        (node n bari,
            (node n bar, terminal child),
            (node prep barii,
                empty,
                (node prep bari,
                    (node prep bar, terminal with),
                    (node n barii,
                        (node a bar, terminal the),
                        (node n bari,
                            (node n bar, terminal bicycle),
                            empty)
                    )
```

```
                    )
                )
            )
        )
    )
```

This tree is of the V' category and has as a left subtree the verb "hit" and as a right subtree the phrase "the child with the bicycle". This phrase is a nominal phrase with a complement and is expressed with a tree of the N'' category. This tree has as a left subtree the article "the" and as a right subtree a tree of the N' category, that has as a left subtree the noun "the child" and as a right subtree the prepositional phrase that is expressed with a right subtree of the Prep'' category. This tree has an empty left subtree and its right subtree is of the Prep' category with the preposition "with" as a left subtree and the nominal phrase "the bicycle" as a right subtree. This nominal phrase is also analyzed in a nominal phrase of the N'' category.

b) the second tree is:

```
(node v bari,
    (node v bari,
        (node v bar, terminal hit),
        (node n barii,
            (node a bar,terminal the),
            (node n bari,
                (node n bar, terminal child),
                empty)
        )
    ),
    (node prep barii,
        empty,
        (node prep bari,
            (node prep bar, terminal with),
            (node n barii,
                (node a bar, terminal the),
                (node n bari,
                    (node n bar, terminal bicycle),
                    empty)
            )
        )
    )
)
```

This tree is of the V' category and has as a left subtree a V' category subtree and as a right subtree, a subtree of the Prep'' category that corresponds to the prepositional phrase "with the bicycle", that states the instrument with which the woman hit the child. The left subtree is of the V' category and corresponds to the phrase "hit the child". This subtree has a left subtree of the V category and the verb "hit" and a right subtree of the N'' category for the nominal phrase "the child".

The phrase "hit the child" is described by the following tree:

```
(node v bari,
    (node v bar, terminal hit),
    (node n barii,
        (node a bar,terminal the),
        (node n bari,
            (node n bar, terminal child),
            empty)
    )
)
```

The prepositional phrase "with the bicycle" describes the instrument with the following tree:

```
(node prep barii,
    empty,
    (node prep bari,
        (node prep bar, terminal with),
        (node n barii,
            (node a bar, terminal the),
            (node n bari,
                (node n bar, terminal bicycle),
                empty)
        )
    )
)
```

Apart from of the operators for the nodes and the terminal elements, there is also the operator that denotes the anaphors of a tree. The anaphors are connections between a tree's elements that have a relation between them, for example:

a) the pronoun and the word or phrase to which it refers
b) the reflexive pronoun and the element to which it refers

In order to state this anaphor we use the operator **anaphor** and the name of the anaphor. The name of the anaphor must be an atom of the prolog, that is, a sequence of letters and numbers with the first letter in lower-case or, if it is a capital letter, it should be enclosed in quotes.
The following are examples of anaphors:
a) **anaphor** i1
b) **anaphor** 'I1'
c) **anaphor** 'anaphor_1'

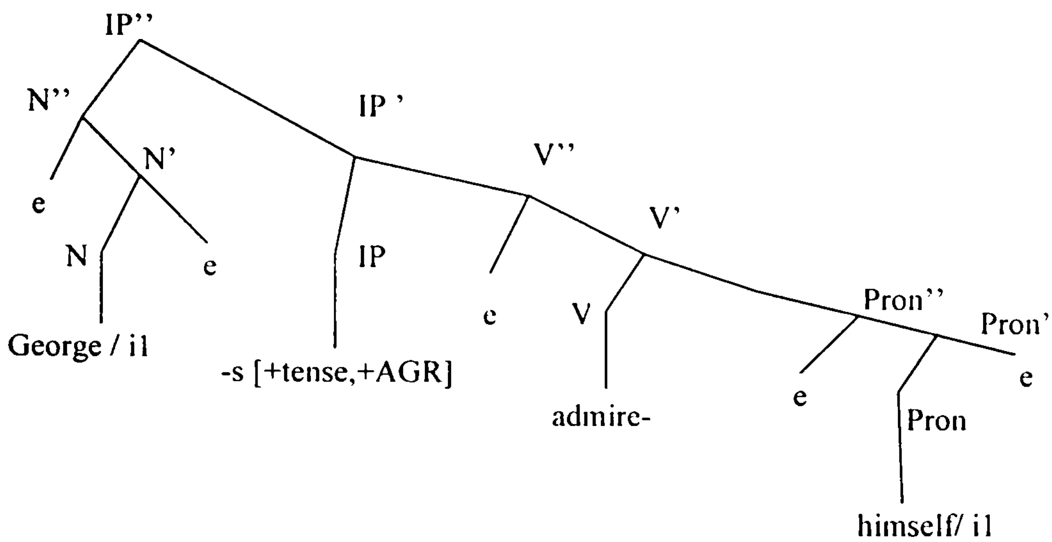
d) **anaphor** 'Anaphor_1'

With the anaphor we can connect elements. These elements can belong to the following categories:

a) terminal element to terminal element
b) terminal element to subtree that can belong to the categories X, X', X''
c) subtree to subtree, that can belong to the categories X, X', X''

Next we shall present an example of a sentence with a binding between the reflexive pronoun and the noun to which it refers. This noun should always be within the same sentence

George admires himself



In this methodology the above tree is expressed as follows:

(node 'IP' **barii,**
(node 'N' **barii,**
empty,
(node 'N' **bari,**
(node 'N' **bar, terminal** 'George':**anaphor** i1),
empty
)
),
(node 'IP' **bari,**
(node 'IP' **bar** : **features** [+tenses,+AGR],**terminal** '-s'),
(node 'V' **barii,**

```
            empty,
            (node 'V' bari,
                (node 'V' bar, terminal 'admire-'),
                (node 'Pron' barii,
                  empty,
                  (node 'Pron' bari,
                     (node 'Pron' bar, terminal 'himself':anaphor i1),
                     empty
                  )
                )
            )
        )
    )
)
```

The above tree is a tree of the IP'' category having as a left subtree the one that corresponds to the nominal phrase "George".

```
(node 'N' barii,
    empty,
    (node 'N' bari,
        (node 'N' bar, terminal 'George':anaphor i1),
        empty
    )
)
```

The right subtree is of the IP' category and has a left subtree the head of the tree with the IP'' top. The head of this tree has the features +tense and +AGR and as a terminal element the ending.

```
(node 'IP' bar:features [+tenses,+AGR], terminal '-s')
```

Also, the IP' has as a right subtree the verbal phrase "admires himself".

```
(node 'IP' bari,
    (node 'IP' bar:features [+tenses,+AGR], terminal '-s'),
    (node 'V' barii,
        empty,
        (node 'V' bari,
            (node 'V' bar, terminal 'admire-'),
            (node 'Pron' barii,
              empty,
              (node 'Pron' bari,
                 (node 'Pron' bar, terminal 'himself':anaphor i1),
                 empty
              )
            )
```

)
)
)

The tree of this phrase has an empty left subtree and a V' category right subtree that its left subtree corresponds to the stem of the verb "admire" and its right subtree is a subtree of the Pron'' category that corresponds to the pronoun "himself".

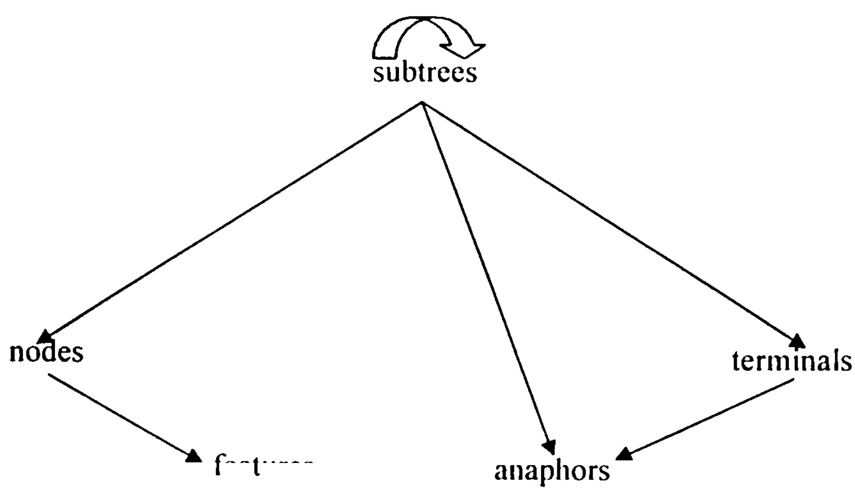
Examining the subtrees that correspond to the nominal phrase "George" and to the pronoun "himself", we observe that the terminal element "George" and the terminal element "himself" are connected to each other with the anaphor "i1".

Finally, regarding the subtrees that the methodology manipulates, apart from the empty subtree which is expressed with the word **empty**, we can also describe subtrees or terminal elements that were moved in the tree structure, leaving in the tree an empty space in the place where this element was. In the place that's left empty we enter the **t** from the word **trace** that states the trace that this element leaves after it was moved. The **trace** can be bound with the element that occupied that place and has been moved to another place on the tree.

For example, if we moved the word "George", then the trace and the word would be connected as follows:

'George':**anaphor** i1
t: **anaphor** i1

Finnlay, the following schema presents the different possibilities of using between the different elements of a x-bar structure of this methodology as has been described above. These elements are of type subtree, node, terminal, features of node and anaphor.



The element at the end of the arrow is used by the other one in order to be structured.

3.1.1.1 The EBNF of the X-bar structures

The EBNF of the structures that the methodology can manipulate is the following:

structure = tree-χ''.

tree-χ'' = "(" node-χ'' "," specifier "," (tree-χ'' | tree-χ') ")" [":" anaphors].

tree-χ' = "(" node-χ' "," (tree-χ' | tree-χ) "," tree-χ'' ")" [":" anaphors].

tree-χ = "(" node-χ "," terminal ")" [":" anaphors].

tree-χ'' = "empty".

tree-χ'' = "t".

tree-χ' = "t".

tree-χ = "t".

specifier = tree-χ'' | tree-χ.

node-χ'' = "node" node-name "barii" [":" node-features].

node-χ' = "node" node-name "bari" [":" node-features].

node-χ = "node" node-name "bar" [":" node-features].

node-features = "features" "[" feature { "," feature } "]".

feature = "+" feature-name | "-" feature-name | feature-name
| feature-name '=' feature-name
| "[" feature { "," feature } "]=" feature-name.

terminal = "terminal" terminal-element [":" anaphors].

anaphors = "anaphor" anaphor-name { ":" "anaphor" anaphor-name }.

node-name = name.

feature-name = name.

anaphor-name = name.

terminal-element = name | "t".

name = lower-letter { lower-letter | capital-letter | number | "_" }.

name = """ capital-letter { lower-letter | capital-letter | number | "_" }""".

lower-letter = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v |
w | x | y | z.

capital-letter = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S |
T | U | V | W | X | Y | Z.

number = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3.1.2 The Principles and Transformations

The principles and the transformations are rules that we define according to the presented methodology(Fouskakis, 2004c). These rules are stated to be applied on the x-bar trees that were described in the previous chapter. The principles are used to control the correctness of an X-bar tree according to the requirements that we state.

The transformations are stated in the same way and have the same abilities with the principles, but they can also change the structure and the elements of the tree to which they are applied, leading on one or more trees.

The principles enable us to study on the tree they are applied on the content of its nodes, the existence of a subtree, the relation between two or more subtrees of the X-bar tree, the bindings, etc.

The transformations provide us with the same abilities and furthermore, we can modify the structure of the X-bar tree and produce one or more new trees that can have a totally different structure from the structure of the original tree. We can also change the content of the nodes, by changing for example the features of the node or we can change the terminal elements by entering new words.

The principles and transformations are the main part of the methodology and are declared in the presented linguistic knowledge system. We can enter in the system a big set of such rules and use only these rules that we wish to apply each time on the x-bar trees.

With these rules we express the main linguistic knowledge that is of our interest and thus we can process the natural language trees accordingly. The complexity and the number of the rules depend on our requirements.

Both the principles and the transformations are stated using the same general pattern.

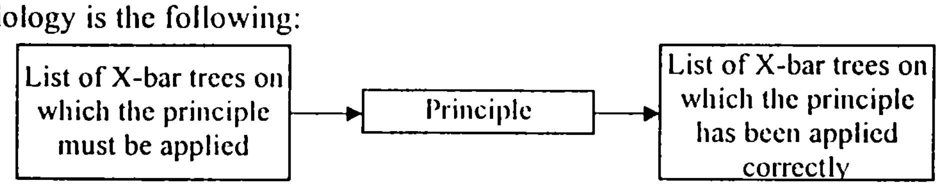
The principles in the methodology have the following pattern:

principle name of principle
variables denotation of the variables that will be used in the next fields
structureDescription description of the X-bar subtreee on which the rule will be applied.
structureCommands the different elements checks, the variables values changes, the new declarations of variables other possible commands are used according to the application

The transformations in the methodology have the following pattern:

transformation name of transformation
variables denotation of the variables that will be used in the next fields
structureDescription description of the X-bar subtreee on which the transformational rule will be applied.
structureCommands the different elements checks, the variables values changes, the new declarations of variables, the transformations and other possible commands are used according to the application

The schema of the general function of the principles of the presented methodology is the following:



The schema of the general function of the transformations of the presented methodology is the following:

3.1.2.1 The EBNF of the principles and transformations of the methodology

The EBNF form of principles and transformations is the following:

principle = "principle" principle-name
"variables" variables-declaration
"structureDescription" structureDescription-structure
"stucturecommands" stucturecommands-principle.

transformation = "transformation" transformation-name
"variables" variables-declaration
"structureDescription" structureDescription-structure
"stucturecommands"stucturecommands-transformation.

principle-name= name.

transformation-name = name.

Regarding the EBNF form of these rules, the name was declared in the previous EBNF denotation that is for the X-bar structures of the methodology.

In the following chapters we will describe the abilities provided in each one of the above fields (variables-declaration, structureDescription-structure, structureCommands-principle, structureCommands-transformation) of principles and transformations.

3.1.3 The Linguistic theory

We can describe a set of rules by using principles and transformations that we have defined in the linguistic system. The set of all the rules that we declare constitute our theory. This theory is the grammar we define.

The general pattern of describing these rules, is the following:

grammar *name of grammar*
the main part of the grammar

The name of the grammar is an atom of prolog, meaning a sequence of letters and numbers. If the first letter is capital, the whole name should be enclosed in quotes, while if it is a lower case letter, there is no need to use quotes.

For example:

- grammar_1
- 'Grammar_1'
- 'GRAMMAR_1'

All the above are acceptable grammar names that we can be used in the methodology.

In the main part of the grammar, that constitutes the second part of grammar and is also its most important part, we use principles and transformations, as well as other grammars that we have already defined. Each one of these rules is used to the main part of the grammar, indicating first the operator and then the name of the principle, the transformation or the grammar respectively.

Therefore we have the following cases of stating rules in the main part of the grammar:

principle *name of principle*
transformation *name of transformation*
grammar *name of grammar*

Apart from the above way of applying the rules that a grammar uses, we can also have a conditional application of the rules in a grammar, depending on the result from the application of some other rules.

The command that is also used by the programming languages is the following:

If *condition* **then** *rules 1* **else** *rules 2*

In the part of the control condition of this command we apply one or more rules on the X-bar structure and depending whether the result is true or false we apply the rules after **then** or after **else** respectively.

In this methodology, this command has the following form:

IfThen(*condition, rules 1)*
IfThenElse(*condition, rules 1, rules 2*)

Every rule that we define in our system is either a principle or a transformation or a grammar. When we apply this rule in an X-bar structure then it gives a true or false value, depending on whether this rule was applied successfully or not in the particular X-bar structure. This enables us to execute logical calculations in the *condition part*.

If in the *condition* we apply more than one rules, then we should use the logic operators **and, or** and **not**.

The first logic operator requires that all rules are successful in order for the *condition* to be true, while the second logic operator **or** requires at least one of the rules to be successful in order for the *condition* to be true.

The general syntactic pattern of the *condition* will be one the following:

rule 1 ***and*** *rule 2* ***and*** *rule 3* ***and***...
rule 1 ***or*** *rule 2* ***or*** *rule 3* ***or***...

The third operator **not** enables us to have a true condition only when the rule fails. In this case the general syntactic pattern of the *condition* will be the following:

not *rule*

The *rule* can be either one or more rules that are connected to each other with the operators **and** and **or**. We can also use in any combination the operators **and, or** and **not** together with the appropriate parentheses that will define the sequence of the logic calculations, in order to perform the appropriate check each time.

The same kind of rule combinations (with the operators **and, or** and not) is possible and in the main body of a grammar.

Also, the command **acceptance_level**(*Level*) exists. It returns the level of acceptance between the number of input structures that a rule is apllied on and the corresponding output structures of this rule (principle or transformation). It is possible to combine the acceptance levels of more than one rules (principles or transformations) by doing arithmetic operations and to calculate a total acceptance level. This possibility permits the implementation of an evolutionary approach in the production and checking of the manipulated X-bar structures, which is a more general and abstract that the Chomsky's minimalist ideas.

Finally we should notice that within the main part of a grammar we can have a rule that is the grammar that we enter. That means that we can perform a repeated application of the grammar. Therefore, in the main part of the grammar we can have the rule, which is the following:

grammar *name of the same grammar*

This ability enables the repeated application of a grammar's rules. Also, if we use a command of the category **if-then-else** then we can repeat the rules of a grammar only if the **if** condition is valid.

From what we described so far regarding the abilities of a grammar that we state, we observe that every grammar uses rules that we have stated. These rules apply on every X-bar structure and with the sequence that they have been defined in this grammar. The transformational rules however are able to produce one or more new X-bar trees. These trees can be used by the next rule for further processing either this rule is a principle or a transformation or a grammar.

If we wish to change the structures to which the next rule of a grammar will be applied, we can use one of the following operators:

- **addStructures**: This operator adds the structures that have been produced by the last principle, transformation on the existing X-bar trees for the next rule of the grammar.
- **setStructures**: This operator sets as X-bar trees for the next rule of the grammar, the trees that have been produced by the last principle, transformation.
- **setSucceededStructures**: This operator sets as X-bar trees for the next rule of the grammar, only the trees that the last rule has been applied on successfully.
- **restoreStructure**: This operator resets the X-bar structure for the next rule to the last X-bar tree that has gotten from the initial X-bar trees of the system.
- **getNextStructure**: This operator gets the next X-bar structure from the initial X-bar trees of the system in order to continue the application of this grammar.
- **getPreviousStructure**: This operator gets the previous X-bar structure from the initial X-bar trees of the system in order to continue the application of this grammar.
- **getParticularStructure**(*Num*): This operator gets a particular structure from the initial X-bar trees of the system according to the value of the *Num* and continues the application of this grammar.

Also, there is an operator that returns the id of an input X-bar tree. This id is a serial number that has the value 1 for the first input tree.

This operator is the following:

- **getInputTreeId**(*Id*)

Except the above operator there are two other operators that change the input structures according to the output structures that are the result of the application of the last principle or transformation. Both need as operand an *Id* as it is described in the previous operator.

The first substitutes the corresponding input trees with the output trees of the last principle or transfotmation:

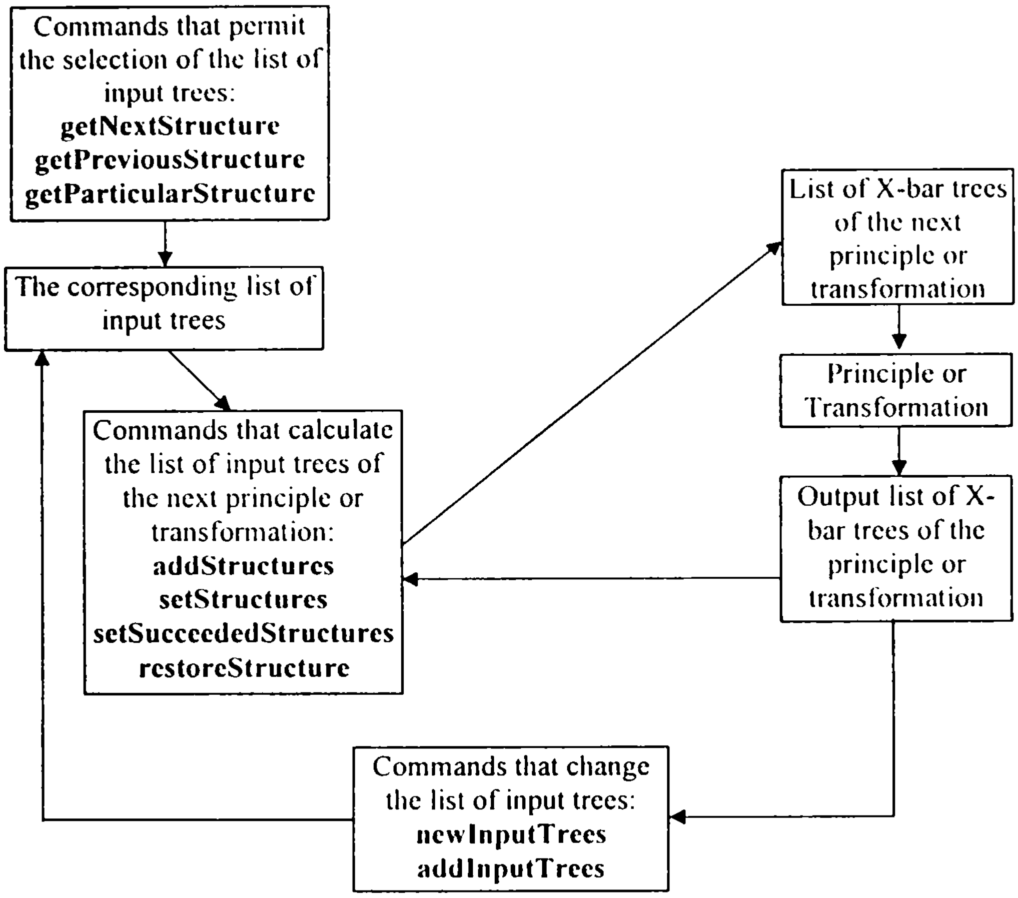
- **newInputTrees** (*Id*)

The second adds at the existing input trees the output trees of the last principle or transfotmation:

- **addInputTrees** (*Id*)

The principle has as output structures the subset of its input structures that it has applied on correctly. The transformation has as output structures the new set of structures that have been produced by it.

The following schema shows the usage of the above operators(Fouskakis, 2004c):



In order to exchange information of different X-bar trees between the different rules that are used by the grammar, there are the grammar variables. They are variables that can be used by more than one principle or transformation. They permit smaller rules that use known information from previous rules. If a grammar variable has been defined in a principle or transformation that has already used it, it is not possible to be defined again in any field of any other principle or transformation but it is only possible to use it or to change its values. It is only possible to remove the grammar variable and then to define it again in another rule. Also, it is possible to be used all the commands for variables that are used in the **structureCommands** field of the principles only rules.

We can use two operators related with grammar variables:

- **addGrammarVariable** *name of variable*
- **removeGrammarVariable** *name of variable*

The first operator defines only the name of a new grammar variable. The second operator deletes a grammar variable. These operators can be used in the main body of a grammar or even outside of a grammar to delete or declare a grammar variable that can be used in the next rules. At the case of using the operator

removeGrammarVariable, this grammar variable will not be available in the next rules or grammars. Both operators need the *name of variable* as an operand.

Finally, there are two operators that check the existence of a variable that has be declared in a principle or transformation:

- **varExists** (*Name of Variable*): it checks if a variable has already been declared
- **grammarVar** (*Name of Variable*): it checks if a variable has already been defined as grammar

The first case checks if a variable existed in the last principle or transformation that was applied. The second case checks if a variable has already been defined as grammar one. The above operators are used in the main body of a grammar and in the if-then-else structures. Also, it is possible to be used in the **structureCommands** field of the principles and transformations.

All the above operators and rules that we can use in the main part of a grammar are separated by commas and end with a full stop after the last rule. If there is a requirement that this grammar should be applied to the X-bar structures, then all the rules and operators are executed according to the sequence they are denoted in the grammar. The operators for structures manipulations and the grammar variables permit efficient checking of problems like scrambling and long distance dependences that appears outside of an x-bar phrase structure.

Next we shall give some examples of grammars:

Suppose we have the following rules:

principle 'Structure Control'
transformation 'Structure Modification'

we can define a grammar.

grammar 'Grammar 1'.
principle 'Structure Control',
transformation 'Structure Modification'.

We observe that this grammar uses two rules with the names 'Structure Control', 'Structure Modification'. The principle examines if the X-bar structure fulfills our requirements and then the transformation 'Structure Modification' is applied, that can produce one or more structures according to the structure it gets at its input. If the principle 'Structure control' fails, then the whole grammar fails as well and the transformation 'Structure modification' cannot be applied. If the principle 'Structure control' is successful and the transformation 'Structure modification' fails, then the grammar fails again.

grammar 'Grammar 2'.
principle 'Structure Control',

transformation 'Structure Modification',
setstructures,
grammar 'Grammar 2'.

We can also define a new grammar, where we have a recursive-application of the grammar and use of the operator **setstructures**. The operator **setstructures** is used so that we can each time modify the input structures of the grammar. Its input structures, after the application of the transformational rule 'Structure modification' change because of the **setstructures** operator and become the structures formed by the transformation. Thus, with the recursive-inflection of grammar we can produce consecutively structures, until we reach in structures that the principle 'Structure control' cannot accept as correct or the transformation 'Structure Modification' can't modify.

3.1.3.1 The EBNF of the grammar rules in the theory part of the system

The EBNF form of the grammar rule in the methodology is the:

grammar = "grammar" grammar-name
grammar-main-part.

grammar-main-part = grammar-part { "," grammar-part }.

grammar-part = rule|
"ifThen(" condition "," rules")" |
"ifThenElse(" condition"," rules"," rules ")" |
"addstructures" |
"setstructures" |
"setsucceededstructures" |
"restorestructure" |
"getnextstructure" |
"getpreviousstructure" |
"getparticularstructure(" number{number} ")" |
"getinputtreeid(" number{number} ")" |
"newinputtrees(" number{number} ")" |
"addinputtrees(" number{number} ")" |
"addGrammarVariable" name |
"removeGrammarVariable" name |
"varExists(" name ")" |
"grammar_var(" name ")" |

"acceptance_level(" number{number}"."number{number} ")" |
see-principle-command.

rules= "("grammar-part {"," grammar-part } ")".

condition = condition | ("(" condition operator condition ")").

condition = "not" "(" condition ")".

condition = rule.

operator = "and" | "or"

rule= "principle" principle-name |
"transformation" transformation-name |
"grammar" grammar-name.

principle-name = name.

transformation-name = name.

grammar-name = name.

The name and number are declared at the EBNF form of the structures that the methodology manipulates.

The see-principle-command is described below in the EBNF of the **structureCommands** field of the principles and transformations rules.

3.1.4 The Linguistic program

In the linguistic program we state only that part of the theory that we have described and we wish to apply on the initial X-bar structure. Our theory has been described by rules. These rules are grammars but they can also be principles and transformations.

The rules that we want to be used and applied by the system on the X-bar structures are stated as follows:

principle *name of principle*
transformation *name of transformation*
grammar *name of grammar*

We observe that we call the rules only with their name and the respective operator that precedes to the rule's name. Depending on whether the rule is a principle, a transformation or a grammar, we have accordingly the operators **principle, transformation, grammar.** The rules apply to the first X-bar tree under the sequence they are stated in the system.

Finally, we can use the operators for the declaration of the grammar variables that are described in the previous chapter.

3.1.4.1 The EBNF of the linguistic program

The EBNF form for the user's program is the following:

program-user = program_rule "." { program_rule "." }

program_rule = rule |
"addGrammarVariable" name|
"removeGrammarVariable" name

The <u>rule</u> has been stated in the previous chapter about linguistic theory.
The <u>name</u> declared at the EBNF form of the structures that the methodology manipulates.

3.2 Description of the principles and transformations fields

As it was described in a previous section, both principles and transformations have three different fields that are the following:

- **variables**
- **structureDescription**
- **structureCommands**

In the next chapters the abilities that are provided by the methodology for each one of these fields will be described.

3.2.1 The variables field

The principles and transformations, apart from their name, have as a next field the field **variables**. It contains the variables (Fouskakis, 2005a, 2005c) that are used by the next fields of principles and transformations.

The variables of this field, depending on the form of data that they can have as values, are of the following types:

1. tree node
2. tree terminal
3. anaphor
4. node features
5. subtree

The variables in this field must always have one or more values that accompany the variable with its statement. That means that we cannot enter a variable in this field unless it has at least one value.

These variables are very important for the next fields of principles and transformations. The necessary generality in the content of the fields **structureDescription** and **structureCommands** is achieved by using these variables. Thus we have the ability to define rules that are general and can be applied to several cases of the x-bar trees.

In order to define a new variable in the **variables** field of principles and transformations, it is used the following general pattern:

type of variable *name of variable* ***set*** *value of variable* ***or***
value of variable...

Regarding the above pattern, it is observed that in order to denote a variable it is necessary to give the variable type that must be one of the five types we mentioned above. The type of the variable is followed by its name that must be different for each variable. After the name we have the operator **set** that it is obligatory to use and which is followed by the values of the variable. When the values of the variable are more than one, they are separated by the operator **or**. The values of each variable depend on the type of the variable. Thus for example, for a variable of the node type, the values assigned to it will be nodes of trees.

As it is mentioned before, the variables of this field are of five different types. Depending on the type of the variable, we use an operator that will determine the type of the variable.

These operators are the following:

1. tree node, operator **node**
2. terminal element, operator **terminal**
3. anaphor, operator **anaphor**
4. node features, operator **features**
5. subtree, operator **subtree**

The name of the variable exists after the type of the variable and must be an atom of the prolog. That means that the name of the variable is a sequence of alphanumeric characters that are enclosed in quotes, unless the first character is a lower case letter and there are no empty characters.

The following are examples of valid variables names:

- 'Node 1'
- 'node 1'
- node_1

After the type and the name of the variable, there are the values of this variable. The values of each variable depend on its type and there are five cases, depending on the variable's type:

1. Tree node

The node of the tree must be in accordance to one of the following two general patterns:

- *name of the node category of the node* : **features** *features of the node*
- *name of the node category of the node*

The first pattern is used when it is additionally denoted the node's features.

The *name of the node* is an atom of the prolog and states the name of the specific node and the *category of the node* determines if the node is of the X, X', X'' type. In order to determine that the node is of the X, X', X'' type, the operators **bar, bari, barii** are used respectively.

2. Terminal element

The terminal element of a tree must be in accordance to one of the following two general patterns:

- *terminal element* : **anaphor** *name of the anaphor* :
  anaphor *name of the anaphor* : ......
- *terminal element*

The first pattern is used if the terminal element has anaphors with other terminal elements or subtrees of the X-bar tree. The second pattern is used if it doesn't have anaphors.

3. Anaphors

The anaphors have a general pattern and this is the name of the anaphor:

- *name of the anaphor*

the name of the anaphor is also an atom of the prolog.

4. Features of the node

The features of the nodes have been described in the respective chapter where the X-bar trees were described.

The general pattern is the following:

[*feature, feature,...*]

where the feature can be one of the following:

- *+name of the feature*
- *-name of the feature*
- *name of the feature*
- *name of the featureX = name of the featureY*
- [*name of the feature1, ..., name of the featureN*] = *name of the featureX*

and the *name of the feature* is an atom of the prolog.

The following are examples of such node features:

- [+human, +singular]

- [-animate]

5. Subtree

The subtree is an X-bar tree of the X, X' or X'' category. The method of describing these subtrees is the same with the one was explained in the section for the description of the X-bar structures.

Next I shall give an example of stating variables that includes all the categories of the variables:

variables

node	n1	**set**	article **bar** : **features** [singular, masculine] **or** noun **barii**
also			
terminal	t1	**set**	a **or** the
also			
anaphor	a1	**set**	i1 **or** j1 **or** k1
also			
anaphor	a2	**set**	l1 **or** &a1 **or** w1
also			
terminal	t2	**set**	the : **anaphor** &a2
also			
terminal	y	**set**	the : **anaphor** &a1
also			
features	f1	**set**	[singular, male, noun] **or** [plural, adjective]
also			
node	n2	**set**	noun **bar** : **features** &f1 **or** noun **bari** : **features** &f1
also			
node	n3	**set**	&n2
also			
subtree	s1	**set**	(**node** &n2, **terminal** man : **anaphor** &a2)
also			
subtree	s2	**set**	(**node** noun **bar**, **terminal** house)
also			
subtree	s3	**set**	(**node** noun **barii**, **empty**, (**node** noun **bari**, **subtree** &s2,**anyTree**)).

Observing the above variables of the **variables** field of the principles and transformations, it is noticed that each variable is separated from the next one with the operator **also.** Therefore, the general pattern for denoting the variables of the **variables** field is the following:

variable statement 1 **also** *variable statement 2* **also**.......

where the *variable statement* is a variable statement performed like the one that was described above.

Every variable must have a different name. If a variable has been declared as grammar and has been defined in a principle or transformation that has already been used, it is not possible to be defined again in this field of a principle or transformation.

At the definition of the new variables, the values of another variable that has already been stated can be used. This helps to designate the total of the variables' values in a more general way. Thus these variables can be used in the next fields in order to describe in general way those cases that must be covered by a rule. This generality helps especially in the **structureDescription** field of the principles and variables, when we wish to describe the appropriate subtotal of natural language trees, to which the specific principle or transformation can be applied. The abilities that are provided by these variables will be presented in the next chapters.

In order to use a variable in the value of a new variable, the following symbolism is used:

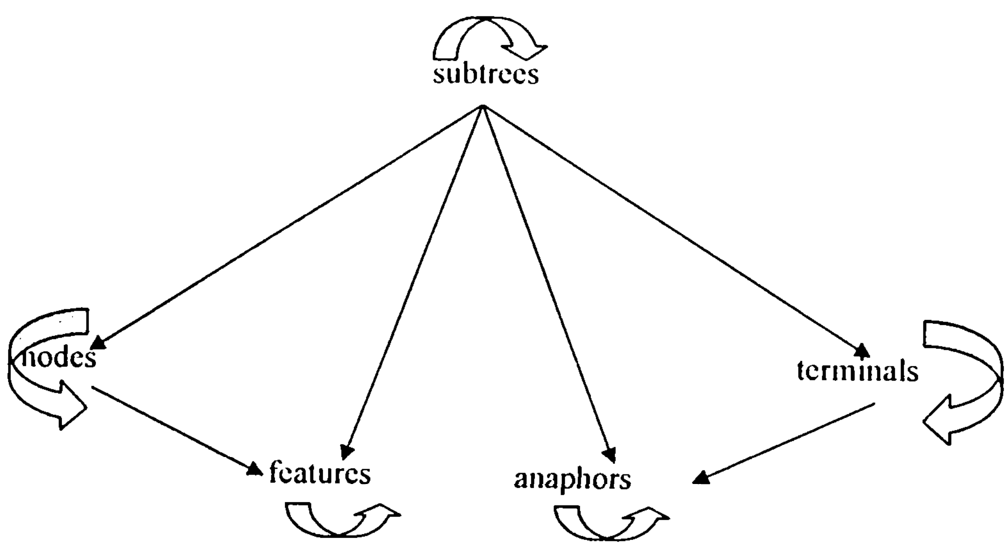
&*name of variable*

where the *name of variable* is the name of a variable that has already been stated. This variable must be stated before the new one that uses it, also can be a grammar variable.

A variable that has been stated can be used by another variable on the following cases:

1. as one of the values for the new variable and in this case, the variable must be of the same type with the new variable.
2. as part of the value of a variable and in this case, the variable must be of the same type with the element it replaces.

The following schema presents the different possibilities of using variables according to their type.



The variable at the end of the arrow is used by the other one. These possibilities facilitate the declaration of general and abstract rules that control the different cases in hierarchical way.

For the first case, there are the following variables from the above example:

1. a2
2. n3

The variable with the name a2 gets values from the variable a1. It is observed that both variables are of the same type and they are anaphors. The variable a1 has the following values: i1, j1, k1, while the variable a2 has the values 11 and w1, as well as the values of the variable a1 that has the values i1, j1, k1. Therefore, the variable a2 has the values:

- 11
- i1
- j1
- k1
- w1

It is observed that the values of the variable n3 are the same with the values of the variable n2. Also, It is observed that both variables are of the same type, they are tree nodes. Therefore, the variable n3 has the values noun bar:features &f1 and noun bari:features &f1. Both values use the variable f1 that has the following two values: [singular, male, noun] and [plural, adjective]. Therefore, the variable n3 has the following four values:

- noun **bar** : **features** [singular,male,noun]
- noun **bar** : **features** [plural,adjective]
- noun **bari** : **features** [singular,male,noun]
- noun **bari** : **features** [plural,adjective]

At the second case of the values of the variables that use variables for the replacement of certain elements in their values, we have the following variables from the above example:

1. t2
2. y
3. n2
4. s1
5. s3

The variable t2 is of the terminal data type and has a value that uses for its anaphor values the values of the variable a2. Therefore, the variable t2 has the following five values of:

- the:**anaphor** 11
- the:**anaphor** i1
- the:**anaphor** j1
- the:**anaphor** k1

- the:**anaphor** w1

The variable y is also of the terminal element type and uses for its anaphors the variable a1. Therefore, the variable y has the following three values of:

- the:**anaphor** i1
- the:**anaphor** j1
- the:**anaphor** k1

The variable n2 is of the tree node type and uses the variable f1 for the node's features. Therefore, the variable n2 has the following four values:

- noun **bar** : **features** [singular,male,noun]
- noun **bar** : **features** [plural,adjective]
- noun **bari** : **features** [singular,male,noun]
- noun **bari** : **features** [plural,adjective]

The variable s1 is of the x-bar tree type and uses the variables n2 and a1 that are of the node type and anaphor type respectively. Therefore, this variable can have all of the ten values.

- (**node** noun **bar** : **features** [singular,male,noun], **terminal** man : **anaphor** l1)
- (**node** noun **bar** : **features** [singular,male,noun], **terminal** man: **anaphor** i1)
- (**node** noun **bar** : **features** [singular,male,noun], **terminal** man: **anaphor** j1)
- (**node** noun **bar** : **features** [singular,male,noun], **terminal** man: **anaphor** k1)
- (**node** noun **bar** : **features** [singular,male,noun], **terminal** man: **anaphor** w1)
- (**node** noun **bar** : **features** [plural,adjective], **terminal** man: **anaphor** l1)
- (**node** noun **bar** : **features** [plural,adjective], **terminal** man: **anaphor** i1)
- (**node** noun **bar** : **features** [plural,adjective], **terminal** man: **anaphor** j1)
- (**node** noun **bar** : **features** [plural,adjective], **terminal** man: **anaphor** k1)
- (**node** noun **bar** : **features** [plural,adjective], **terminal** man: **anaphor** w1)

The second value of the variable n2, the noun **bari**: **features** &f1 is not possible to be used because we have an X type tree.

Finally, the variable s3 is of the x-bar tree type and uses a variable of the x-bar tree type with the name s2. Therefore, the variable s3 has the following value:

(**node** noun **barii**,
    empty,
    (**node** noun **bari**,
        (**node** noun **bar**, **terminal** house),
        anyTree)).

If in a principle or transformation no variable exists in the **variables** field, then the operator **noVariables** is used in the place of the operator **variables**.

In this paragraph, all the abilities of stating variables in the **variables** field of the principles and transformations was described.

3.2.1.1 The EBNF of the variables field

The EBNF form for stating variables in the **variables** field, which has the name variables-declaration in a previous paragraph where the principles and transformations structure was described, is the following:

variables-declaration = "(" variable-declaration
{"also" variable-declaration} ")" "." .

variable-declaration= "node" node-variable-name "set"
tree-node-value {"or" tree-node-value }.

variable-declaration= "features" features-variable-name "set"
node-features-value {"or" node-features-value }.

variable-declaration= "terminal" terminal-variable-name "set"
tree-terminal-value {"or" tree-terminal-value }.

variable-declaration= "subtree" subtree-variable-name "set"
subtree-value {"or" subtree-value}.

variable-declaration= "anaphor" anaphor-variable-name "set"
anaphor-value {"or" anaphor-value}.

anaphor-value = name | "&"anaphor-variable-name.

tree-terminal-value= (terminal-element
[":" subtree-terminal-variable-anaphors]) |
("&"terminal-variable-name).
(note: the terminal-element is declared in the chapter that describes the X-bar trees that the presented methodology manipulates)

subtree-terminal-variable-anaphors =
"anaphor" anaphor-value {":" "anaphor" anaphor-value }.

node-features-value = ("[" feature {"," feature} "]") |
("&"features-variable-name)).
(note: the <u>feature</u> is declared in the chapter that describes the X-bar trees that the presented methodology manipulates.)

tree-node-value= tree-node-value-χ''.

tree-node-value= tree-node-value-χ'.

tree-node-value= tree-node-value-χ.

tree-node-value-χ'' = "&" node-variable-name.

tree-node-value-χ' = "&" node-variable-name.

tree-node-value-χ = "&" node-variable-name.

tree-node-value-χ'' = node-name "barii"
[":" "features" node-features-value].

tree-node-value-χ' = node-name "bari"
[":" "features" node-features-value].

tree-node-value-χ = node-name "bar"
[":" "features" node-features-value].
(note: the <u>node-name</u> is declared in the chapter that describes the X-bar trees that the presented methodology manipulates)

subtree-value= subtree-value-χ'' |
subtree-value-χ' |
subtree-value-χ.

subtree-value-χ'' = "(" "node" tree-node-value-χ'' ","
subtree-value-specifier ","
(subtree-value-χ'' | subtree-value-χ') ")"

[":" subtree-terminal-variable-anaphors].

subtree-value-χ' = "(" "node" tree-node-value-χ' ","
(subtree-value-χ' | subtree-value-χ) ","
subtree-value-χ'' ")"
[":" subtree-terminal-variable-anaphors].

subtree-value-χ = "(" "node" tree-node-value-χ ","
"terminal" tree-terminal-value ")"
[":" subtree-terminal-variable-anaphors].

subtree-value-specifier = subtree-value-χ'' | subtree-value-χ.

node-variable-name = name.

features-variable-name= name.

terminal-variable-name= name.

subtree-variable-name= name.

anaphor-variable-name= name.
(note: the name is declared in the chapter that describes the X-bar trees that the presented methodology manipulates)

3.2.2 The structureDescription field of the principles and transformations

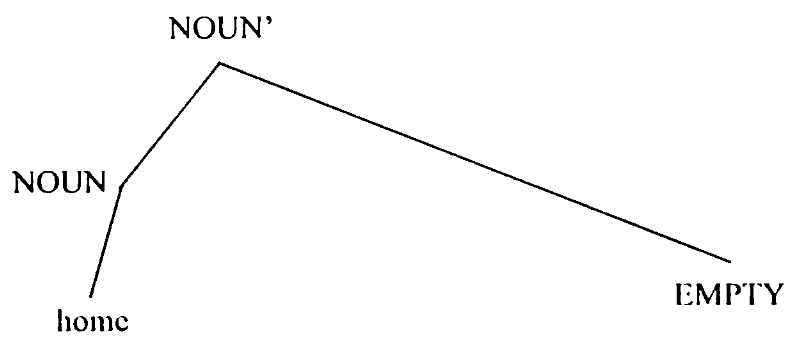
In previous chapters, the structure of the principles and transformations of the methodology was described. It is observed that both the principles and the transformations have the field **structureDescription**.

This field is used for the designation of the subtree to which the specific rule will apply; either this rule belongs to the principles category or to the transformations category. In order to apply the principle or the transformation on an X-bar tree that derives from the x-bar basic scheme, the subtree that is described in the **structureDescription** field of principles and transformations must be part of the tree or even the whole X-bar tree. By describing the structure of the **structureDescription** field is possible to have a large enough structure that most cooccurrence dependencies (predicate-argument, wh-dependencies, filler-gap dependencies) can be localized within this subtree and manipulated by the corresponding principle or transformation.

An example of the subtree in the **structureDescription** field of principles and transformations is the following:

(**node** noun **bari**, (**node** noun **bar**, **terminal** home), **empty**)

This subtree schematically is the following:



Every tree that derives from the x-bar basic scheme and has a subtree similar to the one we described above, is appropriate for the application of the specific principle or transformation that has in the **structureDescription** field the above structure.

The above subtree, apart from the specific structure, has also specific names for its nodes and its terminal elements. This subtree is of the X' category. The variable X has the value NOUN and the terminal element is the word "home". The specific structure and elements of the above tree limit the application of the specific rule in only one subtree. Therefore in order to apply the rule, it is necessary to find an X-bar tree with exactly the same subtree. This constraint however doesn't enable us to state principles and transformations that will cover the general cases of a set of trees that will have a certain common structure and characteristics to which the specific principle or transformation can be applied.

The theory that has been developed by linguists regarding the form and the characteristics of the trees, as well as the rules that should govern these trees and especially the long distance dependencies, demands general rules that should cover a lot of cases that's why the definition of this field is very important. The use of the methodology in order to study new rules requires flexibility in the way that these rules are stated either they are principles or transformations. Also, the methodology can be used for the study of natural language trees that have been produced by another system, but in these cases the definition of a small number of rules that would cover in a general way the different cases regarding the natural language trees is also necessary in order to have an efficient processing system. This is more important in embedded systems that have reduced resources and the recursion and repetition of other theories can be eliminated by defined the appropriate structures in this field. Due to the above, it was found necessary to develop a group of appropriate operators, as well as the use of variables in this field of principles and transformations.

Also an assumption is stated:

> If the tree of **structureDescription** field or a subtree of this tree contains less anaphors or features of nodes than the X-bar tree in its corresponding position, the rule is possible to be applied on this tree.

This assumption is based on the principle:

> If the required information for the application of this rule exists in a X-bar tree then it is possible to apply on it this principle or transformation. The examination of this field is from left to right.

The principles and the transformations are the most important part of the methodology and constitute the base for the statement of the more complicated rules that are the grammars.

3.2.2.1 The variables in the structureDescription field of principles and transformations

A group of variables can be used in the **structureDescription** field of principles and transformations. These variables enable the declaration of principles and transformations in a general way.

There are two categories of these variables (Fouskakis, 2005c):

- the general variables that are the variables of the **variables** field of the principles and transformations
- the transformation variables that are declared in the **structureDescription** field of the principles and transformations and are used in the **structureCommands** field. Their purpose is the declaration of the transformations of the X-bar trees.

The variables of first category can be either variables that have already been defined in the field **variables** or new variables. If a variable has already been defined then it must be of the same type with the corresponding element of the **structureDescription** structure that it substitutes. This variable constraints the corresponding element of an X-bar tree that the rule is applied on, in a specific set of values. Also, we can use new variables of the **variables** type. They are defined automatically the first time they appear in the **structureDescription** structure by taking their values from the corresponding element of the X-bar structure where this rule is applied on. The main importance of these variables is that they provide an easy way to check if two or more elements of the **structureDescription** structure are of the same type and have the same values.

The variables of the second category can be of type node of tree, terminal element or subtree. They can be used in combination with the other category of variables. The result of its definition is the declaration of a new variable. The name of this variable is the name that follows the **transformationVariable** operator. The type of this variable is the type of the corresponding element of the **structureDescription** structure. The initial value of this variable is the value that has the corresponding element of the X-bar structure on which the rule is applied.

We shall present them in the following chapters in details.

3.2.2.2 The variables of the general category

As it was mentioned in a previous chapter where the variables of the **variables** field of principles and transformations were described, there are the following types of variables:

1. anaphor
2. terminal
3. features of the tree node
4. tree node
5. tree

It is know that each variable that is declared in a principle or transformation must have a different name. That means that two different variables are not allowed to have the same name in a principle or a transformation. Special care must be taken for the variables that have been declared as grammar variables. Their functionality was described in the linguistic theory chapther.

In the **structureDescription** field of principles and transformations, in order to use the variables of their **variables** field, the following format is used:

& *name of the variable of the* **variables** *field*

where the name that is the name of a variable that was declared in the **variables** field of principles and transformations. Therefore, in order to use a variable that was declared in the **variables** field of the **structureDescription** field of principles and transformations, the character & is used followed by the name of that particular variable.

According to what has been mentioned up to now, there are the following cases of using the variables of the **variables** field:

1. **node** &*variable name*
2. **node** *node name and node category*: **features** &*variable name*
3. **terminal** &*variable name*
4. **terminal** *terminal element*: **anaphor** &*variable name*
5. **subtree** &*variable name*
6. **subtree** &*variable name*:**anaphor** *anaphor name*
7. **subtree** &*variable name*:**anaphor** &*variable name*
8. *subtree*: **anaphor** &*variable name*

From all the above cases the respective part of the subtree that is described in the **structureDescription** field, can be replaced by one of the above.

In the first case, a node is replaced with a variable of the node category. In the second case, the node's features are replaced with a variable of the **variables** field. In the third case, a terminal is replaced of the subtree in the **structureDescription** field with a variable. In the fourth case, the anaphor of the terminal is replaced with a variable. In the fifth case, a whole subtree is replaced by a variable. In the sixth case, the subtree is replaced by a variable, while the possible anaphors of the tree are given. In the seventh case, the subtree and its anaphors are replaced by two different variables. In the eighth case, only the anaphors of a tree are replaced by a variable.

In all the above cases, the respective operator that designates the type of the variable must be used in front of every variable.

The operators are the following:

1. **node**	for tree node
2. **features**	for features of node
3. **terminal**	for terminal element
4. **anaphor**	for anaphor of terminal element or subtree
5. **subtree**	for subtree

In all the above cases the variables have already been declared in the **variables** field of the specific principle or transformation. There is however a possibility to use variables of the **variables** field category that are not stated in the **variables** field of the specific rule. In this case these eight different cases also apply. There are however two more cases of variables that fall to this category.

These two cases are the following:

1. **node** &*node variable name*: **features** &*variable name*
2. **terminal** &*terminal variable name*: **anaphor** &*variable name*

In the first case the variable of the features is already known, that is, it must have values. In the second case the variable for the anaphor must also have values. However, the variables for the node in the first case and for the terminal element in the second case must be new.

When a variable is not declared then a new variable is defined automatically that has as name, the name that is used in the **structureDescription** field and type, the type that is declared by the respective operator that is before the variable. The values that this new variable will have depend on the X-bar tree that will use the specific principle or transformation. That means that the value of the new variable will be the element of the specific X-bar tree used by the specific principle or transformation in the specific place, as this is designated by the subtree of the **structureDescription** field.

A variable can be used more than once in the subtree that is described in the **structureDescription** field of principles and transformations. If a new variable is used in the **structureDescription** field more than once, then the first time it will have its value automatically from the X-bar structure as if it was declared in the **variables** field of principles and transformations. Therefore, when the same variable is reused in the subtree of the **structureDescription**, then this variable will have values and the respective element of the X-bar tree should be the same with one of the values of this variable.

The great utility of this ability is that it is easy to check if two elements of the subtree in the **structureDescription** field of principles and transformations are the same, without considering the possible values of these elements.

Finally, it must be stressed that when a node of the **structureDescription** field structure is associated with a node of the X-bar tree, apart from the fact that this node must have the same name and the same type, the features of the **structureDescription** structure node must be either the same with the features of the X-bar tree that uses the rule, or a subtotal of them. It also applies for the terminal elements that the terminal element of the **structureDescription** structure must be the same with the respective terminal element of the X-bar tree and that the anaphors of the terminal in the **structureDescription** field must all exist as anaphors in the respective element of the X-bar tree.

Next a series of examples is presented in order to explain the utility of the variables that were described above.

Example 1

In this example we wish to define a rule that will apply only to those trees that include one of the following nodes:

V, N that correspond to the words Verb and Noun respectively

These nodes are of the X category.

The tree that the rule seeks is the following:

V or N

The terminal element

The rule that we need in this case is the following:

principle	'Example 1'.
variables	**node** n1 **set** 'V' **bar or** 'N' **bar**
structureDescription	**(node &**n1,**terminal &**anyTerm)

In this rule we do not use the **structureCommands** field of principles and transformations because it is not necessary in this example.

The principle that we described above has as name the 'Example 1'. It also has a variable declared in the **variables** field under the name n1 and having as values the 'V' **bar** and 'N' **bar**. In the **structureDescription** field we described the subtree that can be seen above. This subtree is of the X category and uses for the node a variable under the name n1 that has the two known values. As a result, this rule identifies only the trees that have a subtree of the X category with node name either 'V' or 'N'. Apart from the node though, the subtree of the X category has also the terminal element connected to this node. However, we are not concerned with the values that the terminal element will have, this is why we use a variable that has not been declared in the **variables** field and does not have values that constrain us. According to those mentioned above about the function of the variables, the variable with the name anyTerm will get values from the terminal element that exists in the respective place of the X-bar tree that uses this rule.

Example 2

In this example we shall define a rule that will identify those trees that have a subtree of the X category and the name of the node will be Noun.
This node can have one of the following features:

a) [+human,+singular]
b) [+singular,+nominative]

The features of the node always express grammatical and semantic information.

Thus we can say schematically that the expressed subtree of this rule is the following:

Noun : **features** [+human,+singular] **or** [+singular,+nominative]

Any terminal

The rule that describes the above is the following:

principle 'Example 2'.
variables **features** fl **set** [+human,+singular] **or** [+singular,+nominative]
structureDescription
(**node** 'Noun' **bar:features** **&**fl,**terminal** **&**anyTerminal)

The principle that we described above has the name 'Example 2'. It also has a variable of the **features** type stated in the **variables** field under the name fl and values of the [+human,+singular] and [+singular,+nominative]. In the **structureDescription** field we described the subtree that appears on the figure above. This subtree is of the X category with Noun as a node name and features that are assigned by the variable fl that has two known values. As a result, this rule identifies only those trees that have a subtree of the X category with node either 'Noun' **bar** : features [+human,+singular] or 'Noun' **bar** : features [+singular,+nominative]. Apart from the node however, the subtree of the X category has also the terminal element connected to this node. However, we are not concerned with the values that the terminal element will have, this is why we use a variable that has not been declared in the **variables** field and does not have values that constrain us. According to those mentioned above about the function of the variables, the variable with the name anyTerminal will get values from the terminal element that exists in the X-bar tree that uses this rule.

Example 3

In this example we shall define a rule that will identify those trees that have a subtree of the X category and one of the following terminal elements:

a) run
b) drink
c) play
d) drive

Schematically we can say that the expressed subtree of this rule is the following:

Any node X category

terminal 'run' **or** 'drink' **or** 'play' **or** 'drive'

The rule that describes the above is the following:

principle 'Example 3'.
variables **terminal** t1 **set** 'run' **or** 'drink' **or** 'play' **or** 'drive'
structureDescription
(**node** &anyNode,**terminal** &t1)

The principle that we described above has the name 'Example 3'. It also has a variable of the **terminal** type stated in the **variables** field under the name t1 and has as values the 'run', 'drink', 'play', 'drive'. In the **structureDescription** field we described the subtree that appears on the figure above. This subtree is of the X category. The name of the subtree's node does not concern us this is why we use a variable that has not been declared in the **variables** field of the above principle. This variable has the name anyNode and since it doesn't have initial values, it takes values from the X-bar tree. More specifically, the value that this variable will have will be the node that the X-bar tree has in its respective position. The terminal element of this X category subtree must be one of those that are assigned as values to the variable of the **terminal** type t1 that has been stated in the **variables** field of this principle. Therefore, this rule can be applied to X-bar trees that have a subtree of the X category and one of the 'run', 'drink', 'play', 'drive' as terminal elements for this subtree.

Example 4

In this example we shall define a rule that will identify only those trees that have a subtree of the X category with terminal element the article "the". Also, they will be bound either to a reflexive pronoun or to the trace that results from the moving of this element from the place that it has occupied to the new one that it occupies now.

These two types of binding are separated by their name that we consider being the **anaphorPro** for the first one and the **anaphorTrace** for the second one.

Schematically we can say that the subtree that the X-bar tree should have is the following:

Any node of X category

terminal 'the' : **anaphor** anaphorPro **or** anaphorTrace

The principle that corresponds to the above is the following:

principle 'Example 4'.
variables
  anaphor a1 **set** anaphorPro **or** anaphorTrace
structureDescription
  (**node** &anyNode, **terminal** 'the'**:anaphor** &a1)

The principle that we described above has the name 'Example 4'. This rule has in the **variables** field a variable of the **anaphor** type under the name a1. This variable has two values, the anaphorPro and the anaphorTrace. In the **structureDescription** field of the rule we describe the subtree of the X category that appears schematically above. In this subtree we are not concerned with the X category node this is why we use a variable with the name anyNode that has not been declared in the **variables** field of the specific principle and as a result, it has no specific values. This variable takes values from the X-bar tree that uses this rule. The value of the variable will be the node that exists in the respective position of the X-bar tree. The terminal tree however that follows the node is specified and must have one of the two anaphors, either anaphorPro or anaphorTrace. The requirement that the terminal element should have one of the above anaphors is covered by the use of the variable a1 that we have stated in the **variables** field of the specific principle.

Example 5

In this example we shall define a rule that will identify those X-bar trees that have one of the following subtrees of the X category:

a) (**node**, article **bar**, **terminal** 'the')
b) (**node** noun **bar**, **terminal** 'home').

Schematically, the X-bar trees should include the following subtrees:

node article **bar** | **node** noun **bar**

terminal 'the' | **terminal** 'home'

The principle that describes the above is the following:

```
principle      'Example 5'.
variables
subtree s1 set (node article bar, terminal 'the') or
               (node noun bar, terminal 'home')
structureDescription
        subtree &s1
```

The principle that we described above has the name 'Example 5'. This rule has in the **variables** field a variable of the **subtree** type under the name s1. This variable has two values, the (node, article bar, terminal "the") and (node noun bar, terminal "home"). In the **structureDescription** field of this principle we describe the subtree that the input structure must have. In the **structureDescription** field of the above principle, the subtree is designated by the variable s1. As a result, the subtree of the **structureDescription** field is either (**node**, article **bar**, **terminal** 'the') or (**node** noun **bar**, **terminal** 'home').

Example 6

In this example we shall define a rule that will identify those X-bar trees that have one of the following subtrees of the X category:

a) (**node** article **bar**, terminal 'the')
b) (**node** noun **bar**, terminal 'home')

with an anaphor that will have the name anaphorTrace.

Schematically, the X-bar trees should include the following subtrees:

node article **bar**

anaphor anaphorTrace

terminal 'the'

node noun **bar**

anaphor anaphorTrace

terminal 'home'

The principle that describes the above is the following:

principle 'Example 6'.
variables
subtree s1 **set** (**node** article **bar**, **terminal** 'the') **or**
(**node** noun **bar**, **terminal** 'home')
structureDescription
subtree &s1 :**anaphor** anaphorTrace

The principle that we described above has the name 'Example 6'. This rule has in the **variables** field a variable of the **subtree** type under the name s1. This variable has two values, the (node, article bar, terminal "the") and (node noun bar, terminal "home"). In the **structureDescription** field of this principle we describe the subtree that the input structure must have. In the **structureDescription** field of the above principle, the subtree is designated by the variable s1 followed by the anaphor under the name anaphorTrace. As a result, the subtree of the **structureDescription** field is either (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorTrace or (**node** noun **bar**, **terminal** 'home'):**anaphor** anaphorTrace.

Example 7

In this example we shall define a rule that will identify those X-bar trees that have an anaphor with the name anaphorTrace.

The principle that describes the above is the following:

principle 'Example 7'.

noVariables.
structureDescription
subtree &anyTree:**anaphor** anaphorTrace

The principle that we described above has the name 'Example 7'. This rule has no variables in the **variables** field this is why we replace the operator **variables** with the operator **noVariables**. In the **structureDescription** field of this principle we describe the subtree that the input structure must have. Since we are not interested in the form of the sutree of the X-bar tree that the rule accepts, but only in having an anaphor with the name anaphorTrace, we use a variable with the name anyTree that has no value. This variable takes each time as a value the subtree of the X-bar tree that has an anaphor with the name anaphorTrace.

Example 8

In this example we shall define a rule that will identify those X-bar trees that have an anaphor with the name anaphorTrace or anaphorPron.

The principle that describes the above is the following:

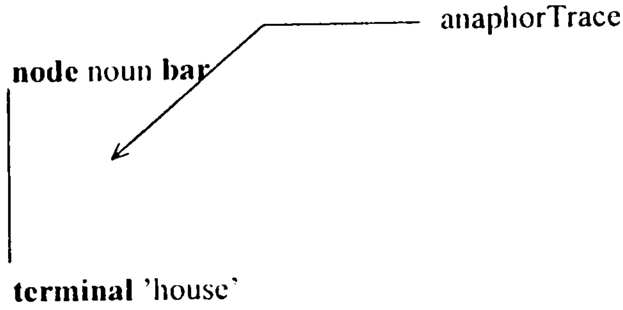
principle 'Example 8'.
variables
anaphor a1 **set** anaphorPron **or** anaphorTrace
structureDescription
subtree &anyTree:**anaphor** &a1

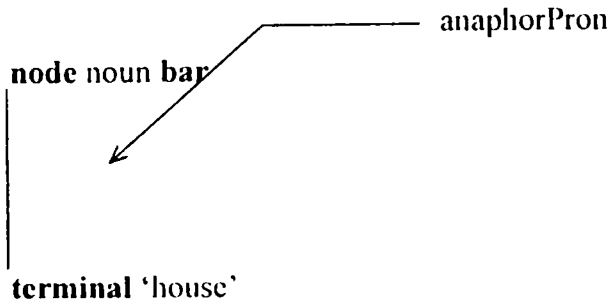
The principle that we described above has the name 'Example 8'. This rule has in the **variables** field a variable with the name a1. It also has the values anaphorPron and anaphorTrace. These two values of the variable are two different anaphors that the X-bar trees can have. In the **structureDescription** field of this principle we describe the subtree that the input structure must have. Since we are not interested in the form the subtree of the X-bar tree that the rule accepts, but only in having an anaphor with the name anaphorTrace or anaphorPron, we use a variable with the name anyTree that has no value. This variable takes each time as a value the subtree of the X-bar tree that has an anaphor with the name anaphorPron or anaphorTrace. The anaphors of the subtree anyTree are specified by the variable a1.

Example 9

In this example we shall define a rule that will identify the X-bar tree of the X category (**node** noun **bar**, **terminal** 'house') that has as anaphor either the anaphorTrace or the anaphorPron.

Schematically, the X-bar trees should include the following subtrees:





The principle that describes the above is the following:

principle 'Example 9'.
variables
    anaphor a1 **set** anaphorTrace **or** anaphorPron
structureDescription
    (**node** noun **bar**, **terminal** 'house'):**anaphor** &a1

The principle that we described above has the name 'Example 9'. This rule has in the **variables** field a variable of the **anaphor** type with the name a1. This variable has two values, the anaphorTrace and the anaphorPron. In the **structureDescription** field of this principle we describe the subtree that the input structure must have. On the above principle the subtree to the **structureDescription** field is the (**node** noun **bar, terminal** 'house'). This subtree however must have an anaphor with the name anaphorTrace or anaphorPron, this is why we use a variable with the name a1 that has the above two values.

Example 10

In this example we shall define a rule that will identify those trees that have a subtree of the X category, but we are not interested in the name of this subtree node. This node is of the X category and must have one of the following features:

a) [+human, +singular]
b) [+singular, +nominative]

The terminal element of this node is not of our interest either.

Schematically we can say that this rule will express the following subtree:

node : **features** [+human, +singular] **or** [+singular, +nominative]

A terminal

The rule that describes the above is the following:

principle 'Example 10'.
variables
    features f1 **set** [+human, +singular] **or**
        [+singular, +nominative]
structureDescription
    (**node** &anyNode:**features** &f1, **terminal** &anyTerminal)

The principle that we described above has the name 'Example 10'. It also has a variable of the **features** type stated in the **variables** field with the name f1 and has the values [+human,+singular] and [+singular,+nominative]. In the **structureDescription** field we have described the subtree shown above. This subtree is of the X category and the name of its node is not of our interest, this is why we use a variable that doesn't have any value. The name of this variable is anyNode. Also, this node must have features that should be either [+human,+singular] or [+singular,+nominative]. This constraint is achieved with the variable f1 that is used in the place of this node's features. Also, we are not interested in the terminal element of the particular subtree, this is why we use a variable that doesn't have values yet. The name of this variable is anyTerminal.

Example 11

In this example we shall define a rule that will identify only those X-bar trees that have a subtree of the X category with a terminal element, that will be bound either with a relative pronoun or with the trace that results from this element's moving from the place it occupied to the new one it occupies now.

These two types of binding are separated by their name that we consider to be anaphorPro for the first and anaphorTrace for the second one.

Schematically, the X-bar tree should include the following subtree:

A node X category

Terminal element : **anaphor** anaphorPro **or** anaphorTrace

The principle that corresponds to the above is the following:

principle 'Example 11'.
variables
anaphor a1 **set** anaphorPro **or** anaphorTrace
structureDescription
(**node** &anyNode, **terminal** &anyTerm:**anaphor** &a1)

The principle that we described has the name 'Example 11'. This rule has in the **variables** field a variable of the **anaphor** type with the name a1. This variable has two values, the anaphorPro and the anaphorTrace. In the **structureDescription** field of this principle we describe the subtree of the X category that is shown on the above figure. In this subtree we are not concerned with the X category node, this is why we use a variable with the name anyNode that has not been stated in the **variables** field of the specific principle and as a result, it doesn't have specific values. This variable takes the values from the X-bar tree that uses this rule. The value of the variable will be the node that exists in the respective place of the X-bar tree. Also, we are not concerned with the terminal element, this is why we use a variable with the name anyTerm that has no values and takes its values from the X-bar tree. It is required however that the terminal element has one of the two anaphors, the anaphorPro or the anaphorTrace. That's why we use the variable a1 that has these two values.

3.2.2.3 The variables of the transformation category

Apart from the variables in the **structureDescription** field of principles and transformations that are described so far and that they belong in the category of the variables that are stated in the **variables** field, there is another category of variables that are used to perform the transformations.

These variables can be stated only in the **structureDescription** field of principles and transformations and they are used by the **structureCommands** field of these rules.

The variables of this category can be one of the following types:

a) tree node
b) terminal
c) subtree

The various ways of declared the transformation variables at the elements of the above types are:

1. **node** *node* : **transformationVariable** *variable name*
2. **node** **&***node type variable name*: **transformationVariable** *variable name*
3. **node** (*node* : **features** *node features*) : **transformationVariable** *variable name*
4. **node** (*node* : **features** **&***node features variable name*) : **transformationVariable** *variable name*
5. **node** (**&***node* : **features** **&***node features variable name*) : **transformationVariable** *variable name*
6. **terminal** *terminal element* : **transformationVariable** *variable name*
7. **terminal** **&***terminal element variable name*: **transformationVariable** *variable name*
8. **terminal** (*terminal element*: **anaphor** *anaphor name*) : **transformationVariable** *variable name*
9. **terminal** (*terminal element*: **anaphor** **&***anaphor variable name*) : **transformationVariable** *variable name*
10. *subtree* : **transformationVariable** *variable name*
11. **subtree** **&***subtree variable name*: **transformationVariable** *variable name*
12. (*subtreee*: **anaphor** *anaphor name*) : **transformationVariable** *variable name*
13. (*subtreee* : **anaphor** **&***anaphor variable name*) : **transformationVariable** *variable name*
14. **subtree** (**&***subtree variable name*: **anaphor** *anaphor name*) : **transformationVariable** *variable name*
15. **subtree** (**&***subtree variable name*: **anaphor** **&***anaphor variable name*) : **transformationVariable** *variable name*

In the above cases the variables that are symbolized as:

&*name of the variable*

can already be stated and have values, but they can also be unstated and take a value from the X-bar tree. In case a variable is not stated then it is stated and gets values automatically, as it was described in the prevcious chapter.

Each of the above cases results in the declaration of a new variable of the transformation category. The name of this variable is the name that follows the **transformationVariable** operator. The type of the variable is the type of the respective element of the **structureDescription** structure. Therefore, in the cases from 1 to 5, the new variable is of the node type and in the cases from 6 to 9 the new variable it of the terminal element type. In the cases from 10 to 15 the new variable is of the subtree type.

The value that this variable will have initially is the corresponding element of the X-bar tree that occupies this place.

Next examples are analyzed that are according to the above cases of stating transformation variables.

Example 1

We shall define a rule that will use a variable of the **transformationVariable** category for the node named noun of the X category subtree. The terminal element of the node may be one of the following:

a) house
b) table
c) chair

The rule will recognize the following subtree:

node noun **bar** : **transformationVariable** nv1

terminal 'house' or 'table' or 'chair'

This rule is expressed as follows:

transformation 'Example 1'
variables
  terminal t1 **set** 'house' **or** 'table' **or** 'chair'
structureDescription
  (**node** noun **bar: transformationVariable** nv1,
  terminal &t1)

The above rule is a transformation that has the name "Example 1". This rule doesn't have the **structureCommands** field, because we are not interested in these examples in demonstrating the abilities of the transformation that the system provides us. This rule has a variable of the terminal element type that has the name t1 and has the values 'home', 'table', 'chair'. This variable is used in order to specify the terminal element of the X category subtree. In the **structureDescription** field of the transformation we describe the subtree of the category X that should exist in the input structure. In the tree's node that is 'noun **bar**' we state a variable of the **transformationVariable** category under the name nv1. This variable enables us, as we shall see in the following chapters, to transform the X-bar tree by changing the node and adding, for example, features to this node.

Example 2

We shall define a rule that will use a variable of the **transformationVariable** category for the node of an X category tree, with one of the following as a terminal element:

a) house
b) table
c) run

The node of this subtree can be either noun **bar** or verb **bar**. We will also see the same example for the case where we don't have specific name of the node of the X category subtree.

The rule will recognize the following subtree:

node noun or verb **bar** : **transformationVariable** nv1

terminal 'house' or 'table' or 'run'

This rule is expressed as follows:

transformation 'Example 2'
variables
  terminal t1 **set** 'house' **or** 'table' **or** 'run'
  node n1 **set** noun **bar** **or** verb **bar**

structureDescription
    (**node &**n1**: transformationVariable** nv1,**terminal &**t1)

The above rule is a transformation that has the name "Example 2". This rule has a variable of the terminal element type that has the name t1 and has the values 'home', 'table'. 'run'. It also has a variable of the node type under the name n1 that has the values noun **bar** or verb **bar.** The variable t1 is used in order to specify the terminal element of the X category subtree in the **structureDescription** field of the transformation. The variable n1 is used in order to specify the node of the subtree in the **structureDescription** field.

In the node of the subtree in the **structureDescription** field we state a variable of the **transformationVariable** category under the name nv1. This variable enables us, as we shall see in the following chapters, to transform the X-bar tree by changing the node and adding, for example, features to this node. The result of the transformation is the change of the node in the X-bar tree and the addition, for example, features.

As an example, the node noun **bar** can become noun **bar : features** [animate].

Apart from the above case, where we know the names of the nodes, maybe we do not wish to constrain the rule in specific nodes. In this case, the rule is as follows:

transformation 'Example 2'
variables
    terminal t1 **set** 'house' **or** 'table' **or** 'run'
structureDescription
    (**node &**n1**: transformationVariable** nv1,**terminal &**t1)

In this transformation apply everything we described for the previous one. The difference is that this transformation uses for the node of the tree a variable with name n1. This variable is not stated in the **variables** field of the rule. Therefore, the above rule can apply to any tree that has an X category subtree and as a terminal element one of the 'house', 'table' and 'run'.

Example 3

In this example we shall define a rule that will specify a variable of the **transformationVariable** category for the node with name noun and the features [+animate, +singular] of an X category subtree. The terminal element of this subtree may be one of the following:

a) house
b) table
c) chair

The rule will recognize the following subtree:

node (noun **bar** : **features** [+animate,+singular]): **transformationVariable** nv1

terminal 'house' or 'table' or 'chair'

This rule is expressed as follows:

```
transformation 'Example 3'
variables
     terminal t1 set 'house' or 'table' or 'chair'
structureDescription
     (node (noun bar : features [+animate,+singular]):
                    transformationVariable nv1, terminal &t1)
```

The above rule is a transformation that has the name "Example 3". This rule has a variable of the terminal element type that has the name t1 and has the values 'home', 'table' and 'chair'. This variable is used in order to specify the terminal element of the X category subtree. In the **structureDescription** field of the transformation we describe the subtree of the category X that should exist in the input structure. In the tree's node that is noun **bar:features** [+animate,+singular] we state a variable of the **transformationVariable** category under the name nv1. This variable enables us to change the node's features and to add the feature +nominative. As a result of this transformation, the respective node of the X-bar tree becomes as follows:

noun **bar** : **features** [+animate,+singular,+nominative]

Example 4

In this example we shall define a rule that will use a variable of the **transformationVariable** category for the node of the X category subtree. The name of this node is noun and its features can be either [-animate, +singular] or [-animate, +nominative]. Also, the terminal element of this subtree can be anything.

The rule will recognize the following subtree:

node (noun **bar** : **features** [-animate,+singular] or [-animate,+nominative]): **transformationVariable** nv1

Any terminal

This rule is expressed as follows:

```
transformation 'Example 4'
variables
    features f1 set [-animate,+singular] or [-animate,+nominative]
structureDescription
    (node (noun bar : features &f1): transformationVariable nv1,
        terminal &anyTerm)
```

The above rule is a transformation that has the name "Example 4". This rule has a variable of the node features type that has the name f1 and the values [-animate, +singular] and [-animate, +nominative]. This variable is used to specify the features of the node in the X category subtree. In the **structureDescription** field of the transformation we describe the subtree of the category X that should exist in the input structure. The node of the tree can be either noun **bar : features** [-animate,+singular] or noun **bar : features** [-animate,+nominative]. In this node we assign a variable of the **transformationVariable** category under the name nv1. This variable enables us to change, for example, the node's features and to leave only the feature +animate. As a result of this transformation, the respective node of the X-bar tree becomes as follows:

noun **bar : features** [+animate]

The terminal of the subtree can be anything. This is why in the respective place of the subtree we use a variable with no initial value.

Example 5

In this example we shall define a rule that will use a variable of the **transformationVariable** category for the node of the X category subtree. The node can have any name and its features can be either [+animate,+singular] or [+animate,+nominative]. Also, the subtree can have any terminal element.

The rule will recognize the following subtree:

node (node name **bar** : **features** [+animate,+singular] or [+animate,+nominative]): **transformationVariable** nv1

Any terminal

This rule is expressed as follows:

transformation 'Example 5'
variables
features fl **set** [+animate,+singular] **or** [+animate,+nominative]
structureDescription
(**node** (&anyNode: **features** &fl): **transformationVariable** nv1,
terminal &anyTerm)

The above rule is a transformation that has the name "Example 5". This rule has a variable of the node features type that has the name fl and the values [+animate,+singular] and [+animate,+nominative]. This variable is used to specify the features of the node in the X category subtree. In the **structureDescription** field of the transformation we describe the subtree of the category X that should exist in the input structure. The node of the tree can have any name. This is why we use a variable that has not been stated in the **variables** field of the transformation. We also use the features [+animate,+singular] or [+animate,+nominative]. In this node we assign a variable of the **transformationVariable** category under the name nv1. This variable enables us to change, for example, the node's features and to set as a feature the +nominative. As a result of this transformation, the respective node of the X-bar tree becomes as follows e.g.:

noun **bar** : **features** [+nominative]

The subtree can have any terminal element. This is why in the respective place of the subtree we use a variable with no initial value.

Example 6

In this example we shall define a rule that will use a variable of the **transformationVariable** category in a terminal element of the X-bar tree. The subtree, to which this rule will apply, shall be a subtree of the X category having as a terminal element the word 'door'.

Schematically, the X-bar tree should include the following subtrees:

Any node of category X

terminal 'door' : **transformationVariable** sv1

The rule that expresses the above is the following:

transformation 'Example 6'
noVariables.
structureDescription
(**node** &anyNode, **terminal** 'door': **transformationVariable** sv1)

The above transformation has the name "Example 6". This rule doesn't have any variable in the **variables** field. In the **structureDescription** field of the transformation we describe the subtree of the category X that should exist in the X-bar tree that uses this particular transformation, in order to apply this transformation on that tree. The subtree of the **structureDescription** field is of the X category, has a node that can have any name and features, and the word 'door' as a terminal element. In this terminal element we assign a variable of the **transformationVariable** category under the name sv1. This variable enables us to change the terminal node. By applying the appropriate transformation we can, for example, change the word and make it a window, or to add an anaphor.

Example 7

In this example we shall define a rule that will specify a variable of the **transformationVariable** category in a terminal element of the X-bar tree. The subtree to which this rule will apply, shall be a subtree of the X category having as a terminal element the word 'door' or the word 'window'.

Schematically, the X-bar tree should include the following subtrees:

Any node of category X

terminal 'door' or 'window' : **transformationVariable** sv1

The rule that expresses the above is the following:

```
transformation 'Example 7'
variables
   terminal t1 set 'door' or 'window'
structureDescription
   (node &anyNode, terminal &t1: transformationVariable sv1)
```

The transformation has the name "Example 7". It also has a variable in the **variables** field. This variable has the name t1 and the values 'door' and 'window'. In the **structureDescription** field we describe the subtree of the category X that should exist in the X-bar tree that uses this particular transformation, in order to apply this transformation on that tree. The subtree of the **structureDescription** field is of the X category, has a node that can have any name and features, and the word 'door' or 'window' as a terminal element. These two values of the terminal element are given by the variable t1. In this terminal element we assign a variable of the **transformationVariable** category with the name sv1. This variable enables us to change the terminal element. By applying the appropriate transformation we can, for example, change the word of the X-bar tree and make it a 'roof', or to add an anaphor to this terminal element.

Example 8

In this example we shall define a rule that will specify a variable of the **transformationVariable** category in a terminal element of the X-bar tree. The subtree to which this rule will apply, shall be a subtree of the X category having as a terminal element the word 'door' bound to the name anaphorTrace.

Schematically, the rule will apply to the following subtree:

Any node category X

terminal ('door':**anaphor** anaphorTrace): **transformationVariable** sv1

The following transformation expresses the above requirements:

```
transformation 'Example 8'
```

noVariables.
structureDescription
(**node** &anyNode, **terminal** ('door':**anaphor** anaphorTrace):
transformationVariable sv1)

The above rule is a transformation that has the name "Example 8". This transformation doesn't have a variable in the **variables** field. In the **structureDescription** field of the above rule we describe the subtree of the X-bar tree, to which this rule will apply. This subtree can have any node of the X category. This is why we use a variable with the name anyNode that has no values. Also, this subtree must have a terminal element which is the word 'door':**anaphor** anaphorTrace and to which we assign a variable of the **transformationVariable** type with the name sv1. This variable enables us to transform the terminal element of the X-bar tree that exists in its respective place.

Thus, for example:

the terminal 'door':**anaphor** anaphorTrace
can become 'window' :**anaphor** anaphorTrace or
'door': **anaphor** anaphorPronoun.

Example 9

In this example we shall define a rule that will specify a variable of the **transformationVariable** category in a terminal element of the X-bar tree. The subtree, to which this rule will apply, shall be a subtree of the X category having as a terminal element the word 'door' bound either to the anaphorTrace or the anaphorPronoun.

We shall also define a rule to which the above will apply, only that we will have no constraints from special anaphors.

Schematically, the rule will apply to the following subtree:

Any node category X

terminal ('door':**anaphor** anaphorTrace or anaphorPronnoun):
transformationVariable sv1

The following transformation expresses the above requirements:

transformation 'Example 9'
variables
anaphor a1 **set** anaphorTrace **or** anaphorPronoun
structureDescription
(**node** &anyNode, **terminal** ('door':**anaphor** &a1):
transformationVariable sv1)

The transformation has the name "Example 9". This transformation has a variable of the **variables** field. This variable is of the anaphor category and has the name a1 and the values anaphorTrace and anaphorPronoun. In the **structureDescription** field of the above rule we describe the subtree of the X-bar tree, to which this rule will apply. This subtree can have any node of the X category. This is why we use a variable with the name anyNode that has no values. Also, this subtree must have a terminal element which is the word 'door':**anaphor** anaphorTrace or 'door':**anaphor** anaphorPronoun' and to which we assign a variable of the **transformationVariable** type with the name sv1. This variable enables us to transform the terminal element of the X-bar tree that exists in its respective place.
Thus, for example,
the terminal 'door' :**anaphor** anaphorPronoun
can become 'window' :**anaphor** anaphorPronoun.

The rule we defined above had known anaphors for its terminal element. If we don't know the anaphors then the above rule becomes as follows:

transformation 'Example 9'
noVariables
structureDescription
(**node** &anyNode, **terminal** ('door':**anaphor** &anyAnaphor):
transformationVariable sv1)

In this rule apply the same as to the above rule, only that instead of the variable a1 that has as values the two anaphors, we use the variable anyAnaphor that has no values. As a result, the above rule applies to any X-bar tree that has a subtree of the X category with terminal element the word 'door' and one of more anaphors regardless of their names.

Example 10

In this example we shall define a rule that will specify a variable of the **transformationVariable** category in a terminal element of the X-bar tree. The subtree, to which this rule will apply, shall be a subtree of the X category with a terminal element that has as an anaphor the anaphorTrace or the anaphorPronoun.

Schematically, the rule will apply to the following subtree:

Any node category X

terminal (any terminal:**anaphor** anaphorTrace or anaphorPronnoun):
transformationVariable sv1

The following transformation expresses the above requirements:

transformation 'Example 10'
variables
  anaphor a1 **set** anaphorTrace **or** anaphorPronoun
structureDescription
  (**node** &anyNode, **terminal** (&anyTerm:**anaphor** &a1):
  transformationVariable sv1)

The above transformation has the name "Example 10". This transformation has a variable of the **variables** field. This variable is of the anaphor category and has the name a1 and the values anaphorTrace and anaphorPronoun. In the **structureDescription** field of the above rule we describe the subtree of the X-bar tree, to which this rule will apply. This subtree can have any node of the X category, this is why we use a variable with the name anyNode that has no values. Also, this subtree must have a terminal element that can be any element this is why we use the variable anyTerm that has no values. This terminal element however must have as an anaphor either the anaphorTrace or the anaphorPronoun. The requirement of having one of these two variables is fulfilled with the variable a1. In this terminal element we assign a variable of the **transformationVariable** type with the name sv1. This variable enables us to transform the terminal element of the X-bar tree that exists in its respective place. Thus, we can change the terminal element of the X-bar tree as we wish.

Example 11

In this example we will define a rule that will apply to X-bar trees that have the following subtree:

(**node** article **bar**, **terminal** 'the')

This rule will specify for this subtree a variable of the **transformationVariable** category with the name sv1.

Schematically, the transformation that we will define will recognize the following subtree:

node article bar

transformationVariable sv1

terminal 'the'

The transformation is the following:

transformation 'Example 11'
noVariables.
structureDescription
(**node** article **bar**, **terminal** 'the'): **transformationVariable** sv1

This transformation has the name "Example 11". It has no variables in the **variables** field. In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree is the (**node** article **bar**, **terminal** 'the'), to which we will assign a variable of the **transformationVariable** type with the name sv1. The transformation can use this variable to modify the respective subtree of the X-bar tree that uses the transformation.

Example 12

In this example we will define a rule that will apply to X-bar trees that have one of the following subtrees:

a) (**node** pron **bar**, **terminal** 'he')
b) (**node** pron **bar**, **terminal** 'she')
c) (**node** pron **bar**, **terminal** 'it')

This rule will specify for this subtree a variable of the **transformationVariable** category with the name sv1.

Schematically, the transformation that we will define will recognize one of the following subtrees:

node pron **bar**

transformationVariable sv1

terminal 'he'

node pron **bar** **transformationVariable** sv1

terminal 'she'

node pron **bar** **transformationVariable** sv1

terminal 'it'

The transformation is the following:

transformation 'Example 12'
variables
        subtree sb1 **set** (**node** pron **bar**, **terminal** 'he') **or**
                        (**node** pron **bar**, **terminal** 'she') **or**
                        (**node** pron **bar**, **terminal** 'it')
structureDescription
    subtree &sb1: **transformationVariable** sv1

The transformation has the name "Example 12" and it also has a variable of the subtree type in the **variables** field. This variable has the name sb1 and the values (**node** pron **bar**, **terminal** 'he'), (**node** pron **bar**, **terminal** 'she') and (**node** pron **bar**, **terminal** 'it'). In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree must one of these that the variable sb1 has as values. Therefore, the variable sb1 enables us to apply the above transformation in trees that have at least one of these three subtrees. Also, to the subtree of the sb1 we assign a variable of the **transformationVariable** type with the name sv1. The transformation can use this variable to modify the respective subtree of the X-bar tree that uses the transformation.

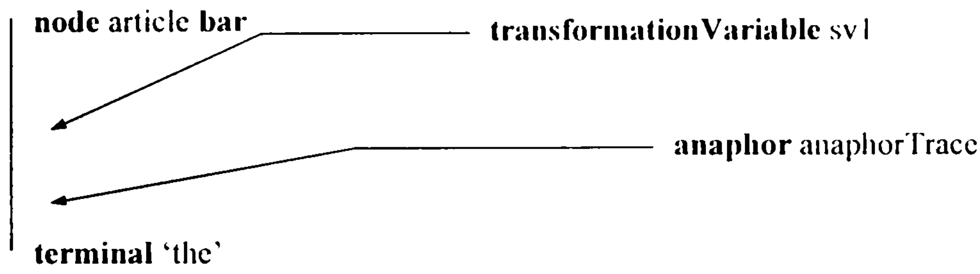
Example 13

In this example we will define a rule that will apply to X-bar trees that have the following subtree:

(**node** article **bar**, **terminal** 'the'):**anaphor** anaphorTrace

This rule will specify for this subtree a variable of the **transformationVariable** category with the name sv1.

Schematically, the transformation that we will define will recognize the following subtree:



The transformation is the following:

transformation 'Example 13'
noVariables.
structureDescription
((**node** article **bar**, **terminal** 'the'):**anaphor** anaphorTrace):
transformationVariable sv1

This transformation has the name "Example 13" and it has no variables in the **variables** field. In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree is the (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorTrace, to which we assign a variable of the **transformationVariable** category with the name sv1. The transformation can use this variable to modify the respective subtree of the X-bar tree that uses the transformation. For example, we can change the article and the anaphor name and make it (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorPronoun.

Example 14

In this example we will define a rule that will apply to X-bar trees that have one of the following subtrees:

a) (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorTrace
b) (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorPronoun

This rule will specify for this subtree a variable of the **transformationVariable** category with the name sv1.

Schematically, the transformation that we define will recognize one of the following subtrees:

node article **bar** **transformationVariable** sv1

anaphor anaphorTrace

terminal 'the'

node article **bar** **transformationVariable** sv1

anaphor anaphorPronoun

terminal 'the'

The transformation is the following:

> **transformation** 'Example 14'
> **variables**
>   **anaphor** a1 **set** anaphorTrace **or** anaphorPronoun
> **structureDescription**
>   ((**node** article **bar**, **terminal** 'the'):**anaphor** &a1):
>   **transformationVariable** sv1

The above transformation has the name "Example 14" and it has a variable of the anaphor category in the **variables** field. This variable has the name a1 and the values anaphorTrace and anaphorPronoun. This variable is used in the **structureDescription** field to describe the desired subtree. In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree is either the (**node** article **bar**, **terminal** 'the'): **anaphor** anaphorTrace or the (**node** article **bar**, **terminal** 'the'): **anaphor** anaphorPronoun to which we assign a variable of the **transformationVariable** category with the name sv1. The transformation can use this variable to modify the respective subtree of the X-bar tree that uses the transformation. For example, we can change the article and make it (**node** article **bar**, **terminal** 'a'): **anaphor** anaphorPronoun instead of the original (**node** article **bar**, **terminal** 'the'): **anaphor** anaphorPronoun.

The above rule can be modified in order to recognize the following subtrees:
(**node** article **bar**, **terminal** 'the'): **anaphor** *any anaphor*

Namely, there will be no constraint for the name of the anaphor.

Therefore, the above transformation becomes as follows:

transformation 'Example 14'
noVariables.
structureDescription
((**node** article **bar**, **terminal** 'the'):**anaphor** &anyAnaphor):
transformationVariable sv1

As we observe, we do not need the variable a1 and instead we have put the variable anyAnaphor that has no values and takes each time the value from the X-bar tree. As a result from the above description in the **structureDescription** field of the rule all the trees that have the subtree (**node** article **bar**, **terminal** 'the') are recognized, regardless of the subtree's anaphor name.

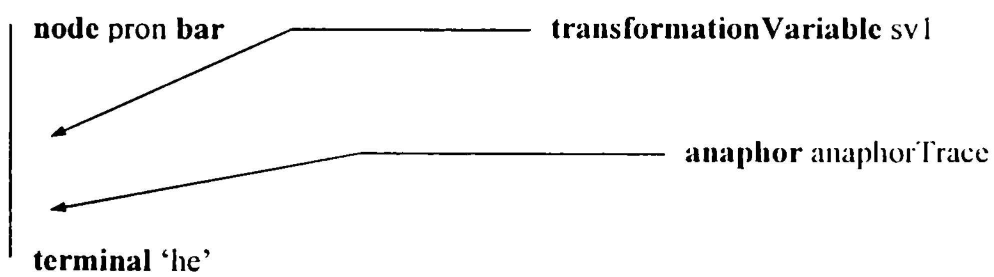
Example 15

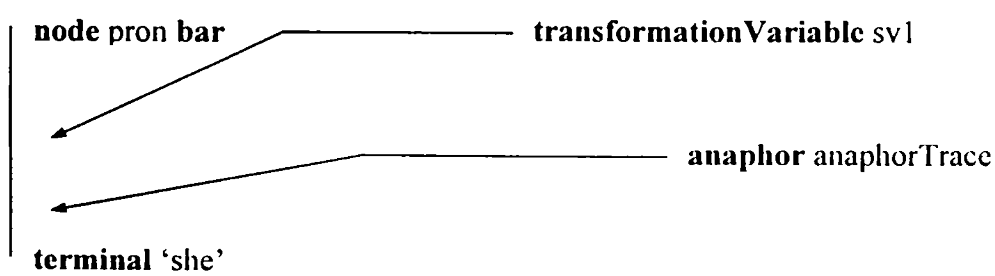
In this example we will define a rule that will apply to X-bar trees that have one of the following subtrees:

a) (**node** pron **bar**, **terminal** 'he'): **anaphor** anaphorTrace
b) (**node** pron **bar**, **terminal** 'she'): **anaphor** anaphorTrace

This rule will specify for this subtree a variable of the **transformationVariable** category with the name sv1.

Schematically, the transformation that we define will recognize one of the following subtrees:





The transformation is the following:

transformation 'Example 15'
variables
  subtree sb1 **set** (**node** pron **bar**, **terminal** 'he') **or**
      (**node** pron **bar**, **terminal** 'she')
structureDescription
  subtree (&sb1:**anaphor** anaphorTrace): **transformationVariable** sv1

This transformation has the name "Example 15" and it also has a variable of the subtree type in the **variables** field. This variable has the name sb1 and the values (**node** pron **bar**, **terminal** 'he') and (**node** pron **bar**, **terminal** 'she'). In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree must be one of these that the variable sb1 has as values. This variable is used in the **structureDescription** field to describe the desired subtree. In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree is either the (**node** pron **bar**, **terminal** 'he'): **anaphor** anaphorTrace or the (**node** pron **bar**, **terminal** 'she'): **anaphor** anaphorTrace to which we assign a variable of the **transformationVariable** category with the name sv1. The transformation can use this variable to modify the respective subtree of the X-bar tree that uses the transformation. For example, we can change the pronoun and make it (**node** pron **bar**, **terminal** 'it'): **anaphor** anaphorTrace instead of the original (**node** pron **bar**, **terminal** 'he'): **anaphor** anaphorTrace.

The above rule can be modified in order to recognize the following subtrees:

any subtree: **anaphor** anaphorTrace

Namely, there will be no constraint for the subtree but only for the name of the anaphor.

Therefore, the above transformation becomes as follows:

transformation 'Example 15'
noVariables.
structureDescription
  subtree (&anyTree:**anaphor** anaphorTrace): **transformationVariable** sv1

As we observe, we do not need the variable sb1 and instead we have put the variable anyTree that has no values and takes each time the value from the X-bar tree. As a result from the above description in the **structureDescription** field of the rule, all the trees that have the subtree with the anaphor anaphorTrace are recognized.

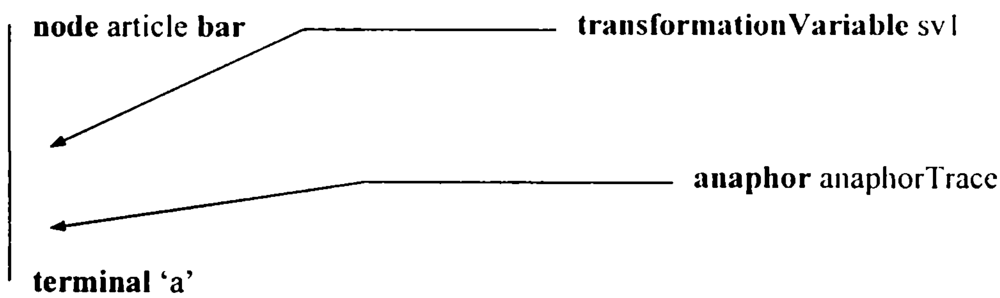
Example 16

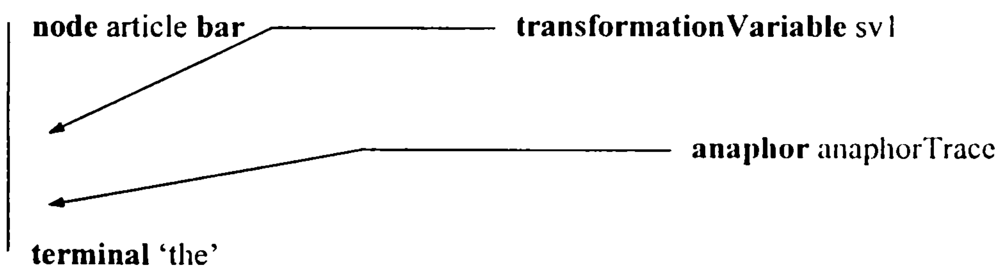
In this example we will define a rule that will apply to X-bar trees that have one of the following subtrees:

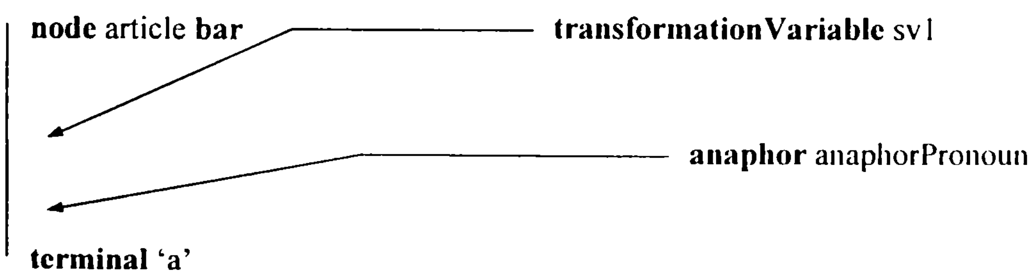
a) (**node** article **bar**, **terminal** 'a'):**anaphor** anaphorTrace
b) (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorTrace
c) (**node** article **bar**, **terminal** 'a'):**anaphor** anaphorPronoun
d) (**node** article **bar**, **terminal** 'the'):**anaphor** anaphorPronoun

This rule will specify for this subtree a variable of the **transformationVariable** category with the name sv1.

Schematically, the transformation that we define will recognize one of the following subtrees:







node article **bar** **transformationVariable** sv1

anaphor anaphorPronoun

terminal 'the'

The transformation is the following:

transformation 'Example 16'
variables
  subtree sb1 **set** (**node** article **bar**, **terminal** 'a') **or**
    (**node** article **bar**, **terminal** 'the')
  anaphor a1 **set** anaphorTrace **or** anaphorPronoun
structureDescription
  subtree (&sb1:**anaphor** &a1): **transformationVariable** sv1

This transformation has the name "Example 16" and it also has two variables in the **variables** field. The first variable has the name sb1, is of the subtree type and has the values (**node** article **bar**, **terminal** 'a') and (**node** article **bar**, **terminal** 'the'). The second variable is of the anaphor type, it has the name a1 and the values anaphorTrace and anaphorPronoun. These two variable are used in the **structureDescription** field to describe the desired subtree. In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply this rule. This subtree is the (&sb1 :**anaphor** &a1) to which we assign a variable of the **transformationVariable** category with the name sv1. The transformation can use this variable to modify the respective subtree of the X-bar tree that uses the transformation. For example, we can change the article and make it (**node** article **bar**, **terminal** 'an'): **anaphor** anaphorTrace instead of the original (**node** article **bar**, **terminal** 'the'): **anaphor** anaphorTrace.

The above rule can be modified in order to recognize subtrees that have an anaphor:

transformation 'Example 16'
noVariables.
structureDescription
  subtree (&anyTree:**anaphor** &anyAnaphor):
    transformationVariable sv1

As we observe, we do not need the variables sb1 and a1 and instead we have put the variable anyTree and the variable anyAnaphor that have no values. As a result from the above description in the **structureDescription** field of the rule, all the trees that have the subtree with an anaphor are recognized.

3.2.2.4 The tree operators in the structureDescription field

The operators for the subtrees in the **structureDescription** field of the principles and transformations are described in this chapter.

These operators belong in the following categories:

a) Operators that express the constraints between two or more subtrees that are described in the **structureDescription** field.

These operators are the following:

1) *Subtree 1* **:subtree** *Subtree 2*
2) *Subtree 1* **:notSubtree** *Subtree 2*
3) *Subtree 1* **:nodeSubtree** *node*
4) *Subtree 1* **:nodeNotSubtree** *node*

From the above operators the first one expresses the constraint that the *subtree 1* should be a subtree of the *subtree 2* subtree. The second one expresses the constraint that the *subtree 1* should not be a subtree of the *subtree 2* subtree. The third one expresses the constraint that the *subtree 1* should be a subtree of a tree that has the head node *node*. The *subtree 1* can be either at the left or at the right subtree of the subtree that has the head *node*. The forth one expresses the constraint that the *subtree 1* should not be a subtree of a tree that has the node *node*. In the first two cases, the *Subtree 1* can be a left or a right subtree of the *Subtree* 2. This is declared by the operator **subtreePosition**.

b) Operators that express the constraints that should apply to one subtree, that may contain other subtrees and operators, in the **structureDescription**.

These operators are the following:

1) **not** *subtree*
2) **aTree** *subtree*
3) **aFirstTree** *subtree*
4) **leftMost** *subtree*

From the above operators the first one states that the *subtree* subtree should not exist as a subtree of the X-bar tree in the respective place. The second operator states that the *subtree* subtree should exist as a subtree in any depth in respective place of the X-bar tree. The third operator states that the *subtree* subtree is the first subtree in any depth if the tree is scanning top-down left to right starting from the respective place of the X-bar tree. The forth operator specifies that the *subtree* is the left most subtree in any depth in the respective place of a X-bar tree if it is scanned top-down left to right.

This operator is very useful for the determination of the X-category node of an X-bar tree. The X-category node has a central role in the X-bar scheme and by using the above operators (**aTree**, **leftMost**) it is possible to determine in an easy way possible governing or c-commands relations (Chomsky, 1981, 1988, 1995) in the structures that are under processing. This can be in more general way than the Chomsky's theory by using variables (Fouskakis, 2005b) that determine possible connections between different elements of x-bar structures or their acceptable values.

c) Operators that express the constraints that should apply to N subtrees in a position of the **structureDescription** field of principles and transformations. N may be bigger or equal to 2.

These operators are the following:

1) *subtree 1* **and** *subtree 2* **and**...
2) *subtree 1* **or** *subtree 2* **or**...

From the above operators the first one states that the subtrees *subtree 1, subtree 2, etc.* should all be subtrees of the subtree that exists in the X-bar tree that uses the rule and in the respective position. While the second operator states that at least one of the subtrees *subtree 1, subtree 2, etc.* should be a subtree of the subtree that exists in the X-bar tree that uses the rule and in the respective position.

d) The operator that has no constraint for a tree and that is used in the place of the subtrees of the tree that we describe in the **structureDescription** field of principles and transformations, is the following:

1) **anyTree**

The above operator takes the place of the subtree of the tree in the **structureDescription** field of principles and transformations, *only* when it is not necessary any constraint for this subtree.

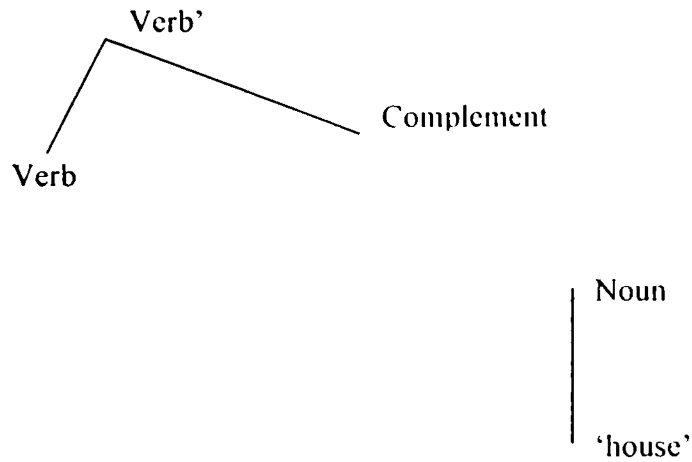
3.2.2.5 The structureDescription field examples with one operator

Next, we will analyze examples that use only one operator on subtrees of the **structureDescription** field of the **principles** and **transformations**.

Example 1

We will define a principle that will apply to those trees that have a subtree of the X category, with noun as a node name and the word 'house' as a terminal element. This subtree should be the right subtree of another subtree of the X' category with verb as a node name.

Schematically the subtree that we wish the X-bar tree to have is the following:



The principle that expresses the above requirements is the following:

principle 'Example 1'.
noVariables.
structureDescription
    (**node** noun **bar**, **terminal** 'house'):
        subtree (**node** verb **bari**, **anyTree**, **subtreePosition**)

The above principle has the name "Example 1" and has no variables in the **variables** field of stating the variables. In the **structureDescription** field we describe the subtree that the X-bar tree must have in order to apply the rule. This subtree, according to the figure we have show above and our requirements for the rule, must be (**node** verb **bari, anyTree, subtreePosition**) that has as a subtree the (**node** noun **bar, terminal** 'house') in the place of the **subtreePosition**. Namely, the (**node** noun **bar, terminal** 'house') is a subtree of the subtree that occupies the place of **subtreePosition.**

For example, the above rule could apply to the X-bar tree of the sentence.

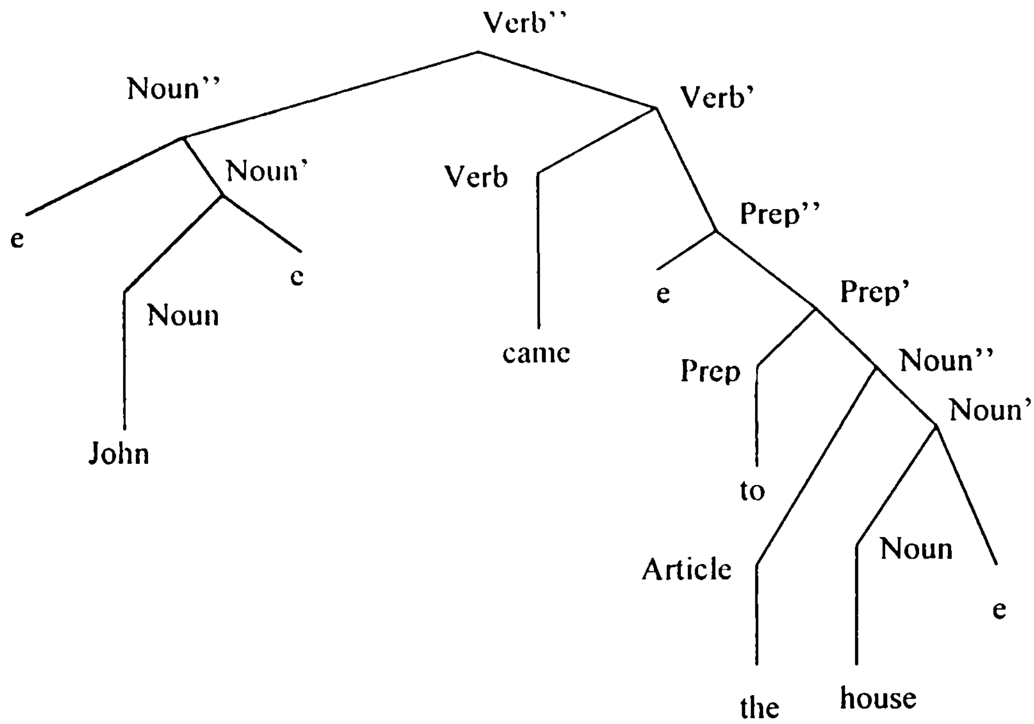
John came to the house.

but it could not apply to the sentence

The house was demolished.

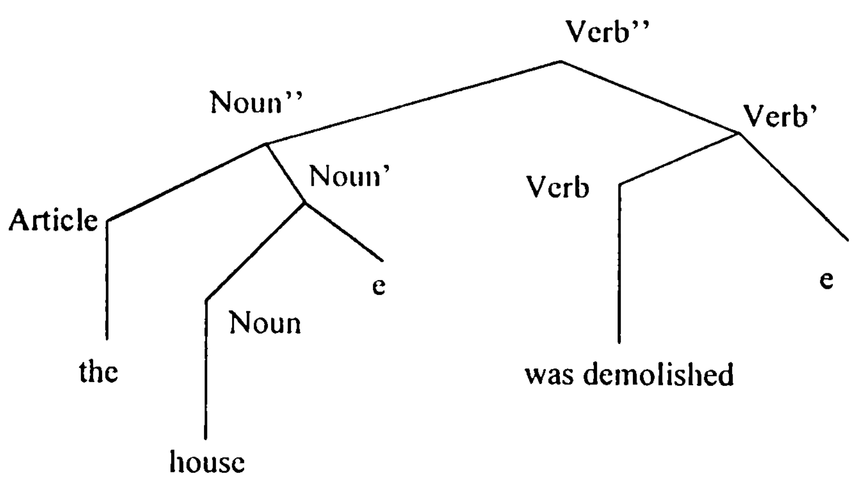
This is shown schematically by the trees that correspond to the above two sentences:

The first sentence has the tree:



Therefore the subtree recognized by the above principle is the subtree that has as a top 'Verb' and includes the subtree of the X category that has as a terminal element the word house.

The second sentence has the following tree:

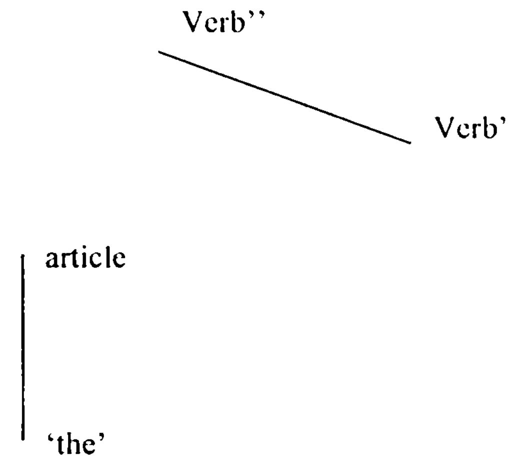


In the second sentence's tree, we observe that we have the subtree of the X category that has the node name Verb but this subtree has no right subtree. As a result the principle does not accept the above tree.

Example 2

We will define a principle that will apply to those trees that have a subtree of the X category, with article as a node name and the word 'the' as a terminal element. This subtree should not be the left subtree of another subtree of the X'' category with verb as a node name.

Schematically the subtree that we do not wish the X-bar tree to have in order to apply the rule is the following:



The principle that expresses the above requirements is the following:

principle 'Example 2'.
noVariables.
structureDescription
(**node** article **bar,terminal** 'the'):
notSubtree (node verb **barii, subtreePosition,anyTree)**

The above principle has the name "Example 2" and has no variables in the **variables** field of stating the variables. In the **structureDescription** field we describe the subtree that the X-bar tree that uses the rule should have. In the case of this rule we do not want the X-bar tree to have a subtree (**node** article **bar**, **terminal** 'the') that is the left subtree of the subtree of the X'' category with verb as a node name and description (**node** verb **barii**, **subtreePosition, anyTree**). We use the operator **anyTree** in the place of the right subtree because we are not concerned with its structure and its element. While we use **subtreePosition** in the place of the left subtree because we don't want the subtree (**node** article **bar**, **terminal** 'the') to be a subtree of the subtree that has Verb'' as a top.

For example, the above rule could apply to the X-bar tree of the sentence

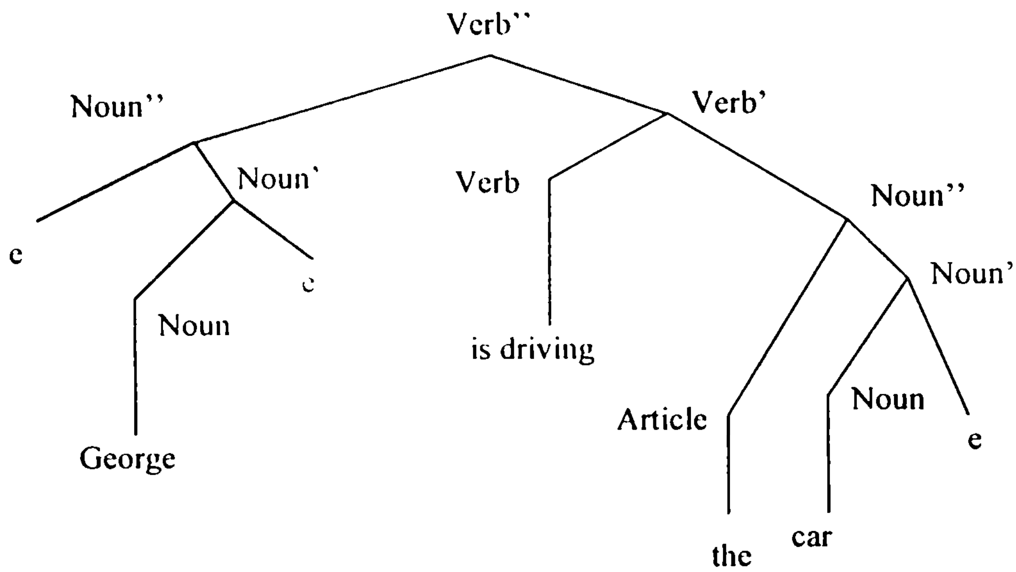
George is driving the car

but it could not apply to the sentence

The car is being repaired

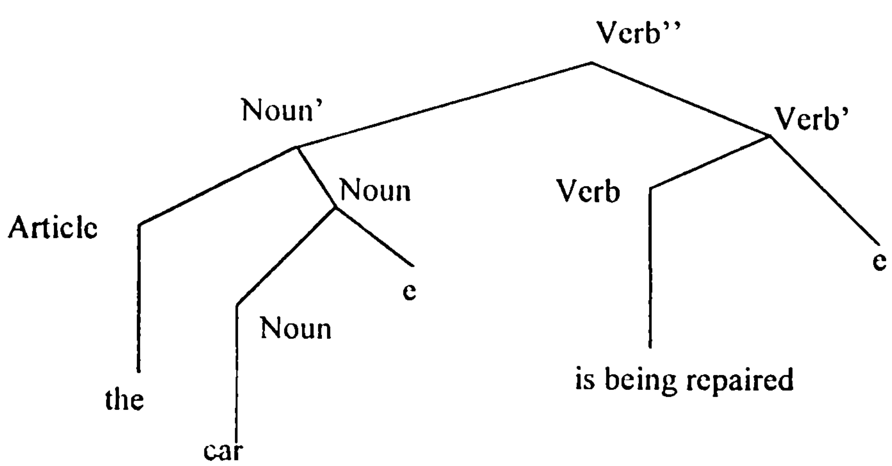
This is shown schematically by the trees that correspond to the above two sentences:

The first sentence has the tree:



Since the above subtree (**node** article **bar**, **terminal** 'the') is the right and not the left subtree of the Verb'' category tree, the specific principle can apply to this X-bar tree.

The second sentence has the following tree:

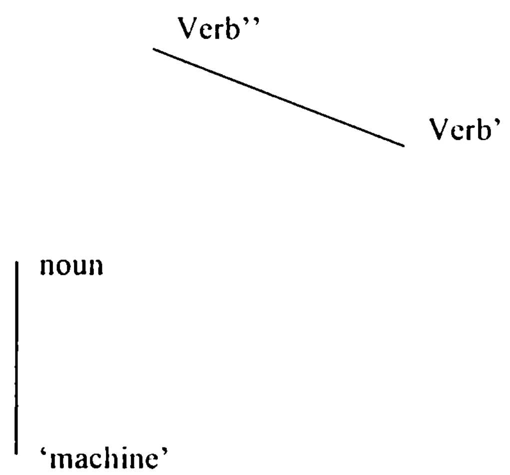


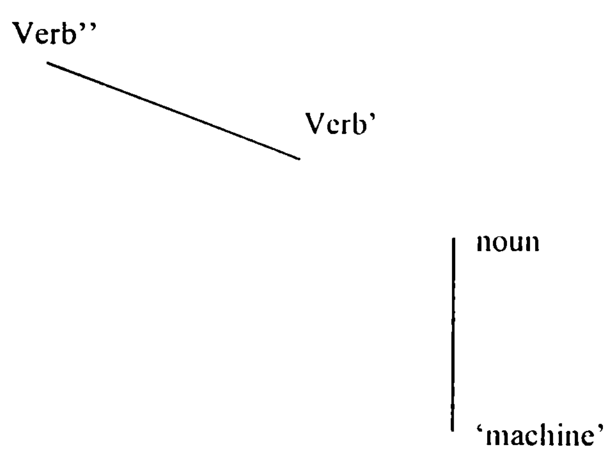
In the second sentence's tree, we observe that the subtree (**node** article **bar**, **terminal** 'the') is the left subtree of the Verb'' category tree. As a result, the subtree that we describe in the **structureDescription** field of the above principle, is not correct and therefore we cannot apply this principle.

Example 3

In this example we will define a principle that will apply to those trees that have a subtree of the X category, with noun as a node name and the word 'machine' as a terminal element. This subtree should be a subtree of another subtree that has Verb'' as a top node.

Schematically the subtree that we wish the X-bar tree to have is one of the following two:





BUPT

The principle that expresses the above requirements is the following:

principle 'Example 3'.
noVariables.
structureDescription
(**node** noun **bar**, **terminal** 'machine'):**nodeSubtree** verb **barii**

The above principle has the name "Example 3" and has no variables in the **variables** field of stating the variables. In the **structureDescription** field we describe the desired subtree that the X-bar tree that uses the rule should have. According to the above figure and our requirements for the rule, the subtree (**node** noun **bar, terminal** 'machine') should be a subtree of another subtree that has at its top the node verb **barii.**

For example, the above rule could apply as a right subtree to the X-bar tree of the sentence below:

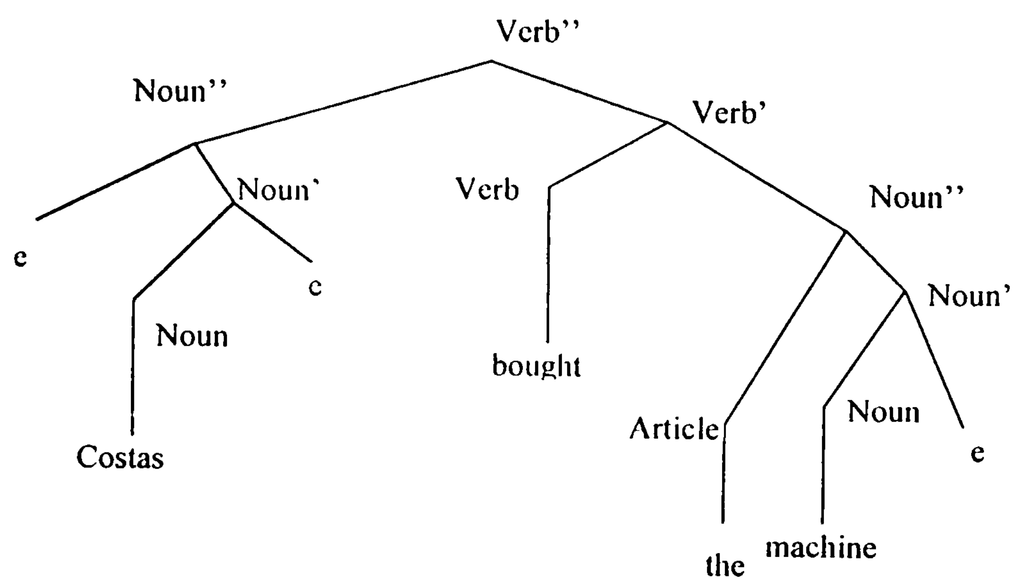
Costas bought the machine

But it could also apply as a left subtree to the sentence below:

The machine has been sold

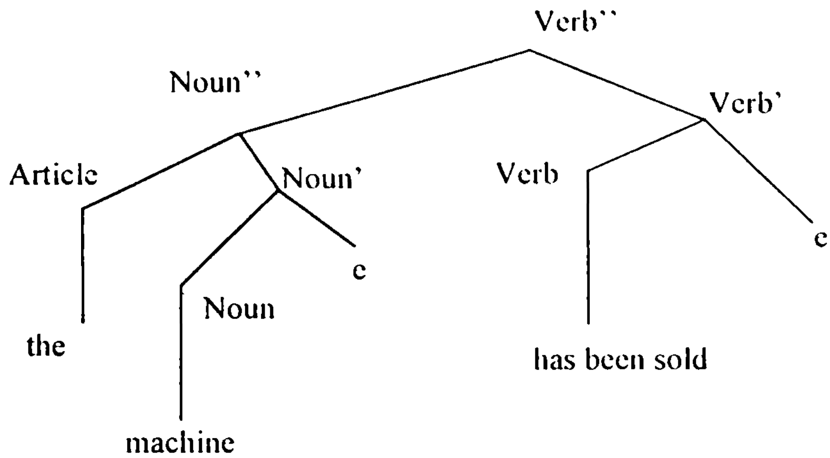
This is shown schematically by the trees that correspond to the above two sentences:

The first sentence has the tree:



Therefore, the subtree recognized by the above principle is the subtree that has Verb'' at the top and includes the subtree of the X category with the word machine as a terminal element.

The second sentence has the following tree:

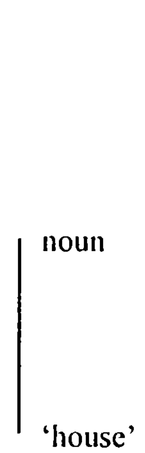


In the tree of the second sentence we observe that there is the subtree of the X category with Verb as node name. We also observe that the subtree of the X'' category with noun as node name and the word 'machine' as terminal element (**node** noun **bar, terminal** 'machine') is the left subtree of the tree that has Verb'' as a top node. As a result, this rule can also apply to the second example.

Example 4

We will define a principle that will apply to those trees that have a subtree of the X category, with noun as a node name and the word 'house' as a terminal element. This subtree should not be a subtree of another subtree that has Verb' as a top node.

Schematically the subtree that we wish the X-bar tree to have is the following:



The principle that expresses the above requirements is the following:

principle 'Example 4'.
noVariables.
structureDescription
(**node** noun **bar**,**terminal** 'house'):**notnodeSubtree** verb **bari**

The above principle has the name "Example 4" and has no variables in the **variables** field of stating the variables. In the **structureDescription** field we describe the desired subtree that the X-bar tree that uses the rule should have. In the **structureDescription** field we state that the X-bar tree should not have the subtree (**node** noun **bar**, **terminal** 'house') that is also a subtree of the tree with Verb' as the top node. That means that the subtree (**node** noun **bar**, **terminal** 'house') should be neither a left nor a right subtree of the tree with Verb' as the top node.

For example, the above rule cannot apply to the X-bar tree of the sentence below:

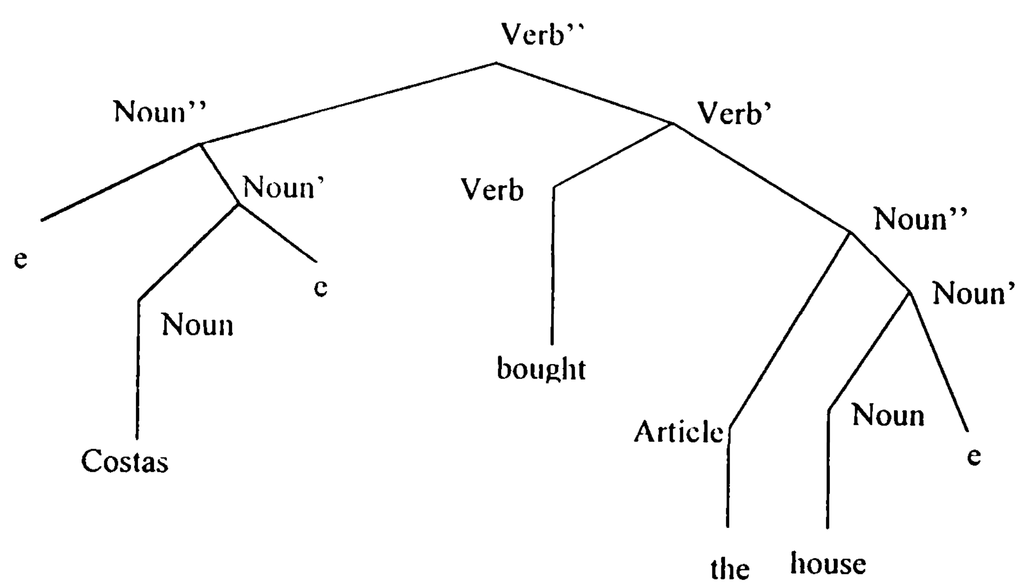
Costas bought the house

but it could apply to the following sentence:

The house has been sold

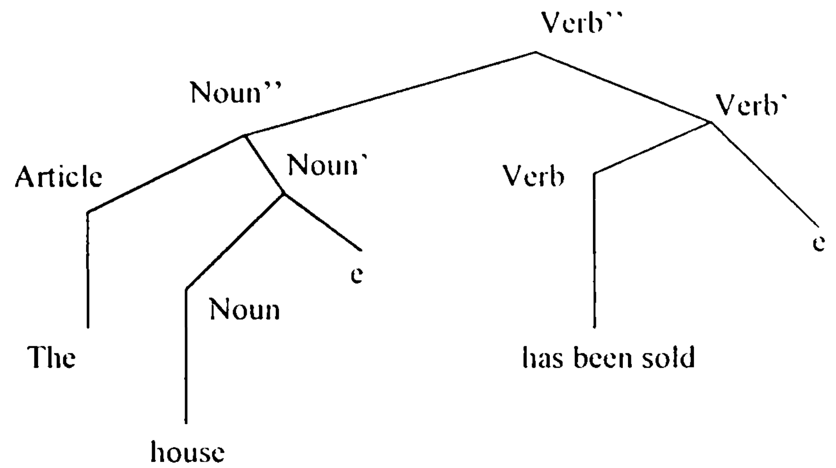
This is shown schematically by the trees that correspond to the above two sentences:

The first sentence has the tree:



This tree has the subtree (**node** noun **bar**, **terminal** 'house') that is a subtree of another subtree with Verb' as the top node. This is why the specific principle cannot apply to this sentence.

The second sentence has the following tree:

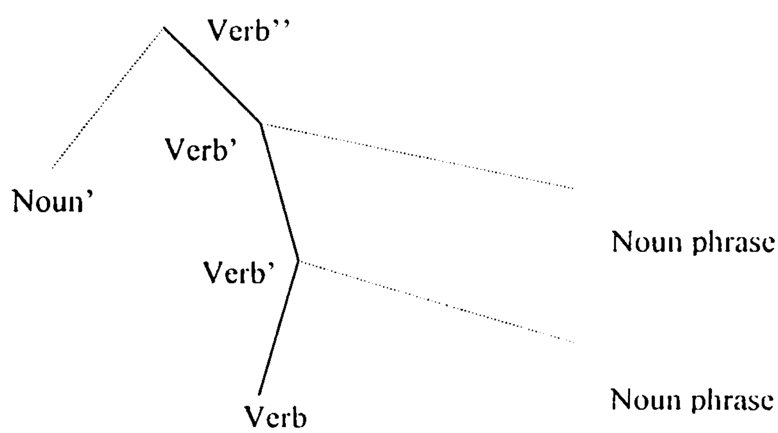


In the tree of the second sentence we observe that there is the subtree of the X' category with Verb as node name. We also observe that the subtree of the X category with noun as node name and the word 'house' as terminal element (**node** noun **bar**, **terminal** 'house') is a left subtree of the tree that has Verb'' and not Verb' as a top node. As a result, this rule can apply to this sentence.

Example 5

In this example we shall define a rule that will apply to those X-bar trees that have a verb with two objects and a noun phrase as subject. These two objects must be noun phrases.

Schematically the subtree that the X-bar tree must have is the following:



Considering our requirements about the definition of the rule and the above subtree, we define the principle as follows:

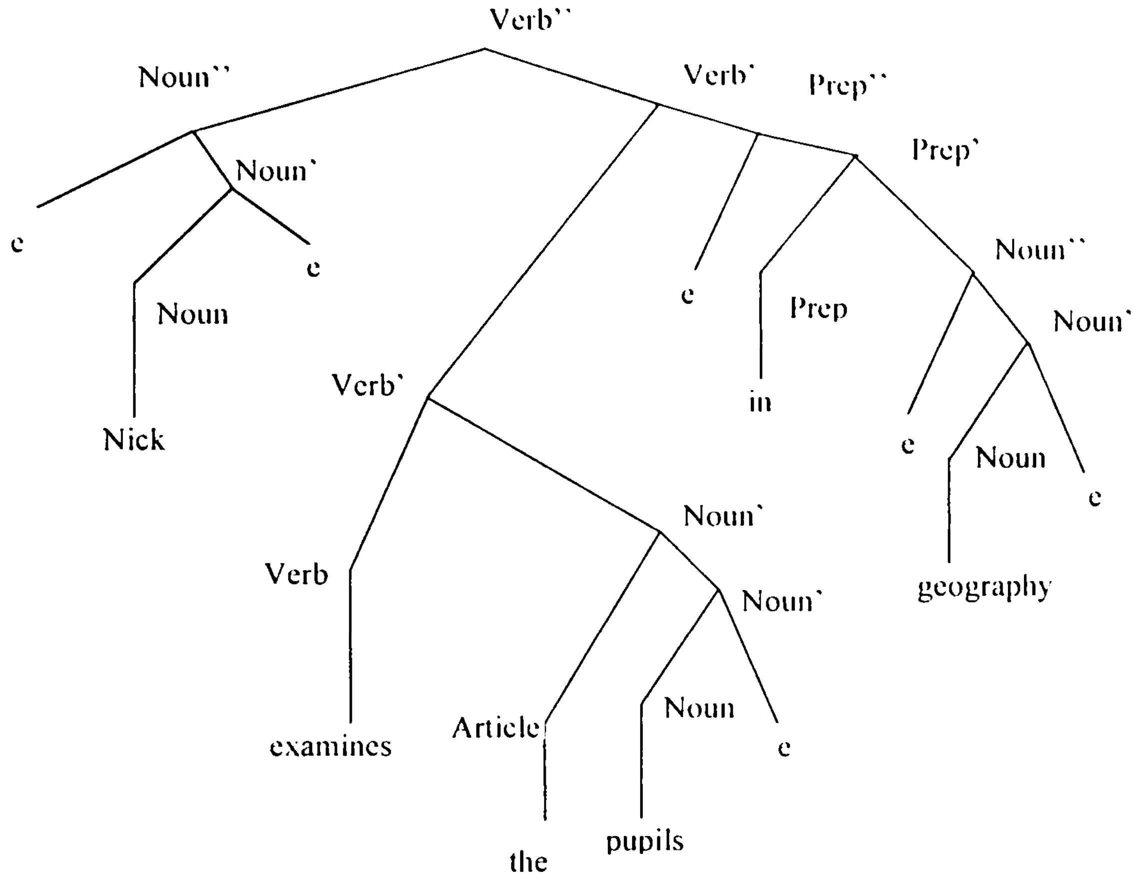
principle 'Example 5'.
noVariables.
structureDescription
(**node** verb **barii**,
leftMost (**node** noun **bari**,**anyTree**,**anyTree**),
(**node** verb **bari**,
(**node** verb **bari**,
anyTree,
aTree (**node** noun **bar**,**terminal** &anyTerminal1),
aTree (**node** noun **bar**,**terminal** &anyTerminal2)))

The principle we defined above has the name 'Example 5'. This principle has no variable in the **variables** field that's why the respective field has been replaced by the operator **noVariables**. In the **structureDescription** field we describe the structure and the elements of the subtree that the X-bar tree must have in order to apply the rule. As we describe in the **structureDescription** field, this subtree is a tree that has the top Verb''. Its left subtree is a tree of a noun phrase. This is described by the structure **leftMost** (**node** noun **bari**,**anyTree**,**anyTree**) with its left and right subtrees out of our interest. This is why we use the operator **anyTree**. The right subtree is a tree that has the top Verb' and the right subtree that we want to have a subtree of the X category with noun as a node name. This is why we use **aTree** (**node** noun **bar**, **terminal** &anyTerminal2). Its left subtree is a tree of the Verb' category. It has as right subtree a tree that includes the subtree of the X category with noun as a node name. This is why we use **aTree** (**node** noun **bar**, **terminal** &anyTerminal2).

Next we shall give an example of a sentence to which the above principle applies:

Nick examines the pupils in geography.

The tree of this sentence is the following:

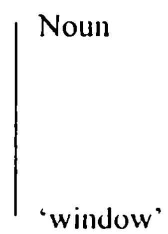


We notice that the tree of the above sentence includes a subtree that has the top Verb'' and the right subtree Verb'. This subtree has as a right subtree the one that includes the subtree (**node** noun **bar**, **terminal** 'geography'). The subtree of Verb' has a left subtree with the Verb' as a top and a right subtree that includes the subtree (**node** noun **bar, terminal** 'pupils'). Also there is the noun phrase with root node Noun', this is the (**node** noun **bari**,(**node** noun **bar**, **terminal** 'Nick'), **empty**).

Example 6

In this example we shall define a rule that will recognize those trees that do not have a subtree of the X category with noun as a node name and the word 'window' as a terminal element.

Schematically the subtree that should not have the tree that uses the rule is the following:



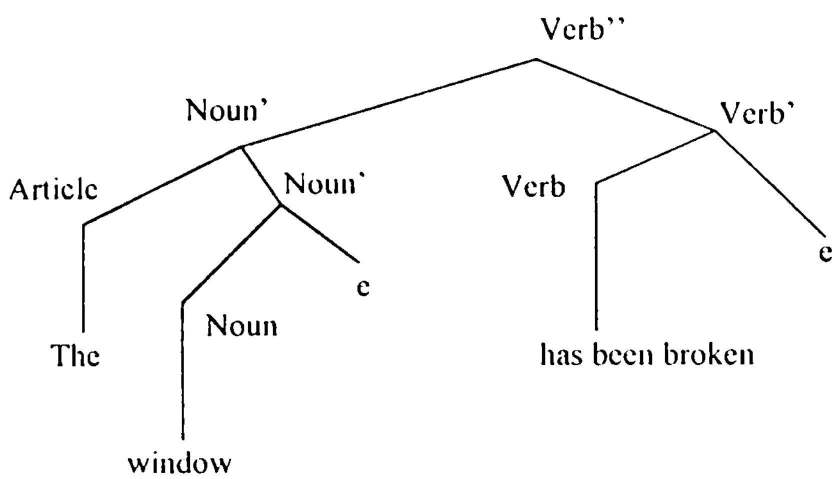
The principle that is in accordance to the above is the following:

principle 'Example 6'.
noVariables.
structureDescription
not (**node** noun **bar**, **terminal** 'window')

This principle has the name 'Example 6' and has no variables this is why it has the name **noVariables**. In the **structureDescription** field we describe the subtree that the X-bar tree that uses the rule should have. In our case, we don't want the X-bar tree to have the subtree (**node** noun **bar, terminal** 'window').

An example of a sentence that cannot use the above rule is the following:

The window has been broken



The above sentence has a subtree of the X category with noun as a node name. This subtree is the (**node** noun **bar, terminal** 'window'). This is why the principle 'Example 6' cannot apply to the above tree.

Example 7

In this example we want to define a rule that will apply to those trees that have two nouns, one in nominative case and the other in the accusative.

According to the above, the two subtrees that the tree that uses the rule should have are the following:

a) The subtree in the nominative case

Noun' with feature +nominative

Terminal element

b) The subtree in the accusative case

Noun' with feature +accusative

Terminal element

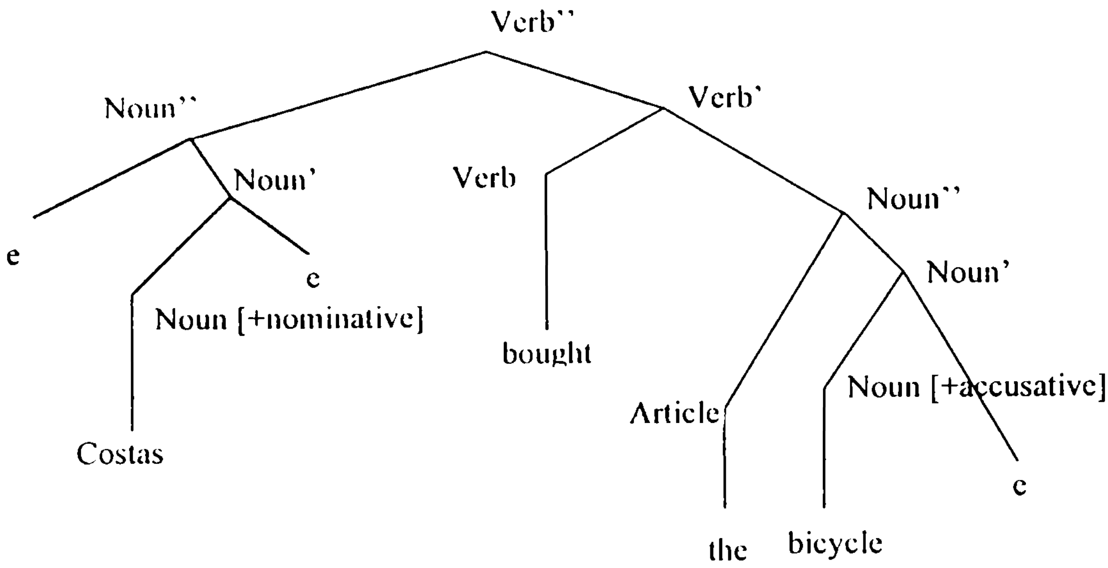
Therefore, the principle that fulfils the above requirements is the following:

principle 'Example 7'.
noVariables.
structureDescription
(**node** noun **bar**:**features** [+nominative], **terminal** &anyTerminal1)
and
(**node** noun **bar**:**features** [+accusative], **terminal** &anyTerminal2)

This rule has the name 'Example 7" and it has no variables in the **variables** field that's why we use the operator **noVariables**. In the **structureDescription** field we describe the subtree that the X-bar tree should have in order to apply on it this rule. We notice that in the **structureDescription** field we describe two subtrees that are connected with the operator **and**. This means that the X-bar tree should have both subtrees as its own subtrees in order to apply the rule on it. The sequence of these two subtrees in the **structureDescription** field is irrelevant. The first of these two subtrees is the (**node** noun **bar** : **features** [+nominative], **terminal** &anyTerminal1). It's a subtree of the X category with noun as a node name and [+nominative] as a node feature. We are not concerned with the terminal element that follows, this is why we use the variable anyTerminal1 that has no values. The second subtree is the (**node** noun **bar** : **features** [+accusative], **terminal** &anyTerminal2). It's a subtree of the X category with noun as a node name and [+accusative] as a node feature. We are not concerned with the terminal element of this subtree, this is why we use the variable anyTerminal2 that has no values. We also notice that we do not use the same variable name for the terminal of both subtrees because we do not wish them to have the same terminal element.

An example of a sentence that fulfills the requirements of the above rule is the following:

Costas bought the bicycle



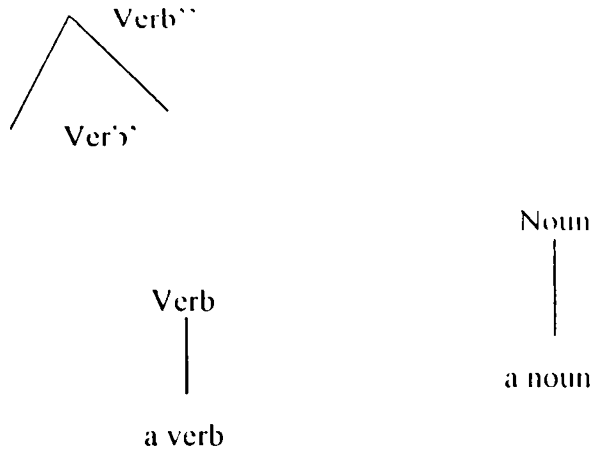
We notice that the tree of the above sentence has a subtree of the X category with Noun [+nominative] as a node and the word Costas as a terminal element. Also, the above tree has another subtree of the X category with Noun [+accusative] as a node and the word bicycle as a terminal element. This is why the above principle can apply to this tree.

Another example of using the operator **and** is the following:

We wish to define a principle that will recognize those trees that have a subtree of the X'' category with Verb as a node name. This subtree should have subtrees that would include the following two subtrees:

a) the one subtree will be of the X category with Verb as name and a verb as terminal element.
b) The other subtree will be of the X category with Noun as name and a noun as terminal element.

Schematically the subtree that the X-bar tree must have in order to apply this rule is the following:



Therefore, according to the above, we have the following principle:

principle 'Example 7'.
noVariables.
structureDescription
   (**node** verb **barii,anyTree,**
      (**node** verb **bar,terminal** **&**anyTerminal1)
      and
      (**node** noun **bar,terminal** **&**anyTerminal2))

The above principle has no variables in the **variables** field that's why we use the operator **noVariables.** In the **structureDescription** field we describe the structure and the elements of the subtree that the X-bar tree should have in order to apply on it this rule. The subtree of the **structureDescription** field is the same with the one shown schematically above. Thus, in the **structureDescription** field we describe a subtree that has the Verb'' as top and a left subtree with structure and elements not of our concern. This is why we use the operator **anyTree**. We want, however, the right subtree to include the following two subtrees:

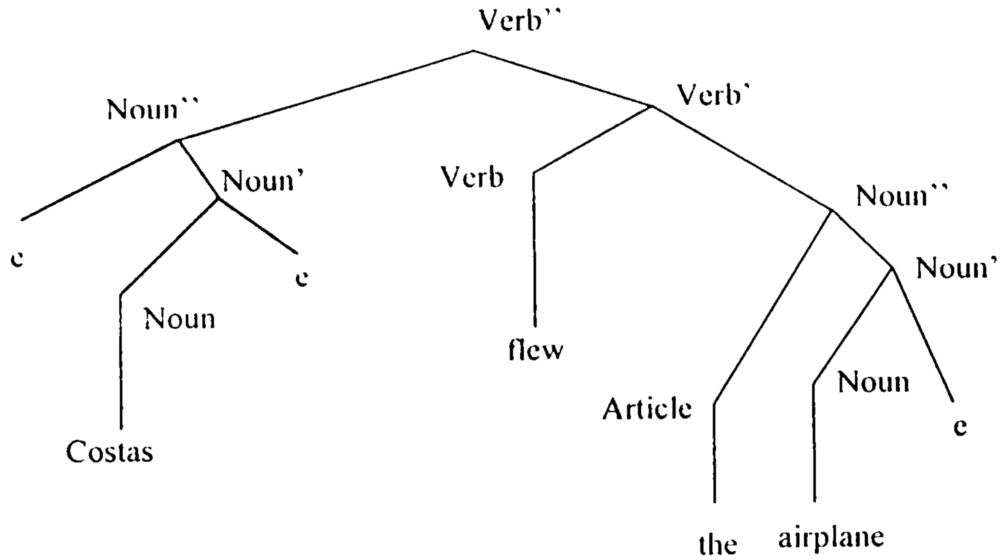
a) (**node** verb **bar**, **terminal** **&**anyTerminal1)
b) (**node** noun **bar**, **terminal** **&**anyTerminal2))

These two subtrees are of the X category. The first subtree has the node name verb and any terminal element. The second one has the node name noun and any terminal element.

An example of a sentence to which the above rule could apply is the following

Costas flew the airplane.

This sentence has the following tree:



We notice that this tree has the subtree that we described schematically above and the one that the principle that we defined demands. This tree has the Verb'' as top and the right subtree that includes the two subtrees that the rule requires.

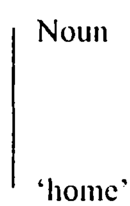
These subtrees are the following:

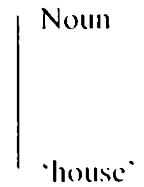
Example 8

In this example we shall define a rule that will apply to those X-bar trees that have one of the following subtrees of the X category:

a) a subtree with noun as a node name and the word 'home' as a terminal element
b) a subtree with noun as a node name and the word 'house' as a terminal element

Schematically, these subtrees are the following:





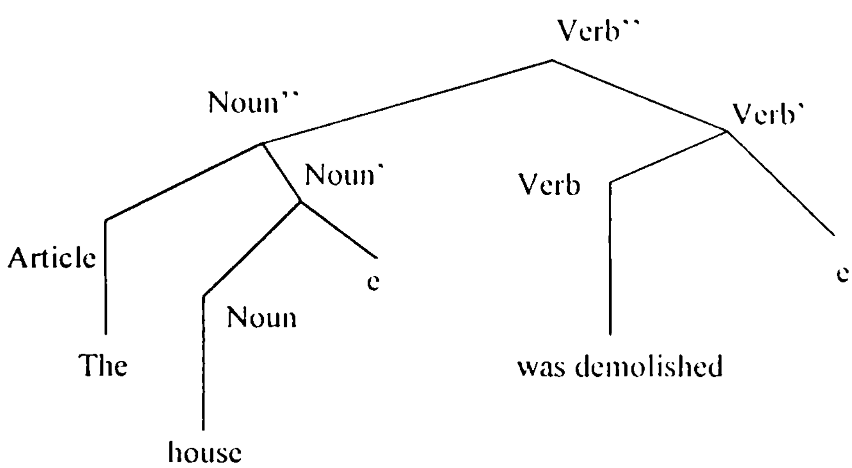
The principle that describes the above is the following:

principle 'Example 8'.
noVariables.
structureDescription
(**node** noun **bar**,**terminal** 'home')
or
(**node** noun **bar**,**terminal** 'house')

This principle has the name 'Example 8" and it has no variables that's why we use the operator **noVariables**. In the **structureDescription** field we describe the subtree that we wish the X-bar tree to have. We wish the tree to have at least one of the (**node** noun **bar, terminal** 'home') and (**node** noun **bar, terminal** 'house'), this is why we use the operator **or**. Both the subtrees are of the X category. The first one has the noun **bar** as a node and the word 'home' as a terminal element. The second element has the noun **bar** as a node and the word 'house' as a terminal element.

An example of a sentence that the subtree we describe in the **structureDescription** field fulfils is the following:

The house was demolished



We notice that this tree has the subtree that the principle 'Example 8' requires. This subtree is the following:

Noun

house

3.2.2.6 The structureDescription field examples with more than one operator

In all the above examples we used operators that could apply to subtrees in the **structureDescription** field of principles and transformations. Each time we have used only one operator but we could apply more than one operator to a subtree.

Next we shall analyze examples that use more than one operators to subtrees of the **structureDescription** field.

Example 1

In this example we shall define a rule that will apply to those X-bar trees that have a subtree of the X category with noun as a node name. This subtree of the X category is the subtree of a subtree that has the Noun' as top and of another subtree that has Verb' as top.

Schematically the subtree that we wish the X-bar tree to have is the following:

Verb' Noun'

Noun

a noun

The principle that is in accordance to the above is the following:

```
principle 'Example 1'.
noVariables.
structureDescription
   (node noun bar,terminal &anyTerminal):
      (nodeSubtree verb bari):(nodeSubtree noun bari)
```

This principle has the name 'Example 1'. It has no variables in the **variables** field and this is why we use the operator **noVariables**. In the **structureDescription** field we describe the subtree that the X-bar tree that uses the rule should have. The subtree that we wish the X-bar tree to have is a tree of the X category that has the noun as a node name and any terminal element. This is why we use the variable anyTerminal that has no values. This subtree is the subtree **(node** noun **bar, terminal &**anyTerminal**)**. There are however two constraints for this subtree. The first constraints is that it should be a subtree of the subtree that has the Verb' as top. This constraint is expressed with the **(nodeSubtree** verb **bari)**. The second constraint is that the above subtree should be a subtree of the subtree that has the Noun' as top. This constraint is expressed with the **(nodeSubtree** noun **bari)**.

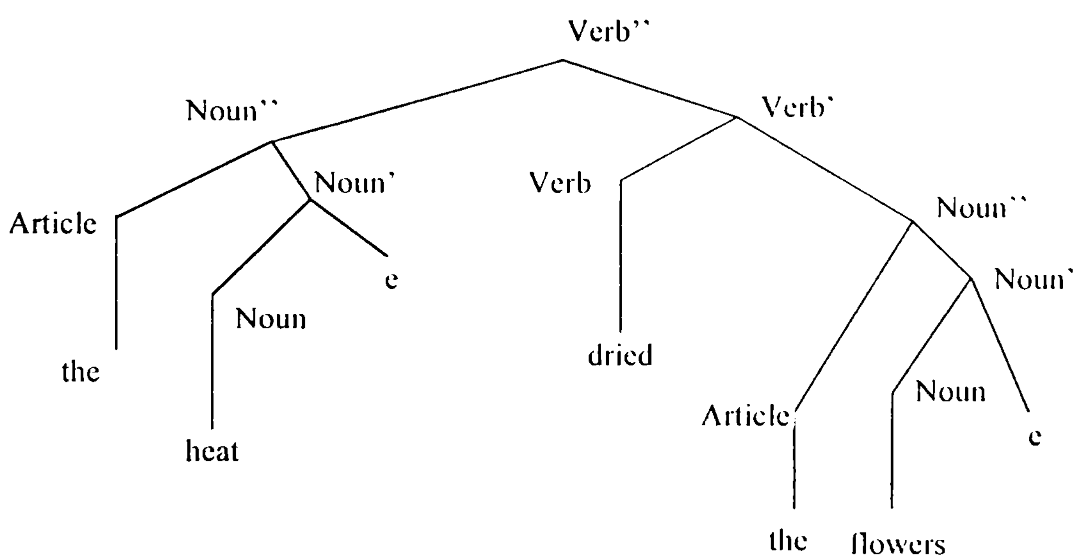
An example of a sentence to which the above rule could apply is the following:

The heat dried the flowers

On the contrary, it cannot apply to the following sentence:

The flowers were dried

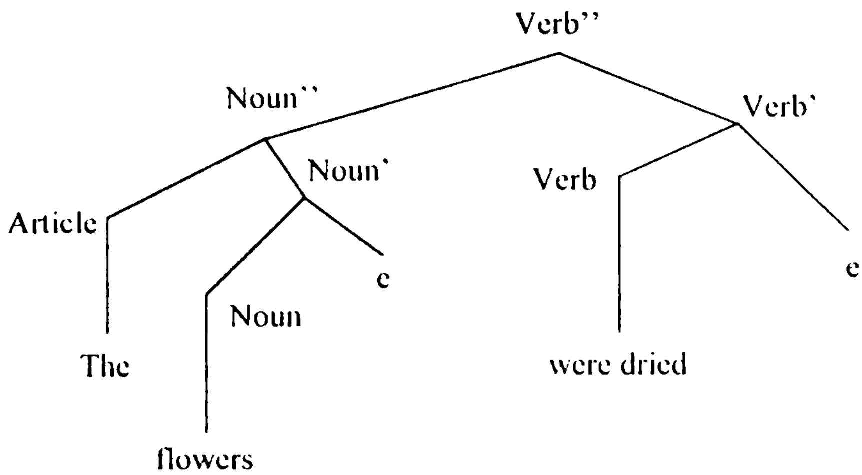
The tree of the first sentence is the following:



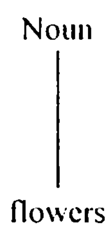
We notice that the above tree has two subtrees of the X category with the Noun as node.

These subtrees are the following:

However, only the right one is a subtree of the subtrees that have the Verb' and the Noun' as top.
The second sentence has the following subtree:



We notice that the above tree has a subtree of the X category with the Noun as node name.
This tree is the following:

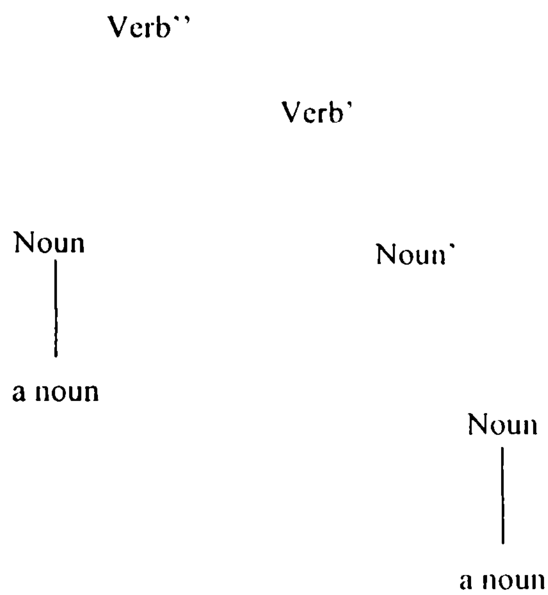


This subtree, however, is not a subtree of the subtrees that have as a top the Verb'.

Example 2

In this example we shall define a rule that will apply to those X-bar trees that have two subtrees of the X category with noun as a node name. The first subtree of the X category should be the subtree of a subtree that has the Noun' as top and of another subtree that has Verb' as top. The second subtree should be the subtree of a tree that has the Verb'' as a top.

Schematically the subtrees that the X-bar tree should have is the following:



According to the above, we have the following principle:

principle 'Example 2'.
noVariables.
structureDescription
    atree ((**node** noun **bar**,**terminal** &anyTerminal1):
        (**nodeSubtree** verb **bari**):(**nodeSubtree** noun **bari**))
    and
    atree ((**node** noun **bar**,**terminal** &anyTerminal2):
        (**nodeSubtree** verb **barii**))

This principle has the name 'Example 2'. It has no variables in the **variables** field of the rule. In the **structureDescription** field we describe the subtree that the X-bar tree must have. As we notice, we want the X-bar tree to have the two subtrees **(node** noun **bar, terminal** &anyTerminal1) and **(node** noun **bar, terminal** &anyTerminal2). There are however constraints for these subtrees. The first should be a subtree of the subtrees that has the Verb' and Noun' as tops. This is expressed with the **(nodeSubtree** verb **bari)** and **(nodeSubtree** verb **bari)** respectively. The second should be a subtree of the subtree that has the Verb'' as top. This is expressed with the **(nodeSubtree** verb **barii)**.

Let us consider as examples of sentences for the above principle the same examples that we used in the previous example:

a) The heat dried the flowers
b) The flowers were dried

From these sentences and according to their trees that are shown in the example 1, we notice that the above principle can apply only to the first sentence. The reason is that the above sentence includes the two subtrees that the principle of the example 2 requires and that they fulfil the constraints of this principle.
These two subtrees are the following:

And we notice that the first one is a subtree of the subtree that has the Verb'' at the top and the second one is the subtree of the subtrees that have the Verb' and Noun' at their top.

3.2.2.7 The EBNF of the structureDescription field

The EBNF form of the **structureDescription** field of principles and transformations is the following:

sd-subtree = "(" (sd-subtree-χ'' | sd-subtree-χ | sd-subtree-χ) ")" "." .

sd-subtree-χ'' = "(" sd-node-χ'' "," sd-specifier ","
(sd-subtree-χ'' | sd-subtree-χ') ")"
[":" sd-anaphors] .

sd-subtree-χ' = "(" sd-node-χ' "," (sd-subtree-χ' | sd-subtree-χ) "," sd-subtree-χ'' ")" [":" sd-anaphors].

sd-subtree-χ = "(" sd-node-χ "," sd-terminal ")" [":" sd-anaphors].

sd-subtree-χ'' = "(" sd-subtree-χ'' ")"
["":" "transformationVariable" sd-variable-name].

sd-subtree-χ' = "(" sd-subtree-χ' ")"
["":" "transformationVariable" sd-variable-name].

sd-subtree-χ = "(" sd-subtree-χ ")"
["":" "transformationVariable" sd-variable-name].

sd-subtree-χ'' = sd-subtree-vars.

sd-subtree-χ' = sd-subtree-vars.

sd-subtree-χ = sd-subtree-vars.

sd-subtree-χ'' = sd-subtree-subtree.

sd-subtree-χ'' = sd-subtree-notSubtree.

sd-subtree-χ'' = sd-subtree-nodeSubtree.

sd-subtree-χ'' = sd-subtree-notnodeSubtree.

sd-subtree-χ' = sd-subtree-subtree.

sd-subtree-χ' = sd-subtree-notSubtree.

sd-subtree-χ' = sd-subtree-nodeSubtree.

sd-subtree-χ' = sd-subtree-notnodeSubtree.

sd-subtree-χ = sd-subtree-subtree.

sd-subtree-χ = sd-subtree-notSubtree.

sd-subtree-χ = sd-subtree-nodeSubtree.

sd-subtree-χ = sd-subtree-notnodeSubtree.

sd-subtree-χ'' = "not" sd-subtree.

sd-subtree-χ' = "not" sd-subtree.

sd-subtree-χ = "not" sd-subtree.

sd-subtree-χ'' = "aTree" sd-subtree.

sd-subtree-χ' = "aTree" sd-subtree.

sd-subtree-χ = "aTree" sd-subtree.

sd-subtree-χ'' = "aFirstTree" sd-subtree.

sd-subtree-χ' = "aFirstTree" sd-subtree.

sd-subtree-χ = "aFirstTree" sd-subtree.

sd-subtree-χ'' = "leftMost" sd-subtree.

sd-subtree-χ' = "leftMost" sd-subtree.

sd-subtree-χ = "leftMost" sd-subtree.

sd-subtree-χ'' = "(" sd-subtree operator sd-subtree ")".

sd-subtree-χ' = "(" sd-subtree operator sd-subtree ")".

sd-subtree-χ = "(" sd-subtree operator sd-subtree ")".

sd-subtree-χ'' = anyTree.

sd-subtree-χ' = anyTree.

sd-subtree-χ = anyTree.

sd-specifier = sd-subtree-χ'' | sd-subtree-χ.

sd-anaphors = subtree-terminal-variable-anaphors.

(Note: The subtree-terminal-variable-anaphors and the terminal-variable-name, node-variable-name, subtree-variable-name, tree-terminal-value, features-variable-name, node-features-value and anaphor-variable-name are defined at the **variables** definition chapter)

sd-terminal = ("terminal" tree-terminal-value |
"terminal" "&"terminal-variable-name ":"
"anaphor" "&"anaphor-variable-name)
[":" "transformationVariable" sd-variable-name].

sd-node-χ'' = ("node" node-name "barii" [":" sd-node-features] |
sd-node-vars) [":" "transformationVariable" sd-variable-name].

sd-node-χ' = ("node" node-name "bari" [":" sd-node-features] |
sd-node-vars) [":" "transformationVariable" sd-variable-name].

sd-node-χ = ("node" node-name "bar" [":" sd-node-features] |
sd-node-vars) [":" "transformationVariable" sd-variable-name].

(Note: The <u>node-name</u> is defined at the structures definition chapter)

sd-node-vars = "node" "&" node-variable-name|
"node" "&" node-variable-name":"
"features" "&" features-variable-name.

sd-node-features = "features" (node-features-value |
("&"features-variable-name)).

sd-subtree-vars = "subtree" "&"subtree-variable-name |
"subtree" "&"subtree-variable-name ":"
"anaphor" "&" anaphor-variable-name |
"subtree" (sd-subtree-χ''| sd-subtree-χ'| sd-subtree-χ)
":" "anaphor" "&" anaphor-variable-name.

sd-subtree-subtree = (sd-subtree-χ' | sd-subtree-χ) ":" "subtree"
(sd-subtree-second-subtree-χ'' |
sd-subtree-second-subtree-χ').

sd-subtree-notSubtree =
(sd-subtree-χ'' | sd-subtree-χ' | sd-subtree-χ)
":" "notSubtree"
(sd-subtree-second-subtree-χ'' |
sd-subtree-second-subtree-χ').

sd-subtree-second-subtree-χ'' =
"(" sd-node-χ'' ","
(sd-specifier |
sd-subtree-second-subtree-χ'' | "subtreePosition") ","
(sd-subtree-second-subtree-χ'' |
sd-subtree-second-subtree-χ' | "subtreePosition") ")"
[":" sd-anaphors].

sd-subtree-second-subtree-χ' =
"(" sd-node-χ' ","
(sd-subtree-second-subtree-χ' |
sd-subtree-χ | "subtreePosition") ","
sd-subtree-second-subtree-χ'' ")"
[":" sd-anaphors].

sd-subtree-nodeSubtree =
(sd-subtree-χ'' | sd-subtree-χ' | sd-subtree-χ)
":" "nodeSubtree"
(sd-node-χ'' | sd-node-χ' | sd-node-χ).

sd-subtree-notnodeSubtree =
(sd-subtree-χ'' | sd-subtree-χ' | sd-subtree-χ)
":" "notnodeSubtree"
(sd-node-χ'' | sd-node-χ' | sd-node-χ).

operator = "and" | "or".

3.2.3 The structureCommands field of the principles and transformations

As it was mentioned in a previous section, both principles and transformations have three different fields.

These fields are the following:

- **variables**
- **structureDescription**
- **structureCommands**

In the **structureCommands** field it is possible to describe the checks, to change the variables values, to declare variables and transformations if the rule is of transformation type or to execute commands directly (e.g. in a man-machine interface software system). These abilities are described in the following chapters.

3.2.3.1 Declaration of variables in the structureCommands field

In the **structureCommands** field of principles and transformations we can define new variables. The variables that we can define are variables of the **variables** field category. These variables enable us to describe the functions of the **structureCommands** field of principles and transformations.

The ways of stating the new variables are the following:

1. *variable type operator* *variable name* **set** *variable values*
2. **features** *name of variable* **set** *tree node*
3. **anaphor** *name of variable* **set** *terminal*
4. **anaphor** *name of variable* **set** *subtree*
5. **subtree** *name of variable* **set** **nextStructure [(*Num*)]**
6. **subtree** *name of variable* **set** **previousStructure [(*Num*)]**
7. **subtree** *name of variable* **set** **particularStructure (*Num*)**

In all the above cases, it is noticed that the definition of a new variable requires a name. Everything regarding the variables of the **variables** field is applied for the name of this variable. Also, the name of each new variable in the **structureCommands** field of principles and transformations must not be the same with the one of the variables of the **variables, structureDescription** and **structureCommands** fields.

From the above cases for declaration of variables in the **structureCommands** field, the first one is the general way of stating variables the same as in the **variables** field of principles and transformations. The italic letters are elements that can change according to the case. Thus, the *variable type operator* can be one of the following depending on the type of the variable:

1. tree node operator : **node**
2. terminal element operator : **terminal**
3. anaphor operator : **anaphor**
4. node features operator : **features**
5. subtree operator : **subtree**

The *variable values* are the values that are given to the variable. The method that gives values to the new variable is the same with the one that is used for the variables of the **variables** field of principles and transformations.

The second case is to state variable of the features type. In this case, the difference is that the values that this new variable takes are specified by the features of the *tree node*. It can be a tree node with its features, a variable of the node type that has been stated or can use a variable that has already been stated.

The third case is about stating variables of the anaphor type. The values that the new variable will be have are anaphors of the terminal that follows the operator **set**. It can be a terminal with its anaphors, a variable of the terminal element type that has already been stated or a terminal element that uses another variable that has already been stated.

The forth case is for stating variables of the anaphor type. The values that the new variable will have are the anaphors of the subtree that follows the operator **set**. It can be a whole subtree with its anaphors, a subtree that uses variables or a variable of the subtree category.

The fifth case defines a new variable of type **subtree** which contains the next **structure** of the X-bar trees of the linguistic system. The sixth case defines a new variable of type **subtree** which contains the previous **structure** of the X-bar trees of the linguistic system. In these two cases it is possible optionally to select an n-th previous or next tree. The seventh case defines a new variable of type **subtree** which contains a particular **structure** (according with the number that we use as parameter) of the X-bar trees of the linguistic system. These cases are useful if we want to move at different X-bar trees of the linguistic system.

The variables of the **transformationVariable** category in the **structureDescription** field of principles and transformations can be used to the above cases of stating new variables, like all the other variables of the **variables** category.

Next, we shall analyze examples of stating new variables according to the above cases:

1) The first case is the general way of stating variables:

a) **node** n1 **set** article **bar** : features [+nominative, +masculine] **or** noun **bari**
In this example we define a new variable of the tree node type that has the name n1 and also has as values the nodes article **bar** : features [+nominative, +masculine] and noun **bari**.

b) **terminal** t1 **set** a **or** the
In this example we define a new variable of the terminal element type that has the name t1 and it also has as values the words 'a' and 'the'.

c) **anaphor** a1 **set** i1 **or** j1 **or** k1
In this example we define a new variable of the anaphor type that has the name a1 and it also has the values i1, j1, k1.

d) **terminal** t2 **set** the: anaphor &a1
In this example we define a new variable of the terminal element type that has the name t2 and it also has as values the terminal elements that derive from the word 'the'. It also has the anaphors i1, j1, k1.

e) **features** f1 **set** [+singular, +human] **or** [+plural, +adjective]
In this example we define a new variable of the node features type that has the name f1 and it also has the values [+singular, +human] and [+plural, +adjective].

f) **node** n2 **set** noun bar: features &f1 **or** noun bari : features &f1
In this example we define a new variable of the tree node type that has the name n2 and it also has as values the nodes that derive from the node noun **bar** and the node noun **bari**, to which we add the features [+singular, +human] and [+plural, +adjective].

g) **subtree** s1 **set (node** &n2, **terminal** person) : **anaphor** &a1
In this example we define a new variable of the subtree type that has the name s1 and it also has the value (node &n2, terminal person) : anaphor &a2, where the node is replaced by the variable n2 and the anaphors are replaced by the variable a1.

2) The second case is about the statement of new variables of the node features type.

a) **features** f1 **set** noun **bar : features** [+nominative,+singular]
In this example the node we are using is fully described without the use of variables. Therefore, the value of the variable f1 is [+nominative, +singular].

b) **features** f1 **set** noun **bari : features** &nf1
In this example the node that exists on the right of the operator **set** uses the variable nf1 for its features. As a result, the values for the new variable f1 will be the values of the variable nf1 that we use to describe the node's features. We must stress that the variable nf1 must be already stated, either in the **variables** field or in the **structureDescription** field, where it takes values from the X-bar tree that uses the rule or to the **structureCommands** field.

c) **features** f1 **set** &n2
In this example we state a variable of the node feature type that has the name f1. This variable takes values from the nodes that give as values the variable of

the node type with the name n2. If we consider that the variable n2 is the one that we have stated in the first category of examples, then the values of the variable f1 will be [+singular, +human] and [+plural, +adjective]. We must stress that the variable n2 must be already stated above in the **variables** field or in the **structureDescription** field, where it takes values from the X-bar tree that uses the rule or to the **structureCommands** field.

3) The third case is about stating new variables of the anaphor type. This variable takes values from the terminal elements.

a) **anaphor** a1 **set** 'window' : **anaphor** t1 : **anaphor** t2
In this example we define a new variable of the anaphor type that has the name a1. The values of this variable are given by the terminal element 'window' and the anaphors t1 and t2.

b) **anaphor** a1 **set** 'window' : **anaphor** &ta1
In this example we define a new variable of the anaphor type that has the name a1. The values of this variable are given by the anaphors of the terminal element. These anaphors are given by the variable of the anaphor type that has the name ta1. We must stress that the variable ta1 must be already stated above in the **variables** field or in the **structureDescription** field, where it takes values from the X-bar tree that uses the rule or to the **structureCommands** field.

c) **anaphor** a1 **set** &t2
In this example we define a new variable of the anaphor type that has the name a1. The values of this variable are given by the terminal elements that are the values of the variable t2. If we consider that the variable t2 is the one that we have stated in the examples of the first category of variables, then the values that this variable will have are the i1,j1,k1. We must stress that the variable t2 must be already stated above in the **variables** field or in the **structureDescription** field, where it takes values from the X-bar tree that uses the rule or to the **structureCommands** field.

4) The fourth case is about stating new variables of the anaphor type. This variable takes values from the subtrees.

a) **anaphor** a1 **set (node** noun **bar, terminal** 'window'):**anaphor** t1: **anaphor** t2.
In this example we define a new variable of the anaphor type that has the name a1. The values of this variable are the t1 and the t2 and they are given by the subtree **(node** noun **bar, terminal** 'window') with the anaphors t1 and t2.

b) **anaphor** a1 **set (node** noun **bar, terminal** 'window'):**anaphor** &ta1
In this example we define a new variable of the anaphor type that has the name a1. The values of this variable are given by the anaphors of the subtree. These anaphors are given by the variable of the anaphor type that has the name ta1.

We must stress that the variable ta1 must be already stated above in the **variables** field or in the **structureDescription** field, where it takes values from the X-bar tree that uses the rule or to the **structureCommands** field.

d) **anaphor** a1 **set** &s1

In this example we define a new variable of the anaphor type that has the name a1. The values of this variable are given by the anaphors of the subtrees that are the values of the variable s1. If we consider that the variable s1 is the one that we have stated in the examples of the first category of variables, then the values that this variable will have are the i1,j1,k1. We must stress that the variable s1 must be already stated above in the **variables** field or in the **structureDescription** field, where it takes values from the X-bar tree that uses the rule or to the **structureCommands** field.

3.2.3.2 The change of variables values in the structureCommands field

Apart from the declaration of new variables in the **structureCommands** field of principles and transformations, there is the possibility to change the values of the variables that have been stated so far in this rule. These variables can fall either in the **variables** category or in the **transformationVariable** category.

All the methods of changing the values of the variables that have already been stated in the **variables, structureDescription** and **structureCommands** fields are described in this chapter.

Depending on the type of the variable, the abilities to change the values of the variables are the following:

1) For variables of the terminal element type:

a) **terminal** &*name of the terminal variable* **set** *new terminal values*
b) **terminalElement** &*name of the terminal variable* **set** *new terminal value*

The first case changes the values of the variable that has the name *name of the terminal variable*. The new values are the *new terminal values*. The *new terminal values* can be a terminal element or a variable of the terminal type or a terminal that uses a variable for its anaphors.

The second case changes the values of the variable that has the name *name of the terminal variable*. The change is that only the terminal element changes without any changes to the anaphors that the terminal element can possibly have. Therefore, all the values of the terminal variable take as a value the very same terminal element. The terminal element must be a constant and not a variable. It must be, for example, a word or an article.

2) For variables of the tree node type:

a) **node** &*name of the node variable* **set** *new nodes*
b) **features** &*name of the node variable* **set** *new value of the node's feature*
c) **nodeName** &*name of the node variable* **set** *new name of the node*
d) **nodeType** &*name of the node variable* **set** *new type of node*

The first case changes the values of the node type variable and sets new nodes as values with their features, if they exist. The *new nodes* can be given without the use of variables or they can use variables for their features or they can be given with a variable of the node type that has some values.

The second case changes only node features of the node type variable. In this case the *new features* can be given or a variable is used that has as values the new features. These nodes acquire all the same features.

The third case changes only the names of the nodes that are the values of the variable *name of the node variable*. All nodes acquire the same name which is the *new name of the node*. The *new name of the node* should be given. It is not permitted to use a variable.

The fourth case changes only the type of the nodes that are the values of the variable *name of the node variable*. The types of the nodes are the X'', X', X. All the nodes acquire the same type which is the *new type of node*. The *new type of node* must be a constant and it should have one of the following values: **barii, bari, bar**.

3) For variables of the subtree type:

a) **subtree** &*name of the subtree variable* **set** *new subtrees*

For the variables of the subtree type there is only one case of changing the values of the variables. The values of a subtree's variable are replaced by the new values of the *new subtrees*. The *new subtrees* can or cannot have variables. If they do have variables, then the values of the variable *name of the subtree variable* have all these subtrees.

Next we shall analyze examples that are according to above cases.

1) First case of variables of the terminal element type

a) **terminal** &tt1 **set** 'computer' : **anaphor** a1
In this example we set a new value to the variable tt1, the value 'computer' : **anaphor** a1

b) **terminal** &tt1 **set** 'computer': **anaphor** &aa1

In this example we set a new value to the variable tt1, the value 'computer': **anaphor** aa1. We notice that for the anaphors of the terminal element we use the variable that has the name aa1. This variable must be stated and it should also be of the anaphor type. The variable aa1 could either be stated either in the **variables** field or in the **structureDescription** field or in the **structureCommands** field. Thus, if the values of the variable are the anaphorTrace and the anaphorPronoun, then the new values of the variable tt1 will be the following:

'computer': **anaphor** anaphorTrace
'computer': **anaphor** anaphorPronoun

c) **terminal** &tt1 **set** &tt0
In this example we set a new value to the variable tt1. The values that this variable will have are the values of the variable tt0. We must stress that the variable tt0 should be already stated. The system will calculate all the values of the variable tt0 and will assign them as values to the variable tt1.

d) **terminalElement** &tt1 **set** 'the'
In this example we set a new terminal element to the variable tt1. Namely, we set new values to the terminal elements of all the values of the variable tt1, without changing the anaphor. This new value is the article 'the'.

2) The second case of variable of the tree node type

a) **node** &nn1 **set** noun **bar:features** [+human]
In this example we set a new value to the variable nn1. This value is the noun bar:features [+human].

b) **node** &nn1 **set** noun **bar:features** &ff1
In this example we set a new value to the variable nn1, the value noun **bar:features** &ff1. We notice that for the features of the node we use the variable ff1. This variable must be already stated and it should be of the node features type. The variable ff1 could either be stated either in the **variables** field or in the **structureDescription** field or in the **structureCommands** field. Thus, if the values of the variable ff1 are the [-human, +singular] and the [-human, +plural], then the new values of the variable tt1 will be the following:

noun **bar:features** [-human, +singular]
noun **bar:features** [-human, +plural]

c) **node** &nn1 **set** &n2
In this example we set a new value to the variable nn1. The new values of this variable are the values of the variable n2. This variable is of the node type and should have already values.

d) **features** &nn1 **set** [+human]

In this example we set a new value to the features of the nodes that are the values of the variable nn1, the value [+human].

e) **features** &nn1 **set** &ff1
In this example we set a new value to the features of the nodes that are the values of the variable nn1. The new values of the features are the features that are the values of the variable ff1. Thus, if the variable nn1 has as a value the nodes 'verb' **bar:features** [+move] and 'verb' **barii**, and the variable ff1 has as values the [+move, +human] and [+human], then the new values of the variable nn1 are the following:

'verb' **bar:features** [+move, +human]
'verb' **bar:features** [+human]
'verb' **barii:features** [+move, +human]
'verb' **barii:features** [+human]

f) **nodeName** &nn1 **set** verb
In this example we change the name of the nodes that are the values of the variable nn1. All the nodes of this variable will have the name verb.

g) **nodeType** &nn1 **set bari**
In this example we change the type of the nodes that are the values of the variable nn1. All the nodes of this variable will be of the **bari** type.

3) The second case of variable of the subtree category

a) **subtree &ss1 set (node** article **bar, terminal** the)
In this example we set a new value to the variable ss1. This new value is the subtree (node article bar, terminal the).

b) **subtree** &ss1 **set (node** article **bar, terminal** &tt1)
In this example we set a new value to the variable ss1. This new value is the subtree (node article bar, terminal &tt1) that uses the variable tt1. If the variable tt1 has the values 'a', 'an', 'the', then the new values of the variable ss1 will be the following subtrees:

(node article bar, terminal 'a')
(node article bar, terminal 'an')
(node article bar, terminal 'the')

c) **subtree** &ss1 **set** &ss2
In this example we set as values of the variable ss1, the values of the variable ss2. We take it as granted that the variable ss2 has already been stated and has values.

All the above operators set new values to variables of the above types. There are however operators that modify the values of the variables.

These cases are the following:

1) *&name of variable* **addAnaphor** *name of anaphor*

In this case the variable can be either of the terminal element type or of the subtree type. The operator **addAnaphor** adds a new anaphor that is given under the name *name of anaphor*. The new anaphor is added to all the values of the variable under the name *name of variable*. The new anaphor must be a constant and not a variable.

2) *&name of variable* **removeAnaphor** *name of anaphor*

In this case the variable can be either of the terminal element type or of the subtree type. The operator **removeAnaphor** removes the anaphor that is given under the name *name of anaphor*. This anaphor is been removed from all the values of the variable *name of variable*. The erased anaphor must be given as a constant and not as a variable.

3) **node** *&name of the node's variable* **addFeatures** *node features*

In this case it is possible to add features to the nodes of the node type variable. The node features are those that follow the operator **addFeatures**. They should be given and it is not permitted to use a variable.

4) **node** *&name of the node's variable* **removeFeatures** *node features*

In this case it is possible to remove features from the nodes of the node type variable. The node features that are removed are those that follow the operator **addFeatures**. The node features should be given and it is not permitted to use a variable.

5) For the variables of every possible type, there are the following two operators:

a) *&variable name* **addValues** *values of variable*
b) *&variable name* **deleteValues** *values of variable*

These operators change the values of variables of any type by adding or removing their values.

Next we shall analyze examples that correspond to the above cases and show the possibilities provided by the methodology.

a) &tt1 **addAnaphor** anaphorTrace

In this example, the anaphor anaphorTrace will be added to all the terminal elements that are the values of the variable tt1. If the variable tt1 has the values 'the': **anaphor** anaphor1 and 'a': **anaphor** anaphor2, then the new values of the variable tt1 are the following:

'the': **anaphor** anaphor1: **anaphor** anaphorTrace
'a': **anaphor** anaphor2: **anaphor** anaphorTrace

b) &tt1 **removeAnaphor** anaphorTrace

In this example, the anaphor anaphorTrace will be removed from all the terminal elements that are the values of the variable tt1. If the variable tt1 has the values 'the': **anaphor** anaphor1: **anaphor** anaphorTrace and 'a': **anaphor** anaphor2: **anaphor** anaphorTrace, then the new values of the variable tt1 are the following:

'the': **anaphor** anaphor1
'a': **anaphor** anaphor2

c) **node** &nn1 **addFeatures** [-human,+singular]

In this example we will add the features [-human,+singular] to all the nodes that are the values of the variable nn1. For example, if the variable nn1 has the values 'computer' **bar** and 'car' **bar:features** [+nominative], then the new values of the variable nn1 are the following:

'computer' **bar :features** [-human,+singular]
'car' **bar:features** [+nominative, -human,+singular]

d) **node** &nn1 **removeFeatures** [-human,+singular]

In this example we will remove the features [-human,+singular] from all the nodes that are the values of the variable nn1. For example, if the variable nn1 has the values 'computer' **bar:features** [-human,+singular] and 'car' **bar:features** [+nominative,-human,+singular], then the new values of the variable nn1 are the following:

'computer' **bar**
'car' **bar:features** [+nominative]

All the above operators can change the values of variables. They calculate all the possible values of the left and right part then they set according to the operator the new set of values for the variable on the left argument. These values do not contain variables. For the calculation of the values of a subtree variable the operator **anyTree** that may exist is substituted by the trace operator **t**. The only exception is the first operator **addValues** that does not calculate all the values of the left and right part. It only adds the right argument in the set of values of the left argument.

Also, it is possible to calculate all the possible values of a variable according to the other variables that it may use.

The format of this case is the:

- *&name of variable* **set** *&name of variable*

The left and right arguments must have the same variable name. This variable must have been declared.

Finally, there is a command that calulates all the values of variable and deletes all the possible duplicate values that may exist.

This command is the following:

- **deleteDuplicates**(*Variable Name*)

The *variable name* can be the name of a variable of every type and kind.

3.2.3.3 The grammar variables in the structureCommands field

Both the general variables and the transformation variables can be declared as grammar variables. These grammar variables can be used by more than one **principle** and **transformation**. This means that a variable that has been declared in a rule can be used and manipulated (use this variable or change the values of this variable) by the next rule or rules in every field of the three fields of a principle or transformation.

There are two operators related with grammar variables:

- **addGrammarVariable** *name of variable*
- **removeGrammarVariable** *name of variable*

The first operator defines as grammar a variable that has already been defined in one of the fields of a principle or transformation or it is possible to be declared by a next principle or transformation.

The second operator resets a grammar variable as a local one but this variable is still availabe in this principle or transformation that the **removeGrammarVariable** was executed.

Both of the above operators are used in the **structureCommands** field of a principle or transformation.

Also, as it was mentioned above, these operators can be used in the main body of a grammar or even outside of a grammar to delete or declare a grammar variable that can be used in the next rules and grammars. At the case of using the operator

removeGrammarVariable, this grammar variable will not be available in the next rules or grammars.

3.2.3.4 The transformations in the structureCommands field of transformations rules

So far, the abilities regarding the change of the values of the variables have been described. Apart though from changing the values of the variables, it is also possible to modify the X-bar structure on which a transformation is applied. The various variables are very important for the modification of the X-bar trees.

The operator in order to state a set of transformations is the **transformations.**

The general pattern for the transformations is the following:

transformations *transformation 1* **also** *transformation 2* **also**...

It is possible to exist more than one such pattern in a transformation rule.
Every transformation is declared by the operator **transformations** and a sequence of transformations that are connected by the operator **also**.

Each *transformation* of **transformations** is defined as following:

&name of variable of type transformationVariable **transform** *new value*

The *name of variable of type transformationVariable* is a variable of type **transformation variable** that have been declared in the **structureDescription** field of the transformation rule.
The *new value* can be a variable of type tree, node or terminal. Also, it can be a tree, a node or a terminal that may contain differrent kinds of variables. These variables can be variables of transformation type. The type of variable with name *&name of variable of type transformationVariable* must match the type of *new value*.
It must be mentioned that it is possible to change the values of the transformation variables with the operators that have been described in the previous sessions or to declare transformation variables as grammar ones.

The above description of the transformations in the presented methodology shows that its possibilities are more general than the Chomsky's minimal program (Chomsky, 1995) that has as central operations the generalized transformation and the move-a. The generalized transformation is a structure building operation that builds trees in a bottom-up order. This is possible in the presented methodology by using transformations rules and initial or produced x-bar structures of category X, X' and X''. These trees can be selected by principles or grammars that use different commands and especially commands that get a specific X-bar structure from the set of available structures. The move a of the Chomsky's theory is a transformation that moves an element in a higher position (it moves left for the position that it has) in a x-bar tree that already has been built. So, the transformation rules and especially the **transformations** command of the presented methodology gives higher and more general possibilities for describing the required transformations than the Comksky's approaches (Fouskakis, 2005b).

The format of the above rules shows that the transformation possibilities are open and more flexible and powerful than in the TAG (R. Millett, 2004). Operations like adjunction or subjunction in TAGs and in the minimalistic program of Chomsky are a subset of the transformation possibilities of this language.

Also, the presented language takes in consideration comments related the parsing strategies with elementary trees (Fong, 2005). The above transformation rules and the variables permits multiply parallel construnction of structures by its elementary trees and overcome these comments.

Next we shall analyze a series of examples about the transformations.

Example 1

We consider that we have stated two variables of the **transformationVariable** category in the **structureDescription** field of a transformation. These variables have been stated on two different nodes of the X-bar tree. These nodes are described in the structure of the **structureDescription** field of this transformation. We want to change the content of these nodes.
We consider that these two variables have the names **sdn1** and **sdn2.**

In order to change these two nodes we have the following possibilities to state transformations:

a) **transformations**
&sdn1 **transform** noun **bar:features** [+human] **also**
&sdn2 **transform** verb **barii:features** [+plural]

In this case we have the alteration of two nodes that result in a new tree that has these nodes changed. The new values of the nodes are given directly without using any variables and are the following:
i) for the variable sdn1 is the noun **bar:features** [+human]
ii) for the variable sdn2 is the verb **barii:features** [+plural]

b) **transformations**
&sdn1 **transform** &n1 also
&sdn2 **transform** verb **barii:features** &f1

In this case we have the alteration of two nodes where both the stated variables of the **transformationVariable** category also change. We could of course change only one of the two values, by using one of the two transformations.

The transformation for the variable sd1 has as a result, the node of the tree that uses the rule to get all the values of the variable of the node type that has the name n1.

The variable that gives the new values could be the sdn1 itself. This could be done because the system changes the structure and the elements of the tree that uses the transformation only if we execute the transformation command. The variable sdVar1 can change in content like all the variables and with the methods that we described in previous chapters.

Thus, for example, we can add a feature to a node or remove a feature from one node.

Suppose that the variable sdn1 had the value:

verb **bari:features** [+human]

and we execute the command

node &sdn1 **addFeatures** [+plural]

then the variable sdn1 has the value:

verb **bari:features** [+human, +plural]

now we can perform the transformation:

transformations &sdn1 **transform** &sdn1

that will change the respective node of the X-bar tree that uses the rule.

The transformation for the variable sdn2 has as a result, the node of the tree that uses the rule to get as values all the nodes verb **barii** (Verb'') that have as features the values of the variable f1. For example, if the variable f1 has the values [+human] and [+plural], then the two new nodes for the X-bar tree are the following:

i) verb **barii** : features [+human]
ii) verb **barii** : features [+plural]

Therefore, from the X-bar tree that uses the transformation we have the production of all the possible new trees that derive from the replacement of the respective tree nodes by the new nodes.

Example 2

We consider that we have stated a variable of the **transformationVariable** category in the **structureDescription** field of a transformation. This variable has been stated on a terminal element described in the structure of the **structureDescription** field of this transformation. We want to change the content of this terminal element. Suppose that this variable has the name **sdt1**.

In order to change this terminal element we have the following possibilities to state transformations:

a) **transformations**
&sdt1 **transform** 'the':anaphor anaphorTrace

In this case we have the alteration of the terminal element of the X-bar tree and the production of a new tree. This transformation assigns a new terminal element, the word 'the' with the anaphor anaphorTrace.

b) **transformations**

&sdt1 **transform** &t1

This transformation changes the respective terminal element of the X-bar tree and assigns as values all the terminal elements that are the values of the variable t1. This variable must be of the terminal type and it should be already stated. This results in the production of as many new trees as the values of the variable t1.

We can also use the variable sdt1 instead of the variable t1. For example, we can add an anaphor to the terminal element of the tree that uses the rule.
In order to do that, we must perform the following steps:
Suppose that the variable sdt1 had as terminal element the word:

'house'

and we execute the command:

&sdt1 **addAnaphor** anaphorTrace

then the variable sdt1 has the value:

'house': **anaphor** anaphorTrace

now we can perform the transformation:

transformations &sdt1 **transform** &sdt1

that will change the respective node of the X-bar tree that uses the rule.

Example 3

We consider that we have stated a variable of the **transformationVariable** category in the **structureDescription** field of a transformation. This variable has been stated on a subtree described in the structure of the **structureDescription** field of this transformation. We want to change the content of this subtree.
Suppose that this variable has the name **sdst1**.

a) **transformations**

&sdt1 **transform (node** noun **bar,terminal** 'the')

In this case we have the alteration of the subtree of the X-bar tree and the production of a new tree. This transformation results in the new subtree (**node** noun **bar, terminal** 'the').

b) **transformations**

&sd1 **transform** &t1

This transformation changes the respective subtree of the X-bar tree with the subtrees that are the values of the variable t1. This variable must be of the subtree type and it should already be stated in this rule before the transformation.

c) **transformations**

&sd1 **transform (node** &n1,**terminal** 'the')

This transformation changes the respective subtree of the X-bar tree with the subtree (**node** &n1, **terminal** 'the'). This subtree has the variable n1 that should be of the tree node type and it should already be stated in this rule before the above transformation. Therefore, we will have as many subtrees as the values of the variable n1 that fit in the subtree.

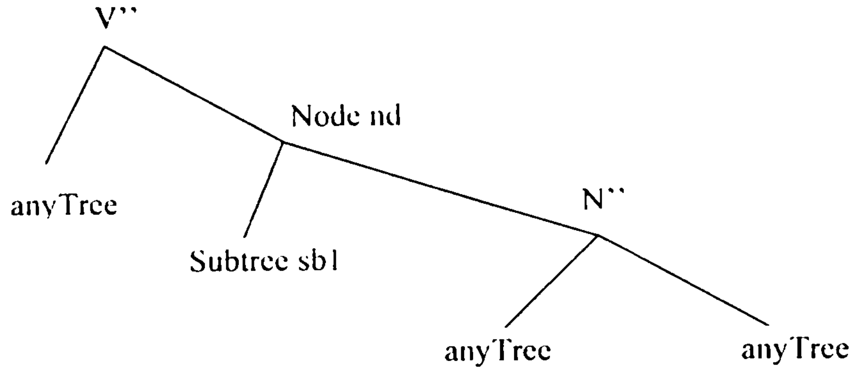
Example 4

Next we shall see how the transformation of the passive voice (Haegeman, 1995) is described in the present methodology.

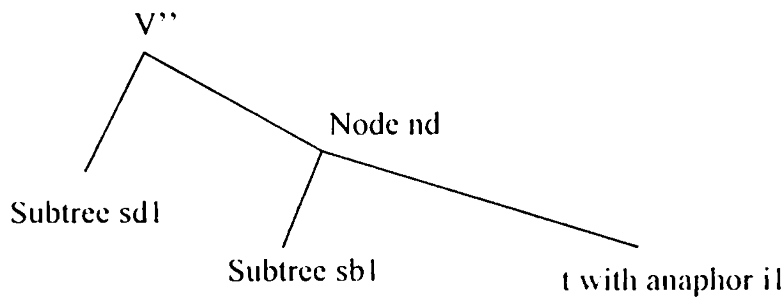
```
transformation 'Passive Voice Transformation'.

noVariables.

structureDescription
   (node 'V' barii,
      anyTree,
      (node &nd,
         subtree &sb1,
         (node 'N' barii, anyTree, anyTree):transformationVariable sd1)
   ):transformationVariable sd2.

structureCommands (
   &sd1 addAnaphor i1,

   transformations
      &sd2 transform
      (node 'V' barii, subtree &sd1,(node &nd,subtree&sb1, t:anaphor i1))
).
```

Analyzing the above transformation, we notice that no variables are stated in the **variables** field, this is why we use the operator **noVariables**.

Also, in the **structureDescription** field we describe the following tree:



In the **structureCommands** field we describe the transformation. In order to achieve the desired transformation, we add first the anaphor i1 to the subtree that has the top N'' and then we use the subtree with the transformation command and we produce the following subtree:



where the subtree sd1 is the subtree with the top N'' and the anaphor i1.

Example 5

We will also describe the rule that shows the shift of the anaphoric element in a sentence that has a relative clause.

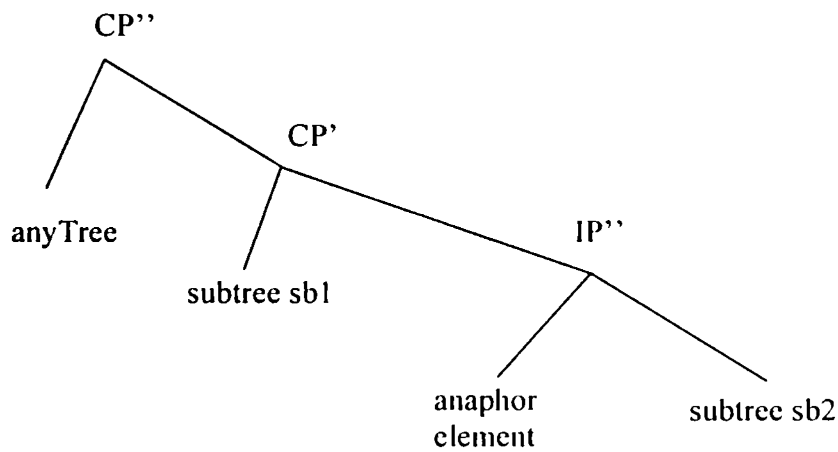
transformation 'Transformation of Anaphor Sentence'.

noVariables.

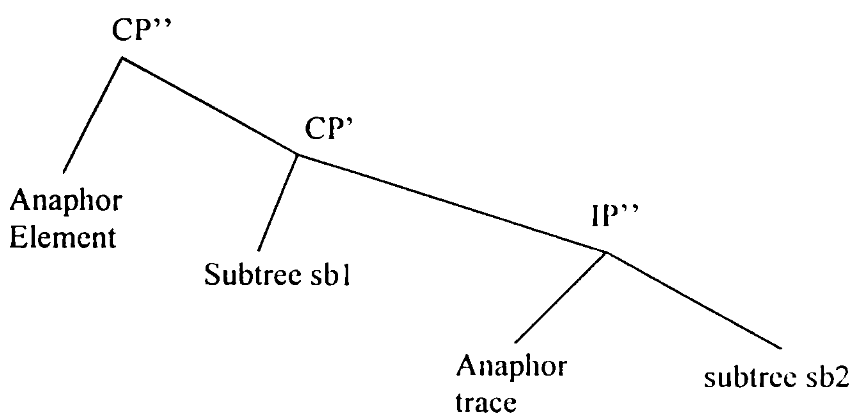
```
structureDescription
   (node 'CP' barii,
       anyTree,
       (node 'CP' bari,
           subtree &sb1,
           (node 'IP' barii,
            (node anaphor bar, terminal &an1), subtree &sb2))
   ):transformationVariable sd1.
```

```
structureCommands (
    &an1 addAnaphor i1,

    transformations
        &sd1 transform
            (node 'CP' barii,
              (node anaphor bar:features [+accusative], terminal &an1),
              (node 'CP' bari,
                 subtree &sb1,
                (node 'IP' barii, t:anaphor i1, subtree &sb2)
              )
            )
).
```

With the transformation that we have stated above, we transform the following subtree.



The new tree of the input sentence will contain the following subtree and in this way we move the anaphoric element to the qualifier of the CP'' node and we put the trace in the previous position of the anaphoric element.



3.2.3.5 The controls in the structureCommands field

In the **structureCommands** field of principles and transformations it is possible to conduct a series of controls by using **if-then-else** structures. These controls deal with the several cases that the rule must cover.

In order to conduct these controls, there is a series of control operators for the control condition **if** which is a part of the **if-then-else** structure.

More specifically, there are the following two control commands:

ifThen(*condition,commands 1)*
ifThenElse(condition,commands 1,commands 2)

It is noticed that both commands have the *condition* which examines if certain desired conditions exist. The result of these control commands can be true or false.

The first of the above two commands has two operands, the *condition* and the *commands 1*. The command **ifThen** examines the condition, namely the first operand, and if it's false, then it doesn't execute the commands of the second operand and the system proceeds in the execution of the next command in the **structureCommands** field of principles and transformations. If the condition is true, then the command **ifThen** proceeds in the execution of the commands of the second operand. If these commands are not executed properly, then the execution of the **ifThen** command is considered to have failed. As a result, the specific principle or transformation also fails, since a command of the **structureCommands** field was not executed successfully.

The second command, the **ifThenElse,** has one more operand apart from the *condition* and *commands 1* which is the *commands 2*. The **ifThenElse** command examines if the *condition* is true and then executes the *commands 1*. If the condition is false, then the command **ifThenElse** executes the *commands 2* and proceeds in the next after the **ifThenElse** command. If one of the *commands 1*, *commands 2* fails, then the whole **ifThenElse** command fails too and as result, the specific principle or transformation also fails, since a command of the **structureCommands** field has failed.

The controls in the *condition* definition of the above commands, can be applied on the elements of the following types:

1) Anaphors
2) Terminal elements
3) Node Features
4) Tree Nodes
5) Subtrees

A) Operators for the anaphors

1. *anaphor 1* **equal** *anaphor 2*
2. *anaphor 1* **notEqual** *anaphor 2*
3. *anaphor 1* **exists** *anaphor 2*

The above operators can have as left and right definition one of the following elements:

a) a sequence of anaphors or a variable of anaphor type
b) a variable of terminal type or a terminal that may contain variables
c) variable of subtree type or a subtree that may contain variables

In the case of a sequence of anaphors, it is necessary to form the left and right arguments as:

anaphor &VariableName1:**anaphor** &VariableName2:...etc....

The other cases do not require operators in the left and right arguments.

The first operator checks if at least one of the values of anaphors of the left part is equal with one of the values of anaphors of the second part. If the left and the right parts are sequence of anaphors, it is necessary to exist at least one sequence of values of anaphors at the left part equal with a sequence of values in the right part.

The second operator is opposite of the first.

The third operator checks if in the left anaphor exists the right anaphor. The right operator must be one specific anaphor. The variables are not permited in this case.

Next we shall analyze a series of examples that show all the cases of using the above operators:

a) &a1 **equal** &a2
b) &a1 **notEqual** &a2

In these two examples we check if the anaphors of the left definition variable are same with the anaphors of the right definition or different from them. The variables a1 and a2 must be of the anaphor type and they should have already been stated. The operator **equal** will check if at least one value of the anaphors of the variable a1 is equal to at least one value of the anaphors of the variable a2, while the operator **notEqual**will check if all the values are different.

c) &a1 **exists** anaphorTrace

In this example we check if the anaphor with the name anaphorTrace is one of the anaphors that has as values the variable a1. This variable should be of the anaphor type and it should have already been stated.

d) &t1 **exists** anaphorTrace

In this example we check if the anaphor with the name anaphorTrace is one of the anaphors of one of the terminal elements that are the values of the variable t1. The variable t1 should be of the terminal element type and it should have already been stated.

e) &st1 **exists** anaphorTrace

In this example we check if the anaphor with the name anaphorTrace is one of the anaphors of one of the subtrees that are the values of the variable st1. The variable st1 should be of the subtree type and it should have already been stated. The anaphors of the subtrees that the operator takes into account, are those that have been stated for the subtree and not for a subtree of this subtree. For example, suppose we have the following subtree:

(**node** article **bar, terminal** 'the':**anaphor** a1):**anaphor** a2

this subtree is of the X category and has article as a node name and the word 'the' as a terminal element. The anaphor that the operators **equal, notEqual**and **exists** take into account is the a2 that concerns the whole subtree that we consider.

f) **anaphor** anaphorTrace:**anaphor** anaphorPronoun **equal** &a1

In this example we check if the variable a1 has as values the anaphorTrace and anaphorPronoun. This variable should be of the anaphor type and it should have already been stated. Also, in this example, we notice the way in which we should describe the anaphors, when they are given in detail and not with the use of a variable. Thus, the two anaphors are described as follows: **anaphor** anaphorTrace:**anaphor** anaphorPronoun. The operator **anaphor** is the operator for the anaphor.

B) Operators for terminal elements

1. **terminal** *terminal 1* **equal terminal** *terminal 2*
2. **terminal** *terminal 1* **notEqual terminal** *terminal 2*
3. **terminalElement** *terminal 1* **equal terminalElement** *terminal 2*
4. **terminalElement** *terminal 1* **notEqual terminalElement** *terminal 2*

The first and the second operators check if the terminals are equal or not. The terminals can be either variables or terminals that may contain variables.

The third and the forth operators check if the terminals are equal or not without checking the anaphors.

The left is equal with the right part if at least one of their possible values is equal.

Next we shall analyze a series of examples that show all the cases of using the above operators:

a) **terminal** &t1 **equal terminal** &t2
b) **terminal** &t1 **notEqual terminal** &t2

In these two examples we compare the terminal elements of the variable t1 to those of the variable t2. The variables t1 and t2 must be of the terminal element type and they should have been stated. In the example a) we compare if a terminal element of the left part is the same as the terminal element of the right part. In the example b) we examine if they are different. In order to compare these terminals we also take into account the anaphors of the terminal elements.

c) **terminal** 'the':**anaphor** anaphorTrace **equal terminal** &t1

In this example we examine if the:**anaphor** anaphorTrace is the terminal element that the variable t1 has as a value. This variable must be of the terminal element type and it should have been stated.

d) **terminalElement** &t1 **equal terminalElement** &t2
e) **terminalElement** &t1 **notEqual terminalElement** &t2

In these two examples we compare the terminal elements of the variable t1 to those of the variable t2. The variables t1 and t2 must be of the terminal element type and they should have been stated. In the example d) we compare if a terminal element of the left part is the same as the terminal element of the right part. In the example e) we examine if they are different. In order to compare these terminals we do not take into account the anaphors of the terminal elements, but only the terminal element.
For example, if we suppose that the variable t1 has the value:

'the':**anaphor** anaph1: **anaphor** anaph2

and the variable t2 has the value:

'the' :**anaphor** anaphorTrace

then we apply the condition of the example d) but not of the example e). The reason is because both variables have as a terminal the word 'the'. Also, we don't apply the condition of the example a) but we apply the condition of the example b). The reason is that the variable t1 has a terminal element with the anaphors anaph1 and anaph2, while the variable t2 has a terminal element with the anaphor anaphorTrace.

C) Operators for node features

1. *features 1* **equal** *features 2*
2. *features 1* **notEqual** *features 2*
3. *features 1* **exists** *feature 2*

4. *features 1* **subsets** *features 2*
5. *features 1* **aCommon** *features 2*

The left and the right arguments of the above operators can be either nodes or features of a node. Also, they can be either variables of the features type or nodes that may contain variables.

It is not required to use the type operators before the left or the right part.

At the third case the right part must be a simple feature:

+ *Name of the feature*
- *Name of the feature*

Name of the feature

Name of the featureX=Name of the featureY

[name of the feature1, ..., name of the featureN]= name of the featureX

The above operators have the following function:

The first operator examines if the features of the left operand are the same as the features of the right operand. If variables are used, the operator examines if they have a value for which the features of the left operand are the same as the features of the right operand.

The second operator examines if the features of the left operand are different from the features of the right operand. If variables are used, the operator must not find a value of these variables, for which value the features of the left operand is the same as the features of the right operand.

The third operator examines if the features of the left operand have the feature of the right operand. The feature of the right operand must be given and it should not be a variable. If a variable is used for the left operand, then the operator should find a value of this variable for which value the features of the left operand have the feature of the right operand.

The fourth operator examines if the features of the left part are a subtotal of the features of the right part. Namely, the features of the left part should exist in the features of the right part. If variables are used in the left or the right operands, then the operator examines if the above applies on a value of these variables.

The fifth operator examines if the features of the left definition have one common feature with the features of the right definition. If variables are used in the left or the right operands, then the operator examines if the above applies to a value of these variables.

Except the general operators for feature checking, there are additional operators that are only for the following kind of features:

- *Name of the featureX=Name of the featureY*
- *[name of the feature1, ..., name of the featureN]= name of the featureX*

These operators check the value of the right part of these kind of features by taking as id their left part. It means that they check the right part if their left part is the same. They return true if there is at least one feature of the above type that has the same left value in both operands and the right part of the feature has a relation between the two operands equal, smaller or greater respectively to the used operator. These operators are the following:

1. **equalFeature**(*FeatureLeftPart,Operand1,Operand2*)
2. **smallerFeature**(*FeatureLeftPart,Operand1,Operand2*)
3. **greaterFeature**(*FeatureLeftPart,Operand1,Operand2*)

The *Operand1* and *Operand2* of the above operators can be either nodes or features of a node. Also, they can be either variables of the features type or nodes that may contain variables.

Next we shall analyze a series of examples that show all the cases of using the above operators:

a) &f1 **equal** &f2
b) &f1 **notEqual** &f2

In these two examples we compare the node features of the variable f1 to those of the variable f2. The variables f1 and f2 must be of the node features type and they should have been stated. In the example a) we compare if one of the values of the variable f1 is the same as the values of the variable f2. In the example b) we examine if all values are different.

c) [+plural,+human] **equal** &f1

In this example we examine if one of the values of the variable f1 is the [+plural,+human]. The variable f1 must be of the node features type and it should have already been stated.

d) &f1 **exists** +plural

In this example we examine if one of the values of the variable f1 has the feature +plural. The variable f1 must be of the node features type and it should have already been stated.

For example if the variable f1 has the values:

1. [+human,+plural]
2. [+human,-plural]

then this operator gives a true value. The reason is that the first value of the variable f1 is the [+human,+plural] that has the feature +plural.

e) &f1 **subsets** &f2
f) &f1 **aCommon** &f2

In these two examples we compare the node features of the variable f1 to those of the variable f2. The variables f1 and f2 must be of the node features type and they should have been stated. In the example e) we compare if one of the values of the variable f1 has features that are a subset of the features assigned to a value of the variable f2. In the example f) we examine if a value of the variable f1 has at least one common feature that exists in a value of the variable f2.

g) [+plural,+human] **subsets** &fl

In this example we examine if the features [+plural,+human] exist in a value of the variable fl. That means that a value of the variable fl should have at least the features +plural and +human. The variable fl must be of the node features type and it should have been stated.

h) [+plural,+human] **aCommon** &fl

In this example we examine if the features [+plural,+human] and one of the values of the variable fl have at least one common feature. The variable fl must be of the node features type and it should have been stated.

D) Operators for tree nodes

1. **node** *node 1* **equal node** *node 2*
2. **node** *node 1* **notEqual node** *node 2*
3. **nodeName** *node 1* **equal nodeName** *node 2*
4. **nodeName** *node 1* **notEqual nodeName** *node 2*
5. **nodeType** *node 1* **equal nodeType***node 2*
6. **nodeType** *node 1* **notEqual nodeType***node 2*

In all the above nodes it is possible to use either node variables or nodes that may contain variables.

The first operator examines if the node of the left operand is the same as the node of the right operand. In order for the two nodes to be the same, they must have the same name, the same type (X'', X' or X) and the same features. If variables are in the left or the right operand, then the above should apply for a value of these variables.

The second operator examines if the node of the left operand is different from the node of the right operand. In order for the two nodes to be different, they must have different name or different type (X'', X' or X) or different features. If variables are used in the left or the right operand, then for all the values of these variables the nodes must be different.

The third operator examines if the node of the left operand has the same name with the node of the right operand. If variables are used in the left or the right operands, then the above should apply for a least one value of these variables.

The fourth operator examines if the node of the left operand has different name from the node of the right operand. If variables are used in the left or the right operand, then the above should apply for every value of these variables.

The fifth operator examines if the node of the left operand has the same type with the node of the right operand. If variables are used in the left or the right operand, then the above should apply for a least one value of these variables.

The sixth operator examines if the node of the left operand has different type from the node of the right operand. If variables are used in the left or the right operand, then the above should apply for all the values of these variables.

Next we analyze a series of examples that show all the cases of using the above operators:

a) **node** &n1 **equal node** &n2
b) **node** &n1 **notEqual node** &n2

In this example we compare the values of the variables n1 and n2. These two variables must be of the tree node type and they should have already been stated. The example a) examines if the variable n1 and the variable n2 have a common value. The example b) examines if all the values of the variable n1 and the variable n2 are different.

c) **nodeName** &n1 **equal nodeName** &n2
d) **nodeName** &n1 **notEqual nodeName** &n2

In this example we compare the values of the variables n1 and n2. These two variables must be of the tree node type and they should have already been stated. The example c) examines if the variable n1 has a value with the same node name with a value of the variable n2. The example d) examines if the variable n1 doesn't have a value with the same node name of a value of the variable n2.

e) **nodeType** &n1 **equal nodeType**&n2
f) **nodeType** &n1 **notEqual nodeType**&n2

In this example we compare the values of the variables n1 and n2. These two variables must be of the tree node type and they should have already been stated. The example e) examines if the variable n1 has a value with the same node type with another value of the variable n2. The example f) examines if the variable n1 doesn't have a value with the same node type of a value of the variable n2.

g) **nodeType** (article **bar) equal nodeType**&n1

In this example we check if the variable n1 has a value of the X category. The variable n1 must be of the node type and it should have been already stated.

h) **nodeName** (article **bar) equal nodeName** &n1

In this example we check if the variable n1 has a value with the node name article. The variable n1 must be of the node type and it should have been already stated.

E) Operators for subtrees

1. **subtree** *subtree 1* **equal subtree** *subtree 2*
2. **subtree** *subtree 1* **notEqual subtree** *subtree 2*

The left and the right subtree of the above operators can be given or it can be a variable or a subtree that uses variables. The above operators take arguments that can be either variables or subtrees that may contain variables. The special operator **anyTree** can be used at any position in the left or right subtrees. This operator declares that it is not interesting the subtree that is going to be at this position of a tree. Also, the operator **t** for the denotation of a trace of a tree is used. Both of these operators can be followed or not by anaphors.

The first operator examines if the subtree of the left definition is the same as the subtree of the right definition. If variables are used in these two operands, then at least one of the values of these variables should have the same subtree.

The first operator examines if the subtree of the left definition is different from the subtree of the right definition. If variables are used in these two operands, then all these variables values of the left and right subtree must be different.

Two trees are equal if they have the same nodes, features of nodes, terminals, anaphors and structure.

Next we shall analyze a series of examples that show all the cases of using the above operators:

a) **subtree** &st1 **equal subtree** &st2
b) **subtree** &st1 **notEqual subtree** &st2

In this example we compare the values of the variables st1 and st2. The variables st1 and st2 should be of the tree node type and they should have been stated. The example a) checks if the variable st1 and the variable st2 have a common value. The example b) checks if all the values of the variable st1 are different from the values of the variable st2.

c) **subtree (node** article, **terminal** 'the': **anaphor** anap1):**anaphor** anap2 **equal subtree** &st1

In this example we examine if one of the values of the variable is the (**node** article, **terminal** 'the': **anaphor** anap1):**anaphor** anap2. The variable st1 must be of the subtree category and it should have been already stated.

F) Finally, there are two operators that check the existence of a variable:

- **varExists** (*Name of Variable)*

- **grammarVar** (*Name of Variable*)

The first case checks if a variable has already been declared.

The second case checks if a variable has already been defined as grammar one.

These operators can be used in a if-then-else rule, in the main body of the **structureCommands** field of the principles and transformations and in the grammars.

So far, various operators have been described that can be applied on elements of different types. There are however operators that combine the above check possibilities.

1. check 1 **and** *check 2*
2. check 1 **or** *check 2*
3. **not** *check*

From the above operators the first one, in order to give a true result, requires both the *check 1* and the *check 2* to be true.

The second operator, in order to give a true result, requires either the *check 1* or the *check 2* to be true or both of them.

The third operator gives a true result if the *check* gives a false result.

At all the above cases, the names of the variables are used with the format:
&N*ame of Variable*

In this part of the principles or transformations, it is possible to use respectively the following commands:

- **principleIncorrect**
- **transformationIncorrect**

These commands declare that the application of a rule on an X-bar tree is false.

3.2.3.6 The EBNF of the structureCommands field

The statement for the <u>scc-principle</u> and <u>scc-transformation</u> of the **structureCommands** field of principles and transformations is the following:

scc-principle = scc-principle-commands".".

scc-transformation = scc-transformation-commands".".

scc-principle-command =
scc-variables-declaration-vars |
scc-variable-value-change |

```
                    ( "ifThen(" scc-condition "," scc-principle-commands ")" ) |
                    ( "ifThenElse(" scc-condition "," scc-principle-commands ","
                                  scc-principle-commands ")" ) |
                    "addGrammarVariable" name |
                    "removeGrammarVariable" name |
                    "varExists(" name ")" |
                    "grammarVar (" name ")" |
                    "deleteDuplicates(" name ")" |
                    "principleIncorrect" |
                    "transformationIncorrect".


scc-transformation-command=
              scc-variables-declaration-vars|
              scc-variable-value-change|
              ("ifThen (" scc-condition "," scc-transformation-commands")") |
              ("ifThenElse (" scc-condition "," scc-transformation-commands
              "," scc-transformation-commands ")") |
              scc-command-tranformations |
              "addGrammarVariable" name |
              "removeGrammarVariable" name |
              "varExists (" name ")" |
              "grammarVar(" name ")" |
              "deleteDuplicates (" name ")" |
              "principleIncorrect" |
              "transformationIncorrect".


scc-principle-commands = "(" scc-principle-command
                              {"," scc-principle-command } ")".


scc-transformation-commands = "(" scc-transformation-command
                              {"," scc-transformation-command} ")".


scc-variables-declaration-vars=
   variables-declaration |
   ("features" features-variable-name "set" tree-node-value) |
   ("anaphor" anaphor-variable-name "set"
              (tree-terminal-value | subtree-value)) |
   ("subtree" subtree-variable-name "set"
                    "nextStructure" ["(" number{number} ")"] ) |
   ("subtree" subtree-variable-name "set"
                    "previousStructure" ["(" number{number} ")"] ) |
   ("subtree" subtree-variable-name "set"
                              "particularStructure (" number{number} ")").
```

```
scc-variable-value-change=
     ("&"name "set" "&"name) |
     ("&"subtree-variable-name "addValues" subtree-value) |
     ("&"node-variable-name "addValues" tree-node-value) |
     ("&"terminal-variable-name "addValues" tree-terminal-value) |
     ("&"anaphor-variable-name "addValues" anaphor-value) |
     ("&"features-variable-name "addValues" node-features-value) |
     ("&"subtree-variable-name "deleteValues " subtree-value) |
     ("&"node-variable-name "deleteValues " tree-node-value) |
     ("&"terminal-variable-name "deleteValues " tree-terminal-value) |
     ("&"anaphor-variable-name "deleteValues " anaphor-value) |
     ("&"features-variable-name "deleteValues " node-features-value).


scc-variable-value-change=
  ("terminal" "&" terminal-variable-name "set"  tree-terminal-value) |
  ("terminalElement" "&" terminal-variable-name "set"  terminal-element).


scc-variable-value-change=
  ("node" "&" node-variable-name "set" tree-node-value) |
  ("features" "&"node-variable-name "set" node-features-value) |
  ("nodeName" "&" node-variable-name "set" node-name) |
  ("nodeType" "&" node-variable-name "set" ("barii" | "bari" | "bar")).


scc-variable-value-change=
  "subtree" "&" subtree-variable-name "set" subtree-value.


scc-variable-value-change=
  "&" (terminal-variable-name | subtree-variable-name)
  ( "addAnaphor" | "removeAnaphor")
  anaphor-name.


scc-variable-value-change=
  "node" "&"node-variable-name
  ("addFeatures" | "removeFeatures")
  "[" feature{"," feature} "]".


scc-condition =
  (   subtree-terminal-variable-anaphors |
      ("&" anaphor-variable-name) |
      tree-terminal-value |
      ("&" terminal-variable-name) |
```

```
    subtree-value |
    ("&" subtree-variable-name)
  )
  ("equal" | "notEqual") |
  (  subtree-terminal-variable-anaphors |
    ("&" anaphor-variable-name) |
    tree-terminal-value |
    ("&" terminal-variable-name) |
    subtree-value |
    ("&" subtree-variable-name)
  ).


sce-condition =
  (  subtree-terminal-variable-anaphors |
    ("&" anaphor-variable-name) |
    tree-terminal-value |
    ("&" terminal-variable-name) |
    subtree-value |
    ("&" subtree-variable-name)
  )
  "exists"
  anaphor-name.


sce-condition =
  ("terminal" tree-terminal-value ("equal" | "notEqual")
                                   "terminal" tree-terminal-value) |
  ("terminalElement" tree-terminal-value ("equal" | "notEqual")
                                          "terminalElement" tree-terminal-value).


sce-condition =
  (tree-node-value | node-features-value )
                    ("equal" | "notEqual" | "subsets" | "aCommon")
  (tree-node-value | node-features-value ).


sce-condition =
  ( "equalFeature(" feature ","     (tree-node-value | node-features-value ) ","
                                    (tree-node-value | node-features-value ) ")"  ) |
  ( "smallerFeature(" feature ","   (tree-node-value | node-features-value ) ","
                                    (tree-node-value | node-features-value ) ")"  ) |
  ( "greaterFeature(" feature ","   (tree-node-value | node-features-value ) ","
                                    (tree-node-value | node-features-value ) ")"  ).


sce-condition =
```

(tree-node-value | node-features-value)
"exists"
feature.

scc-condition =
"node" tree-node-value ("equal" | "notEqual") "node" tree-node-value.

scc-condition =
"nodeName" tree-node-value ("equal" | "notEqual")
"nodeName" tree-node-value.

scc-condition =
"nodeType" tree-node-value ("equal" | "notEqual")
"nodeType" tree-node-value.

scc-condition =
"subtree" subtree-value ("equal" | "notEqual") "subtree" subtree-value.

scc-condition =
"varExists(" name ")" | "grammarVar (" name ")".

scc-condition = "not" "(" scc-condition ")".

scc-condition = "(" scc-condition ("and" | "or") scc-condition ")".

scc-command-tranformations =
"transformations" scc-command-transform
{ "also" scc-command-transform }.

scc-command-transform =
"&" sd-variable-name "transform" scc-variable-value.

scc-variable-value =
tree-node-value |
tree-terminal-value |
subtree-value.

From the above statements the variables-declaration, tree-node-value, tree-terminal-value, subtree-value, anaphor-value, node-features-value, terminal-variable-name, node-variable-name, subtree-variable-name, anaphor-variable-name, features-variable-name, subtree-terminal-variable-anaphors were declared in the chapter for the **variables** field of the principles and transformations. Also, the terminal-element, node-name, anaphor-name, feature, name, number were declared in the chapter that describes the structures that the methodolody.

The software that implements the described functionality has been implemented in SWI-Prolog 5.0.10. This prolog has been created by the Department of Social Informatics (SWI) of the University of Amsterdam. This prolog is possible to be installed as embedded application in a pocket PC. It has been implemented as a set of different modules in the mentioned prolog. These are the following:

1. User
2. Sys_db

   This module contains all the predicates that store the current status of the system when it manipulates the X-bar structures.
3. Operators

   This module contains all the operators that are used by all other modules of the system.
4. General_predicates

   This module contains a set of predicates that are used in different modules of the system.
5. Sys_elements

   This module describes the different elements that are manipulated by the system. Different predicates determine the correct form of the different kind of elements (nodes, terminals, anaphors, features, trees).
6. Main_module

   The main module is the first module that starts the application.
7. Read_files

   This module reads the principles, transformations and grammars.
8. Read_Write_Structures

   This module reads the input structures and produces the output structures according to the rules and the grammars.
9. Execute_rules

   This module executes the grammars, principles and transformations. They are determined by the corresponding operator and the name.
10. Vars_field

    This module manipulates the declaration of variables in the field **vars** of the principles and transformations.
11. Sd_field

    This module analyses the current input structure that the particular rule is applied, according to the structural description of its **sd** field.
12. Scc_field

This module contains all the predicates for variables declaration and change of variables values in the **sec** field of principles and transformations.

13. Sec_checks

The different kinds of checks in the **sec** field of principles and transformations.

14. Sec_transformations

This module has all the necessary predicates for the definition of the transformations that we can apply in the input structure.

15. Comments

This module writes the comments that are declared in rules.

As it is mentioned above:

- The User module is the default module that is visible by all the other modules.
- The module Operators defines the operators in the User module. These operators are used by all the modules of the system.
- The module sys_db stores the current status of the natural processing system.

The remains modules follow with their corresponding dependences:

1. General_predicates
2. Sys_elements
3. Main_module
    - operators
    - read_files
    - read_write_structures
4. Read_files
    - sys_db
    - general_predicates
5. Read_Write_Structures
    - sys_db
    - general_predicates
    - sys_elements
    - execute_rules
6. Execute_rules
    - sys_db
    - general_predicates

- sys_elements
- vars_field
- sd_field
- scc_field
- scc_checks
- scc_transformations
- comments

7. Vars_field
   - sys_db
   - general_predicates
   - sys_elements
8. Sd_field
   - sys_db
   - general_predicates
   - sys_elements
   - vars_field
9. Scc_field
   - sys_db
   - general_predicates
   - sys_elements
   - vars_field
   - scc_checks
   - comments
10. Scc_checks
    - sys_db
    - general_predicates
    - sys_elements
    - vars_field
11. Scc_transformations
    - sys_db
    - general_predicates

- vars_field

12. Comments

- sys_db
- general_predicates
- vars_field

All the above modules will be described in details in the following sections using the notation and predicates of prolog.

3.3.1 Implementation specific details

3.3.1.1 The comment command

In principles and transformations it is possible to state comments in the **structureCommands** field. These comments are entered in the system's output as further information for the specific principle or transformation. Also, they are possible and in the main body of a grammar but at this case it is not possible to use variables.

For the comment we use the command **comment** and then we enter the comment that we wish to be printed in the system's output.
The general form of this command is the following:

comment (*comment 1: comment 2: comment 3:...)*

We notice that in this command the comments are separated with the character :. Each one of the comments can be a constant, a prolog atom. Namely, it is a sequence of letters and numbers included between quotes. The atom of the prolog is printed as it is. Also, every comment can be a variable that should have been stated. Then the system prints all the values of this variable. Each variable is used as follows:

(&*name of variable)*

we notice that we must use parentheses that will include the name of the variable and the character &.

There is also the operator **newline** that changes the line in the output and writes the rest of the comments in the next line.

Next we shall analyze a series of examples:

a) **comment** ('The values of the variable a1 are the following' : (&a1))

This comment prints the message 'The values of the variable a1 are the following' and then a list of the values of the variable a1. The variable a1 should have been stated.

b) **comment** ('The values of the variable n1 are the following' : (&n1) : newline : ' and of the t1 are the' : (&t1))

This comment prints the message 'The values of the variable n1 are the following' and then prints a list of the values of the variable n1 and changes the line. Then it prints the message ' and of the t1 are : ' and then the values of the variable t1. The variables n1 and t1 should have been stated.

The EBNF form for the comments is the following:

scc-message = "comment" scc-comment { ":" scc-comment }.

scc-comment = name | ("(" "&" comment-variable-name ")") | "newline".

comment-variable-name =
node-variable-name |
features-variable-name |
terminal-variable-name |
subtree-variable-name |
anaphor-variable-name.

From the above statements the node-variable-name, features-variable-name, terminal-variable-name, subtree-variable-name have been described in the EBNF of the **variables** field of the **principles** and **transformations**.

3.3.1.2 The user depending application of the rules

Another possibility is the ability to selectively apply a rule according to the response of the user. This is available in the main part of a grammar of the *Linguistic Theories* input and in the *Linguistic Program* input.
These cases are the following:

askprinciple *name of principle*
asktransformation *name of transformation*
askgrammar *name of grammar*

We notice that we can use the operators **askprinciple, asktransformation** and **askgrammar**, instead of the operators **principle, transformation** and **grammar**. When a grammar in the *Linguistic Theories* input wishes to apply one of these rules or in the *Linguistic Program* input it is requested the application of the a principle, transformation or grammar with the operators **askprinciple, asktransformation** and **askgrammar**, it is necessary the positive or negative response of the user.

The EBNF form of the *Linguistic Theories* input has additionally the:

rule= "askprinciple" principle-name |
"asktransformation" transformation-name |
"askgrammar" grammar-name.

3.3.1.3 The changes on the operators and other assumptions

There are some changes on the operators at the implemented system comparing with the description in the previous chapters. These changes facilitate more the use of the software system. These changes are:

- **variables** becomes **vars**
- **noVariables** becomes **noVars**
- **structureDescription** becomes **sd**
- **structureCommands** becomes **scc**
- **structureposition** becomes **position**
- **transformationVariable** becomes **sdVar**

Backtracking is possible in the **sd** field. In the **scc** field it is possible to use any other prolog predicate except the elements (checks, transformations etc) that have been described for this field of the **principles** and **transformations**. Also, in the main body of a **grammar** it is possible to use any prolog predicate except the **principles**, **transformations** and the other commands that had been described about the main body of a **grammar** rule. A **transformation** rule succeeds if at least one of the requested transformations in its **scc** field succeeds and produces a new X-bar tree. The transformation and principle rules are applied on on all the **sd** subtrees that exist in an x-bar tree. If the operator **aFirstTree** is used the rule is applied on the first **sd** subtree (scanning top-down left-right) that exists in an X-bar tree.

3.3.2 Module sys_db

This module contains all the predicates that store the current status of the system when it manipulates the X-bar structures.

These are the following predicates with their corresponding arity:

- input_file/1
- output_file/1
- execute_rule_grammar/1
- grammar/2
- principle_rule/4
- transformation_rule/4
- new_is/1
- new_os/1
- read_is/2
- in_struct/1
- out_struct/1
- is_trees/1
- rule_succeed_trees/1
- variables/2
- grammar_variables/1
- see_transformations/1

The above predicates store the following information in more details:

- It keeps the input file stream for the input structures
    - input_file(_).
- It keeps the output file stream for the results
    - output_file(_).
- It keeps all the grammars, principles and transformations that we want to execute according to the linguistic program
    - execute_rule_grammar([]).
- It keeps every grammar
    - grammar(999999,_). (dummy grammar)
- It keeps every principle
    - principle_rule(999999,_,_,_). (dummy principle)
- It keeps every transformation
    - transformation_rule(999999,_,_,_). (dummy transformation)

- It keeps the X-bar trees that are going to be used by the next grammar, principle or transformation.
  - new_is([]).

- It keeps the final X-bar trees that have been produced by the last grammar, principle or transformation.
  - new_os([]).

- It keeps the last structure that has been gotten from the input file trees
  - read_is(0,[]).

- It keeps the current input X-bar tree for the running rule
  - in_struct(_).

- It keeps the current output X-bar tree for the running rule
  - out_struct(_).

- It keeps the set of all the input file X-bar trees
  - is_trees([]).

- it keeps the only succeeded trees that a rule is apllied on
  - rule_succeed_trees([]).

- It keeps the variables of the current principle or transformation that is executed and applied on an X-bar tree
  - variables([],[]).

- It keeps all the transformations of the transformation rule that is currently executed
  - see_transformations([]).

- It keeps the names of the grammar variables
  - grammar_variables([]).

3.3.3 Module operators

This module contains all the operators that are used by all other modules of the system. These operators are set in the user module.

The operators that we use for the description of the transformations and principles are the following:

```
:-op(100,fy,user:principle).
```

```
:-op(100,fy,user:transformation).
:-op(100,fy,user:grammar).
:-op(950,fy,user:vars).
:-op(950,fy,user:sd).
:-op(950,fy,user:scc).
:-op(100,fy,user:askprinciple).
:-op(100,fy,user:asktransformation).
:-op(100,fy,user:askgrammar).
```

The operators **vars**, **sd** and **scc** are used for the declaration of the corresponding **variables**, **structureDescription** and **structureCommands** fields of the principles and transformations.

The operators that we can use in these fields of the principles and transformations are the:

```
:-op(800,xfy,user:set).
:-op(850,xfy,user:also).
:-op(800,xfy,user:addValues).
:-op(800,xfy,user:deleteValues ).
```

The first operator is used in the fields **vars** and **scc** to set values.
The second operator is used in the field **vars** to declare two or more different variables and also in the field **scc** to declare a sequence of transformations.
The third and forth operator are used in order to add and to remove values of variables respectively

The operators that are used in different parts of a tree are the following:

```
:-op(620,fy,user:node).
:-op(400,fy,user:features).
:-op(620,fy,user:terminal).
:-op(400,fy,user:anaphor).
```

The first operator is for the declaration of a node in a X-bar tree or in the fields of a principle or transformation. The second operator is for the declaration of the features the nodes. The third operator is for the declaration of a terminal element. The forth operator is for the declaration of the anaphors of the trees and of the terminals.

The various categories of nodes are defined by the following operators:

```
:-op(200,yf,user:barii).
:-op(200,yf,user:bari).
:-op(200,yf,user:bar).
```

The first operator is for the X'' nodes the second is for the X' nodes and the third is for the X nodes.

The operators for the declaration of the transformations in the sec field of the transformations are the:

```
:-op(900,fy,user:transformations).
:-op(650,xfy,user:transform).
```

The first operator determines a whole sequence of transformations and the second is used to declare every different part in a transformation sequence.

In the field **sd** of the principles and transformations we can use additionally the following:

```
:-op(400,fy,user:sdVar).
:-op(400,fy,user:subtree).
:-op(400,fy,user:notSubtree).
:-op(400,fy,user:nodeSubtree).
:-op(400,fy,user:nodeNotSubtree).
:-op(400,xf,user:anyTree).
:-op(600,fy,user:aTree).
:-op(600,fy,user:aFirstTree).
:-op(600,fy,user:leftMost).
```

The first operator is used for the declaration of transformation type variables that are going to be mainly used in the field **sec** for the transformations.

The other operators correspondingly declare the following:

- subtree of a tree
- not subtree of a tree
- subtree of a tree that is described by a root node
- not subtree of a tree that is described by a root node

- any tree
- a tree that is a arbitrary subtree of a tree
- a tree that is the first subtree of a tree
- a tree is the left most subtree (top-down left-right tree scanning)

In the see field of the principles and transformations we can use the following operators:

For the determination of a part of an element:

:-op(620,fy,user:terminalElement).

:-op(620,fy,user:nodeType).

:-op(620,fy,user:nodeName).

The first operator is used for the determination of the name of a terminal.
The second is used for the determination of the type of a node.
The third operator is used for the determination of the name of a node.

For the change of anaphors or features of an element:

:-op(650,yfy,user:addFeatures).

:-op(650,yfy,user:removeFeatures).

:-op(650,yfy,user:addAnaphor).

:-op(650,yfy,user:removeAnaphor).

The first operator is used for the addition and the second for the substraction of the features of a node.
The third is used for the addition and the forth for the subtraction of an anaphor in a terminal or subtree variable.

For the checks of the various elements are the operators:

:-op(650,xfy,user:equal).

:-op(650,xfy,user:notEqual).

:-op(650,xfy,user:aCommon).

:-op(650,xfy,user:subsets).

:-op(650,xfy,user:exists).

The above operators determine accordingly the following:

- the first checks if two elements are equal
- the second checks if two elements are not equal

- the third checks if two elements have a common feature
- the forth checks if the features are subset of another set
- the fifth checks if a feature or an anaphor exists

There are the operators for grammar variables:

```
:-op(100,fy,user:addGrammarVariable).
:-op(100,fy,user:removeGrammarVariable).
```

The first operator adds a new grammar variable the second removes a grammar variable from the set of the grammar variables. They are used in the *principles* and *transformations input*, in the *linguistic theory input* and in the *linguistic program input*.

Also, there is the following general operator that is used in different parts of a principle or transformation:

```
:-op(500,xfy,:).
```

We finally use the following sequence of operators:

```
:-op(300,fy,user:(&)).
:-op(670,yfx,user:and).
:-op(675,yfx,user:or).
:-op(900,fy,user:not).
```

The first is used in front of the variables names.

The second, third and forth function as the known logical operators in the different kind of checks.

The operators **and** and **or** can be used in the **sd** field to describe combinations of trees.

```
:-op(520,fy,user:comment).
```

It is used for the description of a comment in a rule (grammar, principle and transformation)

3.3.4 Module general_predicates

This module contains a set of predicates that are used in different modules of the system.

The first determines the module that stores the current information of the system when it functions. We can easily change the module name in order to use another module for storage.

main_database(sys_db).

It follows a brief description of the predicates:

It converts a term of the form (term1 or term2 or ... or termN) in a list of the form [term1,term2,...,termN]

orTermToList(+(term1 or term2 or ... or termN), ?[term1,term2,...,termN])

It searches if an operator is in a set of operators

member_op(?Operator, +(Operator1:Operator2:...:OperatorŒ))

It deletes an operator from a set of operators

remove_op(+Operator, +(Operator1:Operator2:...:OperatorŒ), ?(Operator1:...:OperatorŒ),)

It compares two lists

compare(+List1,+List2)

It deletes the repeated elements of a list

delete_duplicates(List, List2 without the repeated elements)

It substitutes an element of a list with another element

replace(+A, +B, +L_in, ?L_out)

It checks if an element is a list or not

list(+List)

It deletes an element from the main database if it exists

sys_retract_all(+Element)

It inserts an element in the main database

sys_assert(+Element)

It reads an element in the main database

sys_read(+Element)

It succeeds if Condition and Clause succeed or if the Condition fails
ifThen(+Condition,+Clause)

Succeeds if Condition and Clause1 succeed or if the Condition fails and Clause2 succeeds
ifThenElse(+Condition,+Clause1,+Clause2)

Succeeds if Clause1 and Clause2 succeeds
and(+Clause1,+Clause2)

Succeeds if Clause1 or Clause2 succeeds
or(+Clause1,+Clause2)

It converts a set of terms in the corresponding list
list_to_term(?List of elements, ?Compound term)

It converts a list of elements in a compound term with the elements connected by the operator **and**
convert_list_to_and_term(+Input list, ?Compound Output Structure)

It inserts the (not) operator in every element of the list
insert_not_operator(+Input List, ?Output List)

It writes in the output device a tree
writetree(+OutStream,+Subtree)

It gives the correct form in an input X-bar tree
input_str_conv(+InTree,?CovertedTree)

The checking of the main part of a grammar
chk_grammar(+The main body of the grammar)

It checks the term if it is one of the accepted forms
chk_rule_grammar(+Term)

Deletion of the existing elements of list 1
delete_common_features(+features list 1, +features list 2,

+features list 2 without features of list 1)

It checks if the feature exists in a list of features

feature_exists(+Feature, +List of features)

It returns all the anaphors in one list

list_anaphors(?Sequence of anaphors, ?Anaphors List)

It substitutes the node name

replace_node_name(+Node, +New node name, ?New node)

It substitutes the type of a node

replace_node_type(+Node, +New node type, ?New node)

3.3.5 Module sys_elements

This module describes the different elements that are manipulated by the system. Different predicates determine the correct form of the different kind of elements (nodes, terminals, anaphors, features, trees).

<u>ANAPHORS</u>

The form of anaphors that the system accepts, are determined by the following predicates:

A single anaphor

a_anaphor(+Anaphor).

A sequence of anaphors

a_anaphors(+Anaphor).

TERMINALS

The form of the terminals that the system accepts is determined by the following predicates.

Terminals without anaphors:

a_simple_terminal(+Terminal).

Terminals with anaphors

a_terminal(+Terminal).

FEATURES
The form of the features of the nodes that the system manipulates is determined by the following predicates:

The following determines the different kind of single feature:
a_feature(+Feature).

The following determines a complete set of the features of a node:
a_features(+Features).

NODES
The form of the nodes that the system accepts, are the following:

The following determines the nodes of type X'', X', X in a general form:

a_node(+Node).

The following determines the nodes of type X'', X', X in a form without features:

a_simple_node(+Node).

The following predicates determine in details the nodes of type X'', X', X in a form with or without features:

a_node_barII(+Node).

a_node_barI(+Node).

a_node_bar(+Node bar).

TREES
The following predicates determine the X-bar trees that the system accepts as input, produces as output and manipulates.

The following predicates determines a tree of type X2, X1, X0 and returns the list of all its anaphors.

a_tree_value(+Tree,AnaphorsList) :-

But the most important predicates that determine the exact form of trees that the system manipulates are described in the following lines. Every predicate has two anaphor lists. The first is the input list of anaphors and the second pair is the new list of anaphors.

The different kinds of trees are represented with the different predicates:

A tree of type X without anaphors

```
a_tree_bar(LAnaphors,(node Node,terminal Terminal),NodeName,LAnaphors) :-
 return_node_name(Node,NodeName),
 a_node_bar(Node),
 a_simple_terminal(Terminal).
```

A tree of type X with anaphors in the terminal element

```
a_tree_bar(LAnaphors1,(node Node,terminal Terminal:Anaphors), NodeName,
        LAnaphors2) :-
 return_node_name(Node,NodeName),
 a_node_bar(Node),
 a_simple_terminal(Terminal),
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2).
```

A tree of type X with anaphors at X

```
a_tree_bar(LAnaphors1,(node Node,terminal Terminal):Anaphors, NodeName,
        LAnaphors2) :-
 return_node_name(Node,NodeName),
 a_node_bar(Node),
 a_simple_terminal(Terminal),
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2).
```

A tree of type X with anaphors at the terminal element and at X

```
a_tree_bar(LAnaphors1,(node Node,terminal Terminal:Anaphors1):Anaphors2,
              NodeName, LAnaphors3) :-
 return_node_name(Node,NodeName),
 a_node_bar(Node),
 a_simple_terminal(Terminal),
 value_anaphors_seq(Anaphors1,ValAnaphors1),
 chk_anaphors_connections(LAnaphors1,ValAnaphors1,LAnaphors2),
 value_anaphors_seq(Anaphors2,ValAnaphors2),
 chk_anaphors_connections(LAnaphors2,ValAnaphors2,LAnaphors3).
```

A tree trace with anaphors

```
a_tree_bar(LAnaphors1,t:Anaphors,_,LAnaphors2) :-
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2).
```

A tree trace without anaphors

```
a_tree_bar(LAnaphors,t,_,LAnaphors).
```

A tree of type X'

```
a_tree_barl(LAnaphors1,(node Node,SubTree1,SubTree2),NodeName,
                LAnaphors3) :-
 return_node_name(Node,NodeName),
 a_node_barl(Node),
 a_tree_barl(LAnaphors1,SubTree1,NodeName,LAnaphors2),
 a_tree_barl1(LAnaphors2,SubTree2,_,LAnaphors3).
```

A tree of type X' with anaphors

```
a_tree_barl(LAnaphors1,(node Node,SubTree1,SubTree2):Anaphors, NodeName,
                LAnaphors4) :-
 return_node_name(Node,NodeName),
 a_node_barl(Node),
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2),
 a_tree_barl(LAnaphors2,SubTree1,NodeName,LAnaphors3),
 a_tree_barl1(LAnaphors3,SubTree2,_,LAnaphors4).
```

A tree of type X'

```
a_tree_barl(LAnaphors1,(node Node,SubTree1,SubTree2), NodeName,
        LAnaphors3) :-
 return_node_name(Node,NodeName),
 a_node_barl(Node),
 a_tree_bar(LAnaphors1,SubTree1,NodeName,LAnaphors2),
 a_tree_barl1(LAnaphors2,SubTree2,_,LAnaphors3).
```

A tree of type X' with anaphors

```
a_tree_barl(LAnaphors1,(node Node,SubTree1, SubTree2):Anaphors, NodeName,
        LAnaphors4) :-
 return_node_name(Node,NodeName),
 a_node_barl(Node),
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2),
 a_tree_bar(LAnaphors2,SubTree1,NodeName,LAnaphors3),
 a_tree_barl1(LAnaphors3,SubTree2,_,LAnaphors4).
```

A tree trace with anaphors

```
a_tree_barl(LAnaphors1,t:Anaphors,_,LAnaphors2) :-
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2).
```

A tree trace without anaphors

```
a_tree_barI(LAnaphors,t,_,LAnaphors).
```

A empty tree

```
a_tree_barI(LAnaphors,empty,_,LAnaphors).
```

A tree of type X"

```
a_tree_barII(LAnaphors1,(node Node,SubTree1,SubTree2),NodeName,
        LAnaphors3) :-
  return_node_name(Node,NodeName),
  a_node_barII(Node),
  a_tree_bar(LAnaphors1,SubTree1,_,LAnaphors2),
  a_tree_barI(LAnaphors2,SubTree2,NodeName,LAnaphors3).
```

A type X" tree with anaphors

```
a_tree_barII(LAnaphors1,(node Node,SubTree1,SubTree2):Anaphors, NodeName,
        LAnaphors4) :-
  return_node_name(Node,NodeName),
  a_node_barII(Node),
  value_anaphors_seq(Anaphors,ValAnaphors),
  chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2),
  a_tree_bar(LAnaphors2,SubTree1,_,LAnaphors3),
  a_tree_barI(LAnaphors3,SubTree2,NodeName,LAnaphors4).
```

A type X" tree

```
a_tree_barII(LAnaphors1,(node Node,SubTree1,SubTree2),NodeName,
        LAnaphors3) :-
  return_node_name(Node,NodeName),
  a_node_barII(Node),
  a_tree_barII(LAnaphors1,SubTree1,_,LAnaphors2),
  a_tree_barI(LAnaphors2,SubTree2,NodeName,LAnaphors3).
```

A type X" tree with anaphors

```
a_tree_barII(LAnaphors1,(node Node,SubTree1,SubTree2):Anaphors,NodeName,
        LAnaphors4) :-
  return_node_name(Node,NodeName),
  a_node_barII(Node),
  value_anaphors_seq(Anaphors,ValAnaphors),
  chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2),
  a_tree_barII(LAnaphors2,SubTree1,_,LAnaphors3),
  a_tree_barI(LAnaphors3,SubTree2,NodeName,LAnaphors4).
```

A tree trace with anaphors

```
a_tree_barII(LAnaphors1,t:Anaphors,_,LAnaphors2) :-
 value_anaphors_seq(Anaphors,ValAnaphors),
 chk_anaphors_connections(LAnaphors1,ValAnaphors,LAnaphors2).
```

A tree trace without anaphors

```
a_tree_barII(LAnaphors,t,_,LAnaphors).
```

A empty tree

```
a_tree_barII(LAnaphors,empty,_,LAnaphors).
```

3.3.6 Module main_module

The main module loads the module with the operators. It is the first module that is loaded in order to start the application. Also, it uses the modules read_files and read_write_structures.

The available predicates to other modules are the:

- read_rules/0, read_rules/3
  - read_rules without arguments it uses the default file names (f1.rg, f1.gr, f1.rl)
  - read_rules(+LinguisticProgramFile, +LinguisticTheoryFile, +PrinciplesTransformationsFile)

  It reads the following files:
  1. The sequence of rules, *linguistic programm*, (principles, transformations and grammars) that we want to apply on the input X-bar trees.
  2. The grammars that we have declared.
  3. The principles and the transformations that we have declared.

- rw_trees/0, rw_trees/2
  - rw_trees without arguments it uses the default file names (f1.is, f1.os)
  - rw_trees(+InputStructuresFile, +OutputStructuresFile)

  It applies the rules on the input X-bar trees and produces the output file with the results.

The used predicates from other modules are the:

- from the read_files module:

  - get_execute_grammar_rule/1
  - get_grammars/1
  - get_rules/1
- from the read_write_structures module:
  - get_structures/2

The next section describes in details the functionality of these predicates.

3.3.7 Module read_files

This module reads the three input files:

1. The sequence of rules (principles, transformations and grammars) that we want to apply on the input X-bar trees.
2. The grammars that we have declared.
3. The principles and the transformations that we have declared.

The first kind of file is read by the following predicate:

get_execute_grammar_rule(+The file name)

The second kind of file is read by the following predicate:

get_grammars(+The name of file)

The third kind of file is read by the following predicate:

get_rules(+The name of the rules file)

The above predicates use a set of supplementary predicates that mainly are the following:

- It reads the principles, transformations and grammars that we want to apply on the input X-bar trees – The linguistic program
  - read_execute_grammar_rule(+FileHandle)
- It reads the grammars from the input file
  - read_grammar(+FileHandle)
- It reads the principles and transformations from the input file
  - read_rules(+FileHandle)
- It reads a principle or transformation from the input file
  - readRule(+FileHandle, ?Rule, ?Rule name)

- It checks if the rule is priciple or transformation
  - rule_cat(+Rule name, ?Rule structure,
    ?Rule name, ?Variables field, ?SD structure field,
    ?SCC field)
- It checks if the principle or transformation has variables or not
  - rule_vars(+The input field, +The output field)

3.3.8 Module read_write_structures

Reading of the input structures and production of the output structures according to the principles, transformations and grammars that we want to apply on input structures.

The main predicate is:

get_structures(+The name of the input file, +The name of the output file)

The above predicate uses the following:

- read_structures(+TreeNumber, +Input file, ?TreesList)
- execute_rules_on_input_trees(+Grammars, +Input Trees List)

It reads every input structure and applies on them the set of principles, transformations and grammars that we have declared that we want to apply.

The above uses the predicate:

perform(+List of rules that we want to apply)

3.3.9 Module execute_rules

The grammars, principles and transformations are determined by the corresponding operator and the name.

The following predicates define the operators:

grammar
askGrammar
principle
askPrinciple
transformation
askTransformation

and the:

addStructures
setStructures
setSucceededStructures
restoreStructure
getNextStructure
getPreviousStructure
addGrammarVariable
removeGrammarVariable
getInputTreeId
newInputTrees
addInputTrees

The predicate for the application of a grammar with user question:

askgrammar(+Grammar name)

and without user question

grammar(+Grammar Name)

The predicate for the application of a principle according to the user answer

askprinciple(+Principle name)

The procedure for the application of a principle on all the X-bar trees

principle(+Principle name)

It applies the principle on one X-bar tree

perform_principle(+Principle name, +Input structure)

The predicates for the transformations according to the user choice

asktransformation(+Transformation name)

The application of a transformation on a set of input structures

transformation(+Transformation name)

The application of the transformation in one input structure

perform_transformation(+Transforamtion name, +Input structure)

The procedures for the change of the input structures in different cases:

An addition of the produced structures in the current input structures for the next rule

addstructures

It sets as a new set of structures for the next rule the structures that have been prodused by the last rule

setstructures

It sets as X-bar trees for the next rule of the grammar, only the trees that the last rule have been applied on successfully

setsucceededstructures

It restores as input structure the structure that has read from the input file

restorestructure

It gets the next input structure from the input file for use by the next rule

getnextstructure

It gets the previous input structure from the input file for use by the next rule

getpreviousstructure

It gets the a particular input structure from the input file for use by the next rule

getparticularstructure(+Location)

It returns the id of the current input X-bar tree.

getinputtreeid(?Id)

They change the input structures and needs as operand an *Id*
newinputtrees (+Id)
addinputtrees(+Id)

The operators for the grammar variables are the following:
It adds a variable to the grammar variables list
addgrammarvariable(+VariableName)

It removes a variable from the grammar variables list
removegrammarvariable(+VariableName)

3.3.9.1 Module vars_field

This module manipulates the declaration of variables in the field **vars** of the principles and transformations.

The main top level predicate for the declaration of the predicates of **vars** field is the:

declareVars(+The field for the declaration of the variables)

which takes as a parameter the field **vars** of a principle or a transformation in order to declare its variables.

The following predicate is used for the declaration of a variable

varList(+The variables that the user have declared)

The above use the predicate newVariable/5 that analyses and checks every new variable:

newVariable(+a variable with its type,
+Values of a variable,
?Name of a variable,
?Type of a variable,

?Values of a variable)

3.3.9.2 Module sd_field

This module analyses the current input structure that the particular rule is applied, according to the structural description of its **sd** field. The analysis of the input structure results at a new output structure that is used by the **scc** field of principles and transformations.

It checks if the input structure is according with the structure that is described in the field SD.
The predicate is:

declareSD(+SD Description, +Input Structure, ?Output Structure)

The following predicate checks the input structure in order to find the structure of the SD field (the input structure must be subtree of the categories X2, X1, X0):

searchSD(+Input Variables,
+Structure of the SD, +Input Structure, ?Output Structure,
?Output Variables)

It checks the tree of the SD if it is equal with the input structure according to the operators and the abilities of that had been described in the above sections.
The predicate is the:

check_sd_is_tree(+Variabels, +SD structure Variables,
+Structure of the SD, +Input Structure,
?Output Structure,
?New set of Variables,
?New set of Variables of the SD type)

3.3.9.3 Module scc_field

This module contains all the predicates for variables declaration and change of variables values in the SCC field of principles and transformations.

The declaration of the operator **set** for the variables of the SCC field

+variable set +values of variable

The variables are separated in different categories according with their type and we have the following cases:

The general case for the declaration of the new variables of the **Vars** field type is implemented by the following predicates:

It declares a new variable with value the next input structure

set(subtree Var, nextstructure).

set(subtree Var, nextstructure(Num)).

It declares a new variable with value the previous input structure

set(subtree Var, previousstructure).

set(subtree Var, previousstructure(Num)).

It declares a new variable with value a particular input structure that is according to the number that we use as parameter

set(subtree Var, particularstructure(Num)).

It declares the new variable and adds to the list of variables

set(+Var,+VarValues).

Variable declaration for the node features that takes values from another variable or node formula

set(features +VarName, +Node).

Variable declaration for the terminals elements and subtrees anaphors that takes values from other variables or formulas of type subtree or terminal

set(anaphor +VarName, +SubtreeTerminal)

The other case is the setting of new values in variables that we have defined. The cases are the followings:

The change of the values of a terminal variable

set(terminal &+VarName,+Terminal).

The change of the values of a terminal variable without to change the existing anaphors:

set(terminalElement &+VarName,+TerminalElement).

The change of the value of a tree node:

`set(node &+VarName,+NewNode).

The change of the value of node features:

set(features &+VarName,+NewFeatures).

The change of the name of a node:

set(nodeName &+VarName,+NewNodeName).

The change of the node type (barII,barI,bar):

set(nodeType &+VarName, +NewNodeType).

The change of a subtree variable values:

set(subtree &+VarName, +SubTree).

Except the operators that set new values in different variables, there are also operators for the addition or the deletion of nodes features and the addition or the deletion of terminals or subtrees anaphors.

The case of addition of an anaphor in the values of the variable:

+variable addAnaphor anaphor name

addAnaphor(&+VarName, +AnaphorName).

The following case describes the subtraction of an anaphor from a variable with anaphors. The variable can be of anaphor type, subtree or terminal.

+variable removeAnaphor anaphor name

removeAnaphor(&+VarName, +AnaphorName).

The case for the declaration of the command for the addition of the node features of a node variable.

+variable addFeatures +features list

addFeatures(node &+NodeVar,+FeaturesList).

The case for the declaration of the command for the substraction of the node features of a node variable.

+variable removeFeatures +features list

removeFeatures(node &+NodeVar,+FeaturesList).

Operators for addition or deletion of variables values for variables have already been declared.

addValues(&+VarName, +VarValues)

deleteValues (&+VarName, +VarValues)

It possible to calculate all the values of a variable and to delete the duplicate values.

deleteDuplicates(+VarName)

The above predicates require a set of additional predicates.

The most important are described in brief:

The addition of an anaphor in terminals and subtrees
add_anaphor(+Element, +New anaphor, ?New element)

The deletion of anaphor of terminals or subtrees
remove_anaphor(+Element, +Anaphor, ?New Element)

The features addition in a node
add_node_features(+Node, +Features that we have to add, ?The node with the new features)

The deletion of features of a node
remove_node_features(+Node, +Features that we have to delete, ?The node with the new features)

3.3.9.4 Module scc_checks

The different kinds of checks in the scc field of principles and transformations are in this module. They are used in the condition part of the **ifThen** or **ifThenElse** at the **scc** field.

The anaphors are equal
equal(+L_Anaphors, +R_Anaphors).

The terminal elements are equal

equal(terminal +L_Terminal, terminal +R_Terminal).

The terminals are equal independent of their anaphors

equal(terminalElement +L_Terminal, terminalElement +R_Terminal).

The features are equal

equal(+L_Features, +R_Features).

The comparison of nodes and their features
equal(node +L_Node, node +R_Node).

The comparison the node names
equal(nodeName +L_Node, nodeName +R_Node).

The comparison of the node types
equal(nodeType +L_Node, nodeType+R_Node).

The operator equal for the comparison of the trees:
equal(subtree +L_Subtree, subtree +R_Subtree).

Also, there are the operators **notEqual** that are the opposite of all the above.

Except the operator **equal** and **notEqual** there are also the operators **exists, subsets, aCommon**.

An anaphor exists in the anaphors
exists(+Anaphors, +Anaphor).

A feature exists in the features of a node or a features variable
exists(+L_Features, +R_Feature).

It checks if the features of the left part are subset of the features of the right part. The node or features variables can be used in both parts.
subsets(+L_Features, +R_Features).

If the features of the left part have at least one common feature with the features of the right part.
aCommon(+L_Features, +R_Features).

The cases of special checks of the features with the format (name of the featureX = name of the featureY or [name of the feature1, ..., name of the featureN]= name of the featureX)

equalFeature(+*FeatureLeftPart*,+*Operand1*,+*Operand2*)
smallerFeature(+*FeatureLeftPart*,+*Operand1*,+*Operand2*)
greaterFeature(+*FeatureLeftPart*,+*Operand1*,+*Operand2*)

The above predicates use a set of additional predicates. Most of them are described in the following predicates:

It compares the two terminal elements

compare_terminals(+Terminal 1 , +Terminal 2)

It returns the value of a terminal formula
get_terminal(+Variables, +Formula, ?Terminal Value)

It returns the terminal element without its anaphors
get_terminal_element(+Variables, +Formula, ?Node Value)

It compares two nodes
compare_nodes(+Node 1 , +Node 2).

It returns the value of the corresponding node representation
get_node(+Variables, +Formula, ?Node Value)

It returns the node name
get_node_name(+Variables, +Formula, ?Node Value)

It returns the type of node
get_node_type(+Variables, +Formula, ?Node Type)

It compares the two sets of features
compare_features(+Features List 1 , +Features List 2)

It compares the two features sets / if the first is subset of the second set
subset_features(+Features List 1 , +Features List 2)

It compares two sets if the first has a common element with the second
aCommon_features(+Features List 1 , +Features List 2)

It returns the features of a formula or node
get_features(+Variables, +Formula, ?Features List)

It compares two subtrees if they are equal
compare_subtree(+Left Subtree, ?Right Subtree)

3.3.9.5 Module scc_transformations

This module has all the necessary predicates for the definition of the transformations that we can apply on an X-bar tree. The transformations are declared only in the transformation rules in the **scc** field.

The transformations can be applied on the nodes, terminals and trees.

transformations(+Transformations)

The sequence of transformations that we want to apply on an X-bar tree are manipulated be the predicate:

transformation_list(+Transformations)

The different transformations in a transformation sequence are declared by the:

transform_to(+Transformation that we want to execute)

3.3.10 Module comments

This module writes to the output the comments. The main predicate that implements this is the:

comment(+Sequence of comments)

3.4 General examples of principles and transformations and anaphoric connections

<u>The principle of case filter</u>(Haegeman, 1995) and in the greek language the case problem(Drachman,1984)

principle 'Case Filter'. % definition of principle of case

variables
  node noun **set** 'NP' **bar or** 'O' **bar**.

structureDescription
    (**node** &noun: **transformationVariable** sd1, **terminal** &t):
                                                transformationVariable sd2.

structureCommands
  (**features** case **set** [+ptosi] **or** [+case],

    ifThenElse(&sd1 **aCommon** &case,
      comment "The principle of case filter is valid at : ":&sd2,
      comment " The principle of case filter is not valid at : ":&sd2)).

The above principle acts upon X-bar structures which have one of the following two sub trees:

Then at the field **structureCommands** checks if the node NP or O has the feature +case or the feature +ptosi and sends the corresponding message at the output.

The following are examples of linguistics rules that have been expressed according the presented methodology. Also, a grammar variable is declared in the field **structure commands** of the **principle** 'The rule of dominance'.

The rule of c-command (Roberts, 1997)

C-commands

An X element commands structurally (c-commands) an Y element, if and only if the first bifurcated node that dominates the X, dominates also the Y and neither the X dominates the Y nor the Y dominates the X.

principle 'The rule of constituent command'.

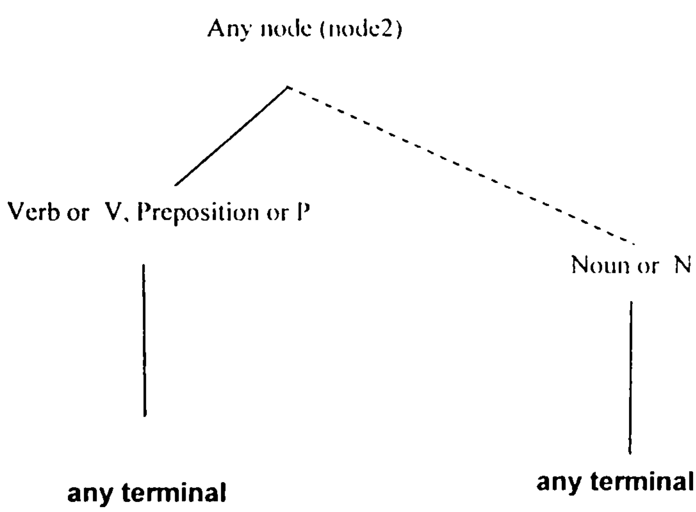
variables
node node1 **set** 'Verb' **bar or** 'V' **bar or**
'Preposition' **bar or** 'P' **bar**
also node n **set** 'Noun' **bar or** 'N' **bar**.

structureDescription

(**node** &node2,
(**node** &node1, **terminal** &t1): **transformationVariable** sd1,
aTree (**node** &n, **terminal** &t2): **transformationVariable** sd2)

structureCommands
comment &sd1:' c-commands ':&sd2,
addGrammarVariable sd1.

The above principle acts upon a X-bar structure that has a sub tree of the following structure:



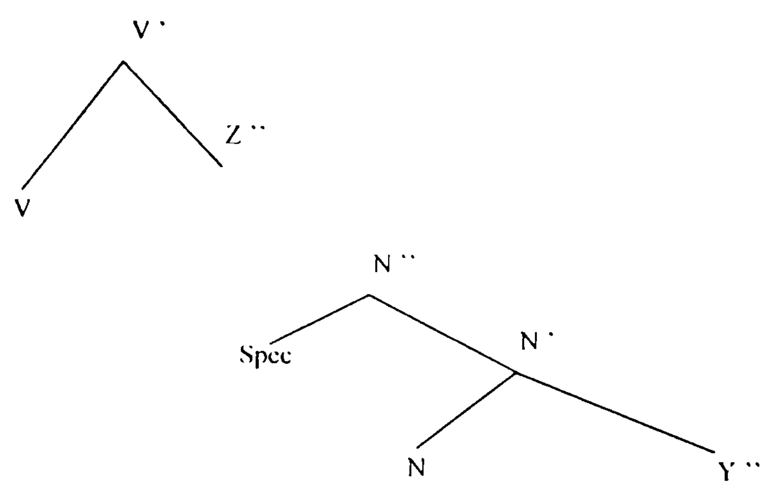
The discontinuous line means that the right sub tree can be at any depth as the operator **aTree** describes.

BUPT

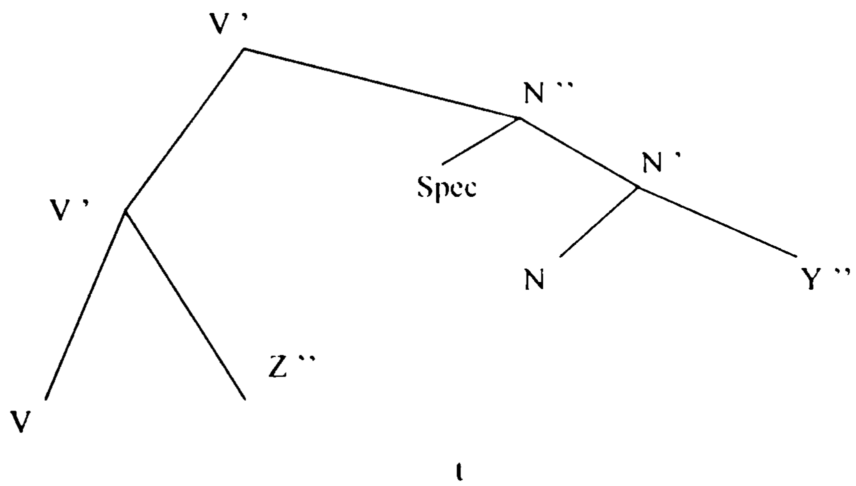
The rule of noun phrase attachment(Roberts, 1997) and about the greek language passive voice and movements(Campos, 1987)(Theofanopoulou, 1986, 1989b)

```
transformation        'Attachment of noun phrase'.

variables
node 'Noun'           set     'N' barii or 'Noun' barii
also node 'V' set     'V' bari or 'Verb' bari.

structureDescription
   (node &'V':sdVar sd3,
        subtree &sb1,
        (node &'Noun', anyTree, anyTree):sdVar sd1
   ):sdVar sd2.

structureCommands
        (&sd1 addAnaphor i1, % addition of anaphor reference

          % declaration of transformations
          transformations  &sd2 transform
                              (node &sd3,
                                   (node &sd3, subtree &sb1, t:anaphor i1),
                                   subtree &sd1)
        ).
```

The above transformation acts upon a X-bar structure that has a sub tree of the following structure and produces a new X-bar structure:



And the produced X-bar structure is the following:



Since, the **sd1** variable of the first rule was declared as a grammar variable and the second rule use a variable with the same name we will have a conflict, if we execute these two rules, because we have two variables with the same name in the transformation rule (the grammar one from the first rule and the local one from the transformation). We must always use a different notation for the names of the variables that we intend to use as grammar variables.

Additionally, the above transformation it is possible to be used in a grammar rule and to be applied repeatedly. In this case it is possible to use a grammar variable for the anaphoric connections in order to have the same anaphoric connections between the different traces.

3.4.1 The problem of anaphoric connections outside of an X-bar tree

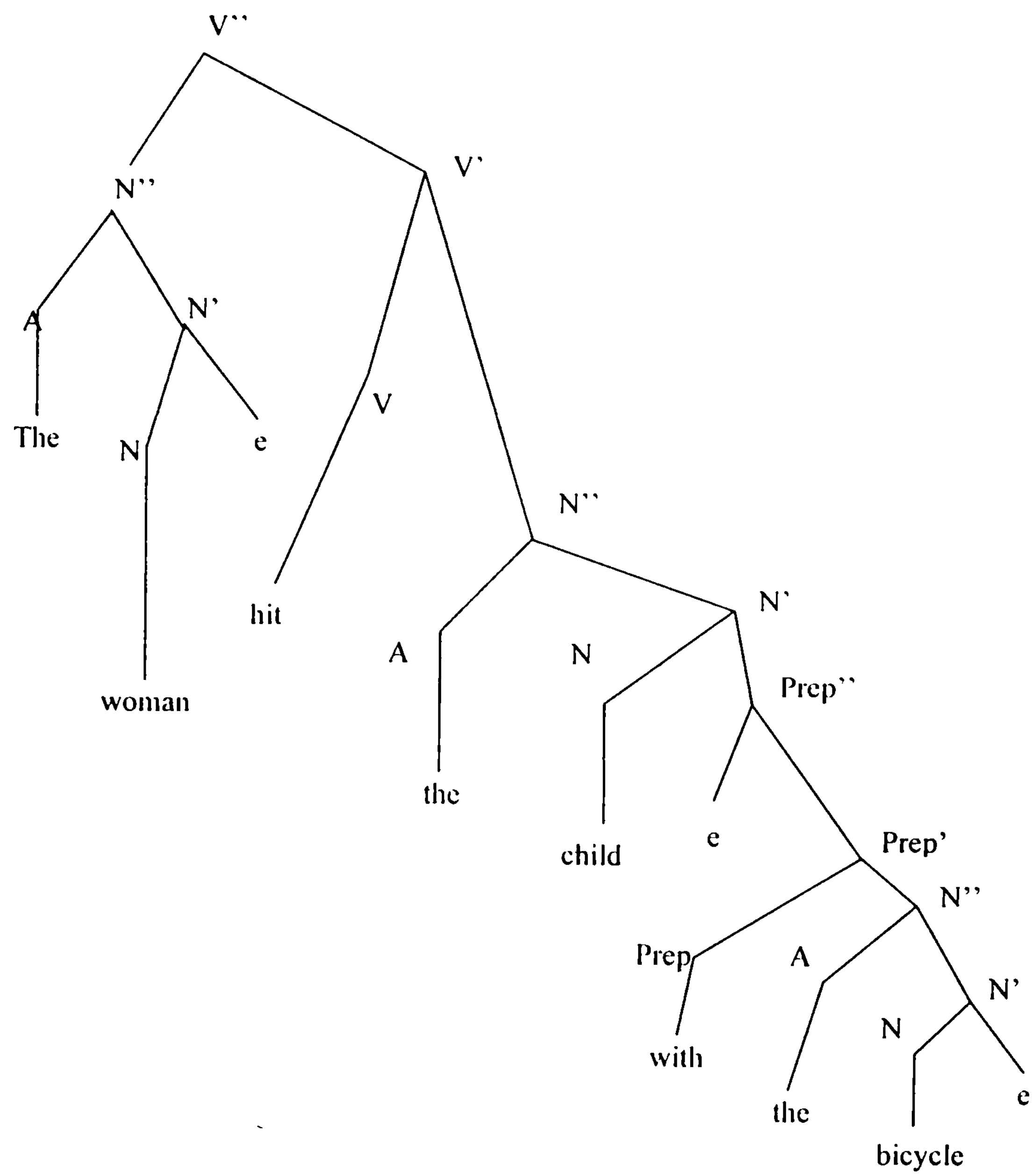
In an application, it is possible to translate the sentences without to know explicitly the anaphoric connections that are outside of the X-bar tree of a sentence. If it is necessary for an application to have anaphoric connections outside of an X-bar tree, there is the ability to connect two or more trees (backward or forward from the current tree) with a conjunction by using the word 'and' and declaring the necessary transformations for the production of the new trees. Also, it is possible to connect two or more X-bar trees by using other words or pseudo-words that are related with the semantic connection (result, explanation, parallel information, analysis) between these sentences. It is necessary for this semantic connection to declare principles that specify the type of a sentence according to its elements (verbs, nouns, articles, pronouns and their combinations). Additionally, it is possible to use grammar variables in the rules of the above cases in order to exchange information between them.

An example with two phrases is the following:

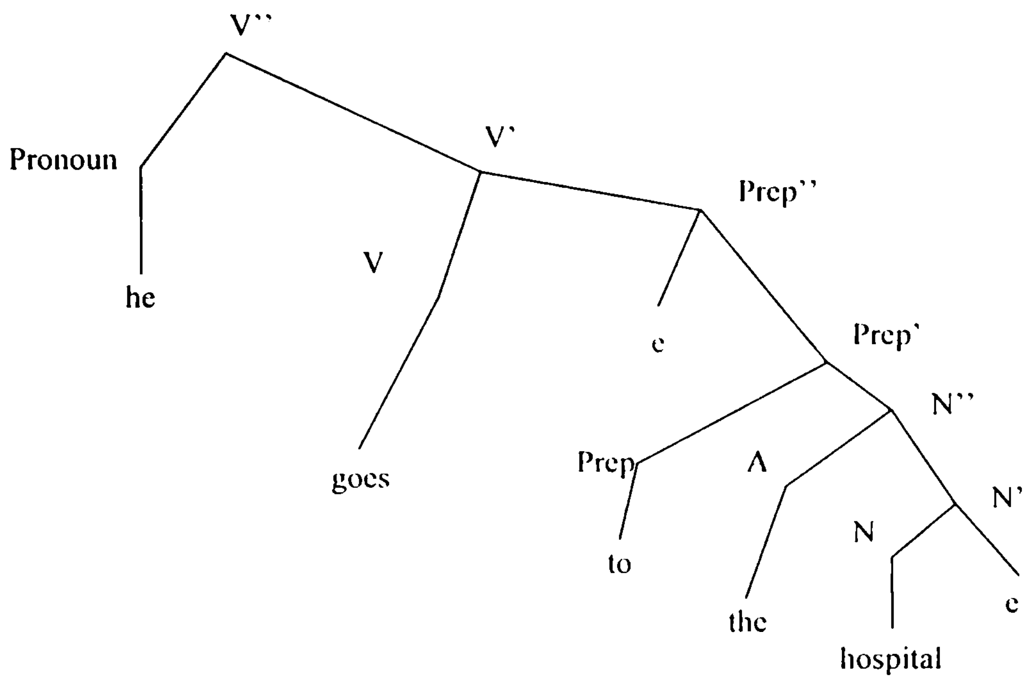
- The woman hit the child with the bicycle
- He goes to the hospital.

The initial X-bar trees of the above two phrases can be produced by using the phrase structure rules of the X-bar scheme. Their correct final forms are produced by using a set of principles, transformations or theories that are necessary.

The tree of the first sentence is the following:



The tree of the second sentence is the following:



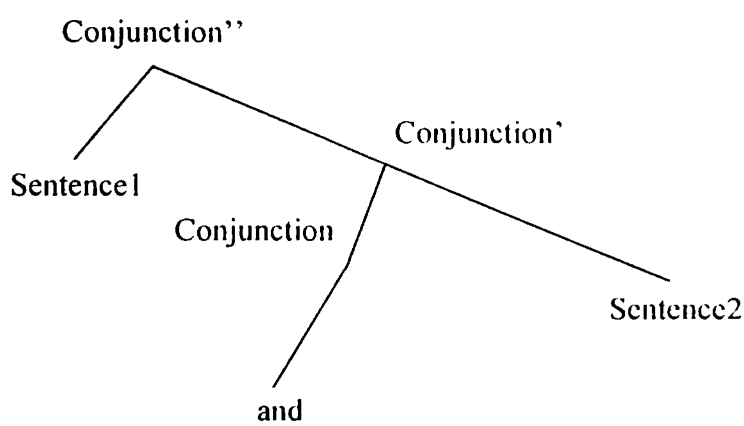
Let's assume the following:

- *Sentence1* is the X-bar tree of the sentence:
    - The woman hit the child with the bicycle
- *Sentence2* is the X-bar tree of the sentence:
    - He goes to the hospital.

Then the sentence that combines that above two sentences is the following:

The woman hit the child with the bicycle and he goes to the hospital.

The corresponding final tree has the form:



The rule that produces the above new tree is the following:

transformation 'Conjunction'.

noVariables.

structureDescription
subtree (&sentence1):**transformationVariable** conj_tree

structureCommands
subtree sentence2 **set nextstructure**,
transformations
&conj_tree **transform**
(**node** conjunction **barii**,
subtree &sentence1,
(**node** conjunction **bari**,
(**node** conjunction **bar**, terminal 'and'),
subtree &sentence2
)
).

The above transformation has the name 'Conjunction'. This transformation does not have any variable in the **variables** field and the operator **noVariables** is used. In the **structureDescription** field it is declared a transformation variable with name *conj_tree*. This variable is used in order to produce a new tree that has the tree of the current sentence and the tree of the next sentence. Also, there is the general variable *sentence1* that its value is the tree of the sentence that the rule is applied on. In the **structureCommands** field, the X-bar tree of the next sentence is read and set as value of the variable *sentence2* by using the first command of this field of the above transformation rule. Since, both the trees are values of the variables *sentence1* and *sentence2* we can produce the new tree by using the transformation variable *conj_tree*. This is that second rule in the **structureCommands** field of the above transformation rule.

Except the above rule that produces the new sentence, it is necessary to declare a new rule that sets the anaphoric connection between the two words of the sentences that refer to the same object.

transformation 'Anaphoric_Connection'.

noVariables.

structureDescription
(**node** conjunction **barii,**
aTree (**node** n **bar:features** &feat_first,
(**terminal** &nounTerm): **transformationVariable** &tv1),
aTree (**node** pronoun **bar:features** &feat_second,

```
        (terminal &pronounTerm): transformationVariable &tv2)
).

structureCommands
  features man set [+human,+masculin],
  features woman set [+human,+feminine],

  ifThenElse(
        (&man subsets &feat_first and &man subsets &feat_second) or
        (&woman subsets &feat_first and &woman subsets &feat_second),

        (&tv1 addAnaphor a1,
         &tv2 addAnaphor a1,
         transformations      &tv1 transform &tv1 also
                              &tv2 transform &tv2
        ),
        fail)
```

This transformation rule does not use any variables in the variables field. The two terminal elements one of the first sentences and one of the second sentences are described in the **structureDescription** field. Also, there are the two transformations variables tv1 and tv2 that are used in order to change the tree and add the anaphors. The variables *feat_first* and *feat_second* contain the features of the nodes of category X that are in the corresponding trees. In the **structureCommands** field we declare two new variables that describe the features that must have the terminals in order to be connected. The **ifThenElse** checks if both terminals have the same required features, adds the anaphoric connection with the name a1 between them and then it does the transformations of the two terminal elements. The result is a new tree that has the required anaphoric connection.

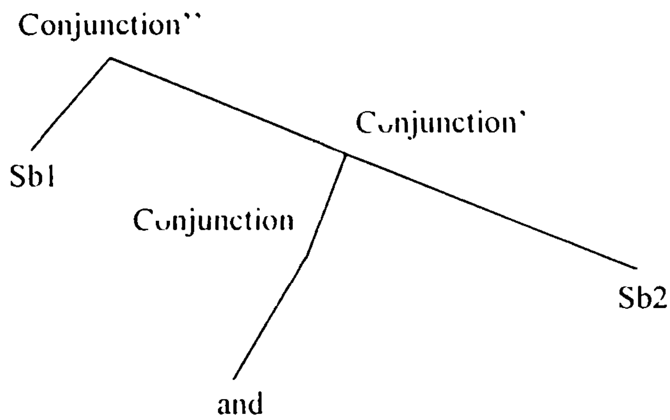
Additionally, it is possible to separate the two sentences. The necessary transformation rule is the following:

```
transformation 'Separation'.

noVariables.

structureDescription
  (node conjunction barii,
        subtree &sb1,
        (node conjunction bari,
              (node conjunction bar, terminal 'and'),
              subtree &sb2
        ):transformationVariable &treeVar.

structureCommands
         transformations  &treeVar transform &sb1,
```

transformations &treeVar **transform** &sb2.

The above transformation produces the two trees that had been connected with the 'Conjunction' transformation. These two trees will contain the anaphoric connection that has been added by the transformation 'Anaphoric_Connection'.

The transformation 'Separation' does not use variables in its **variables** field. In the **structureDescription** field it is described the following tree:



The variables sb1 and sb2 take as values the trees of the first and second sentences respectively and the two transformations in the **structureCommands** field produce the two trees.

The above rules can be applied and in the following case:

- The woman hit the child with the bicycle.
- It is difficult for him to ride.

The result will be the connection of the terminal 'child' with the pronoun 'him'.

Also, in the following case:

- The woman hit the child with the bicycle.
- She is very angry.

But there is problem in the following sentences:

- The woman hit the boy with the bicycle.
- She believes that he stole it.

This case has two sentences that can have more than anaphoric connections.

These connections are the following:

- Woman -> she
- Boy -> he
- Bicycle -> it

The problem is how we can extend the above rules in order to cover this case of anaphoric connections.

The only rule that has to be changed is the transformation rule with name 'Anaphoric_Connection'. The other two rules must remain unchanged.

This transformation rule connects only two elements that is feminine or masculine; this is examined by their features. One element is noun and the other is pronoun. But in the last two sentences the there is the word 'bicycle' and the pronoun 'it' that refers to things not to humans. So, it is necessary to extend the checks and for the features [+thing]. This extension permits the anaphoric connection between the words 'bicycle' and 'it'.

The above transformation rule will be as following:

```
transformation 'Anaphoric_Connection'.

noVariables.

structureDescription
 (node conjunction barii,
   aTree ( node n bar:features &feat_first,
           (terminal &nounTerm): transformationVariable &tv1),
   aTree ( node pronoun bar:features &feat_second,
           (terminal &pronounTerm): transformationVariable &tv2)
).

structureCommands
  features man set [+human,+masculin],
  features woman set [+human,+feminine],
  features thing set [+thing],

  ifThenElse(
   (&man subsets &feat_first and &man subset &feat_second) or
   (&woman subsets &feat_first and &woman subsets &feat_second) or
   (&thing subsets &feat_first and &thing subsets &feat_second),

        (&tv1 addAnaphor a1,
         &tv2 addAnaphor a1,
         transformations      &tv1 transform &tv1 also
                              &tv2 transform &tv2
        ),
        fail)
```

There is another problem. The anaphoric connection must be different between the different elements. This requires an implementation specific predicate that returns different anaphors; let's say **new_anaphor** (e.g. a counter that returns a different value for every anaphoric connection). In order to have all the anaphoric connection (three in this example) it is necessary the repeated application of the above rule. This is possible by a grammar rule with recursion.

```
grammar 'Anaphoric Connections'.
        transformation 'Anaphoric_Connection'.
        grammar 'Anaphoric Connections'.
```

The above transformation rule 'Anaphoric Connection' in order to have correct repeated application must check if an anaphoric connection has already been created. So, it is necessary for a new check to be added and the transformation rule becomes finally as following:

```
transformation 'Anaphoric_Connection'.

noVariables.

structureDescription

 (node conjunction barii,
   aTree ( node n bar:features &feat_first,
           (terminal &nounTerm): transformationVariable &tv1),
   aTree ( node pronoun bar:features &feat_second,
           (terminal &pronounTerm): transformationVariable &tv2)
).

structureCommands

 ifThen(&tv1 aCommon &tv2,fail),

 features man set [+human,+masculin],
 features woman set [+human,+feminine],
 features thing set [+thing],

 ifThenElse(
        (&man subsets &feat_first and &man subsets &feat_second) or
        (&woman subsets &feat_first and
                                   &woman subsets &feat_second) or
        (&thing subsets &feat_first and &thing subsets &feat_second),

        (new_anaphor(CommonAnaphor),
         &tv1 addAnaphor CommonAnaphor,
         &tv2 addAnaphor CommonAnaphor,

         transformations      &tv1 transform &tv1 also
                              &tv2 transform &tv2
        ),
        fail)
```

The predicate **new_anaphor**(CommonAnaphor) can have additionally a second argument that is a grammar variable of anaphor type and return accordingly a new anaphor name.

3.4.2 The problem of anaphoric connections inside an X-bar tree

Let's assume the following examples of anaphoric connections between an element of a sentence and a reflexive pronoun: (Theofanopoulou,1994)

- $John_i$ admires $himself_i$.
- John thinks that $George_i$ is $himself_i$.
- $John_i$ considers $himself_i$ to be the best.

All the above examples have anaphoric connection between the proper nouns 'John' and 'George' and the reflexive pronoun 'himself'. The problem is how it is possible to definine a general transformation rule that automatically puts the anaphoric connection between the two elements in the above examples.

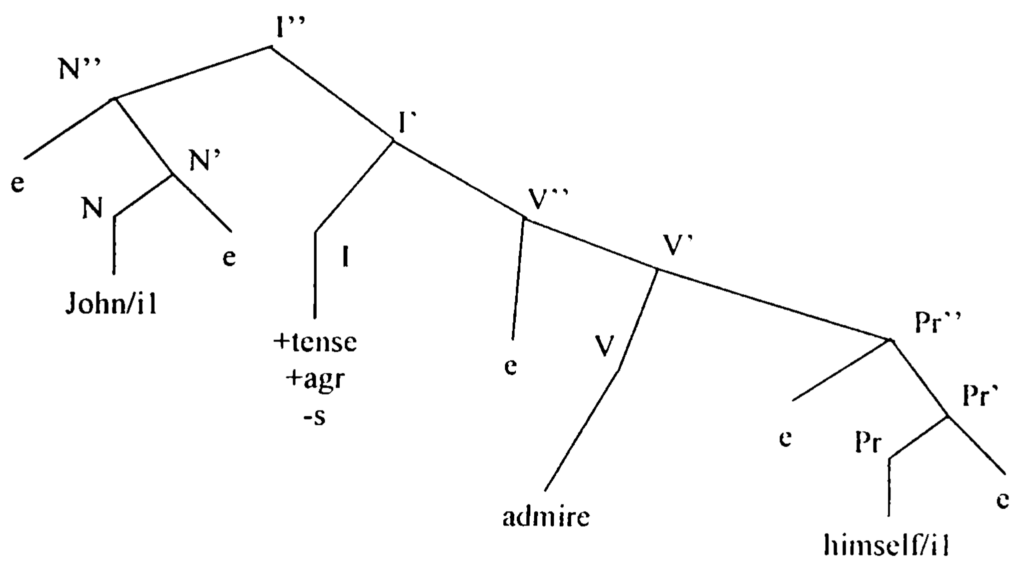
In the above examples it is noticed that the reflexive pronoun is connected with the closest proper noun.

In the first sentence the pronoun is connected with the proper noun. It is the simplest case.

In the second case the reflexive pronoun is possible to be connected with the closest proper noun that is the word 'George'.

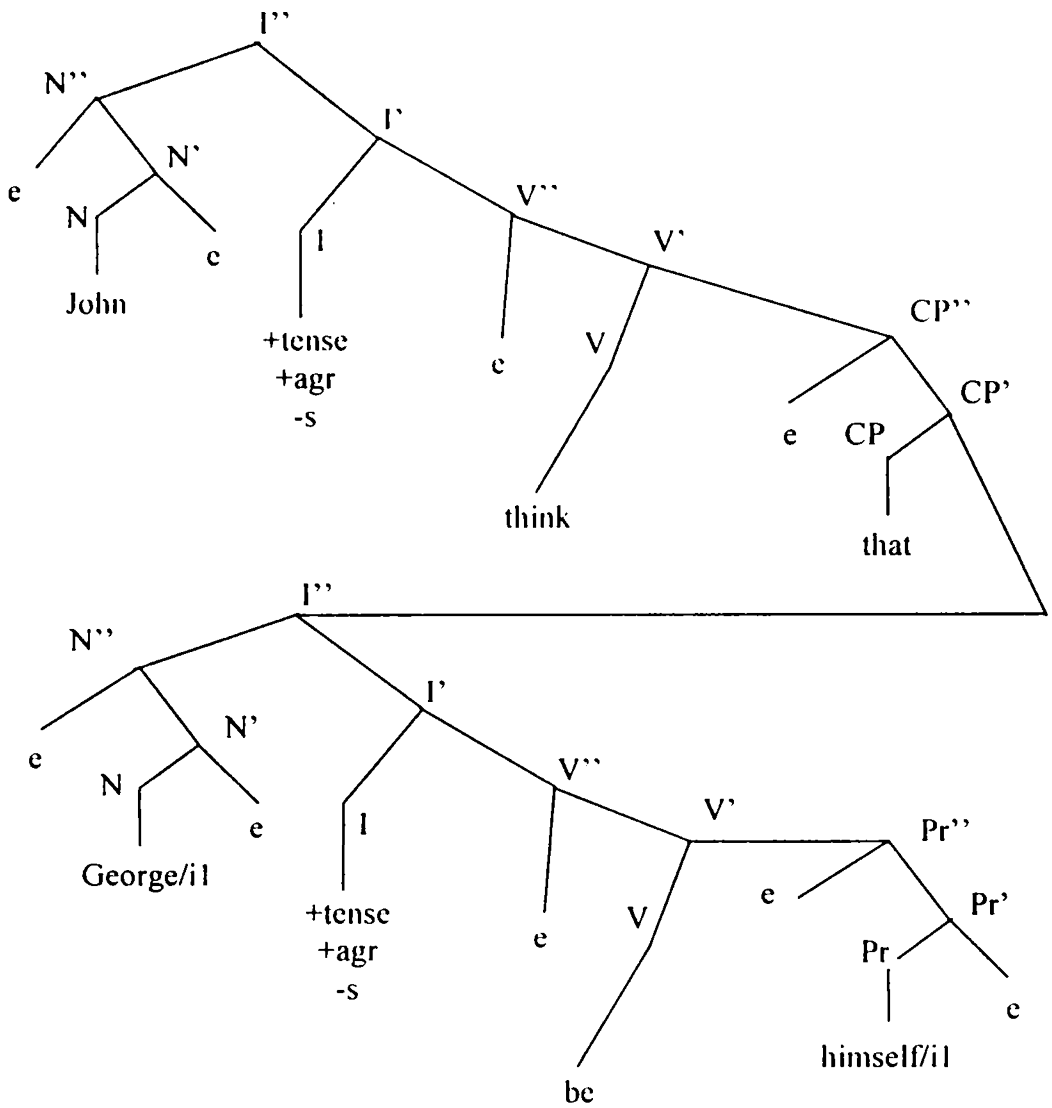
In the third case the connection is between the reflexive and the closest proper noun.

The first sentence is represented by the following tree:



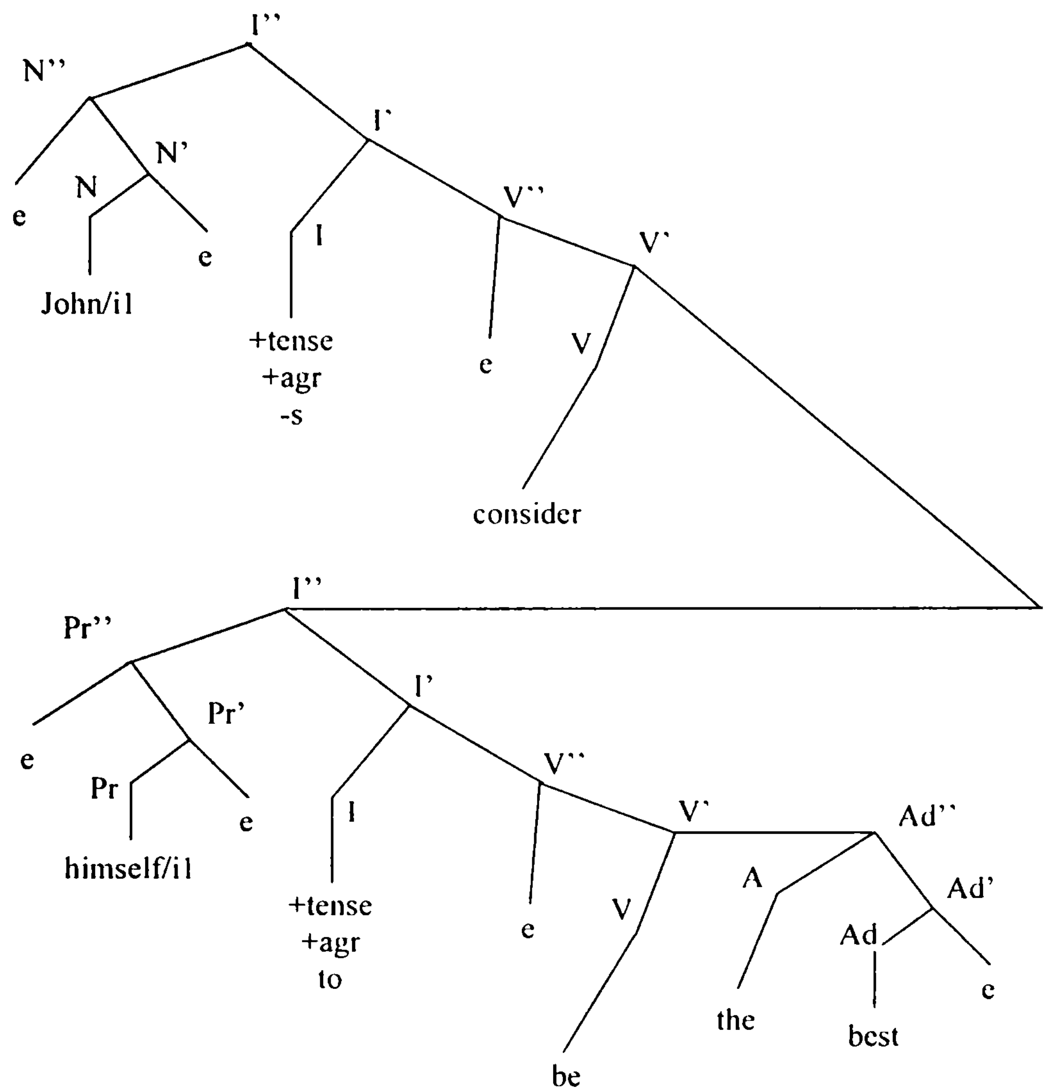
BUPT

The second sentence is reprersented by the following tree:



BUPT

The third sentence is represented by the following tree:



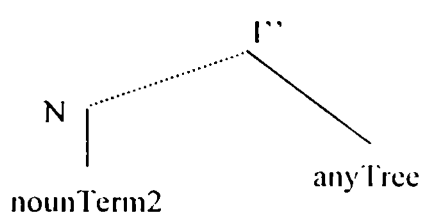
The transformation rule that covers the case of anaphoric connection of a reflexive pronoun with the corresponding noun is the following:

transformation 'Reflexive_Anaphoric_Connection'.

noVariables.

```
structureDescription
(node &node1,
  aTree ( node noun bar:features &feat_first,
          (terminal &nounTerm1): transformationVariable &tv1),
  ( not (node i barii,aTree (node noun bar, terminal &nounTerm2),
anyTree)
    and
    aTree ( node pr bar:features &feat_second,
            (terminal &pronounTerm): transformationVariable &tv2) )
).

structureCommands
 ifThen(&tv1 aCommon &tv2,fail),

 features man set [+human,+masculin],
 features woman set [+human,+feminine],

 ifThenElse(
        (&man subsets &feat_first and &man subsets &feat_second) or
        (&woman subsets &feat_first and &woman subsets &feat_second)

        (new_anaphor(CommonAnaphor),
         &tv1 addAnaphor CommonAnaphor,
         &tv2 addAnaphor CommonAnaphor,
         transformations      &tv1 transform &tv1 also
                              &tv2 transform &tv2
        ),
        fail)
```

This rule is the same as the same as the last one that was described in the previous section. The additional constraint in the **structureDescription** field is that the subtree (**node** i **barii,aTree** (**node** noun **bar**, **terminal** &nounTerm2), **anyTree**) is forbidden in the right subtree. It means that it is not permitted the another noun that is closest in the reflexive pronoun. This tree is as following:



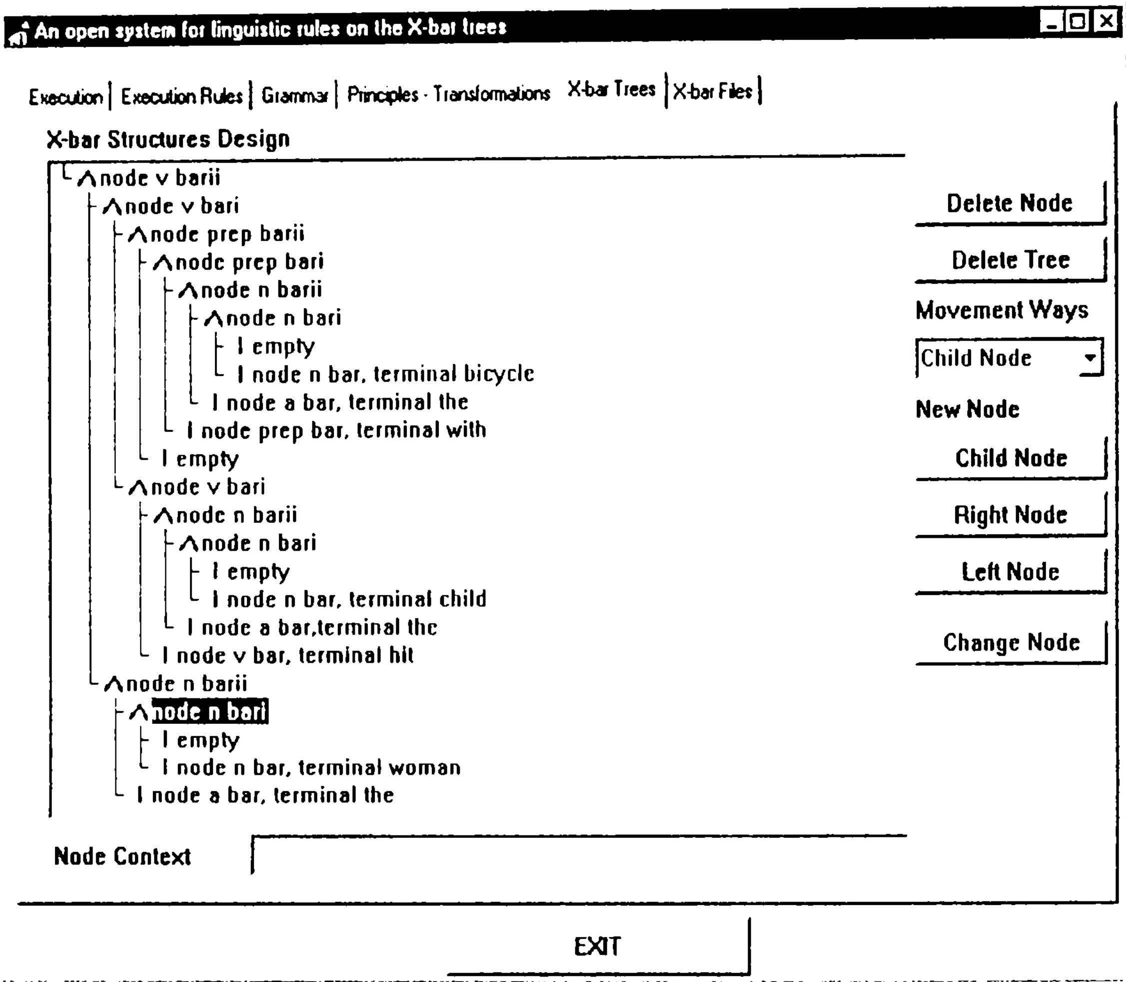
3.5 The graphical monitoring of the system

The linguistic knowledge of the system that we described in the previous chapters has a series of input-outputs. We can process these files with a text editor. However, in order to be able to change these files more easily and in an integrated environment, a system was implemented that runs in a window environment.

Generally we can say that this system comprises of a window with multiple tabs. In each tab we can process a different input or output. There is also on-line help that describes the system's function and the abilities provided by the system in each tab. Also, the help includes the rules and the commands of the system that we described in the previous chapters.

Next we shall analyze each tab of the system.

The first tab is for the design, the presentation and the alteration of the system's input trees.



The empty space on the above tab is the space where the tree is drawn.

In the bottom we see a button that has the name EXIT. When we press this button we exit the system.

In the bottom there is a field in which we write the content of the node that exists on the tree that we draw. This node is given with the name, the type and the possible features of the node, as well as with the possible anaphors of the subtree that has this node at the top. Also, we write the node of the X category together with the terminal element and the possible anaphors of the terminal element.

In the vertical column on the right of the empty space where the subtree is, we have a series of buttons.

The first one has the name Delete Node. His button deletes a tree node that we have already selected with the mouse, as well as the whole subtree that has this node at the top.

The second button has the name Delete Tree. This button deletes the whole drawn tree.

The third button has the name Child Node. This button sets a new node as a child node of the node that we have selected with the mouse.

The fourth button has the name Right Node. This button sets a new node as the right node of the one that we have selected. In the tree that we see, this node is above the node that we have selected.

The fifth button has the name Left Node. This button sets a new node as the left node of the one that we have selected. In the tree that we see, this node is below the node that we have selected.

The sixth button has the name Change Node. This button changes the content of the node that we have selected with the mouse and sets as new content the one that exists in the field Node Content.

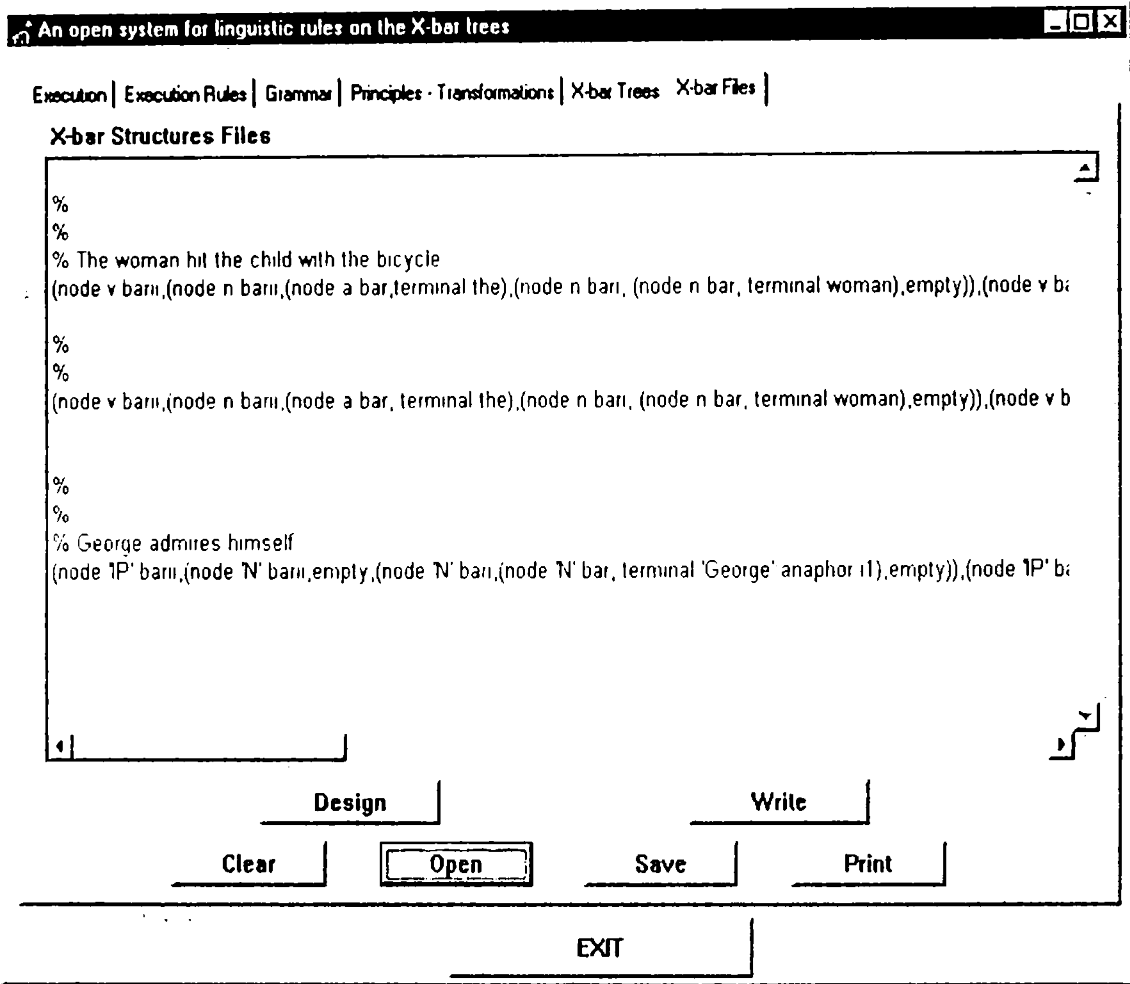
Finally, we have the ability to move a part of the tree that we have drawn. The move is made with the drag and drop method. For the way that the moved subtree will be connected, we have the following possibilities provided by the selector:

1. Child node
2. Left node
3. Right node

We must finally point out that each new child node takes the last place from all the children of the node. Also, if we move a node with the drag and drop method and set it as a child node of another node, then it takes the last place from the children of this node.

In the above window we can see this tab and a designed tree.

Next we shall wee the following tab. This tab processes the input tree files and the result output files of the system.



This tab has a series of buttons used in the processing of these files.

The button Design enables us to see in a graphically the tree that we have in this tab. In order to see a tree graphically this tree should be in one line and we should highlight it with the mouse. Next we press the Design Button.

We also have the Write button that writes on this tab's text editor the tree that's been drawn in the tree drawing tab that we described above.

Apart from these two buttons that are used for the trees, we also have a series of other buttons. These are:

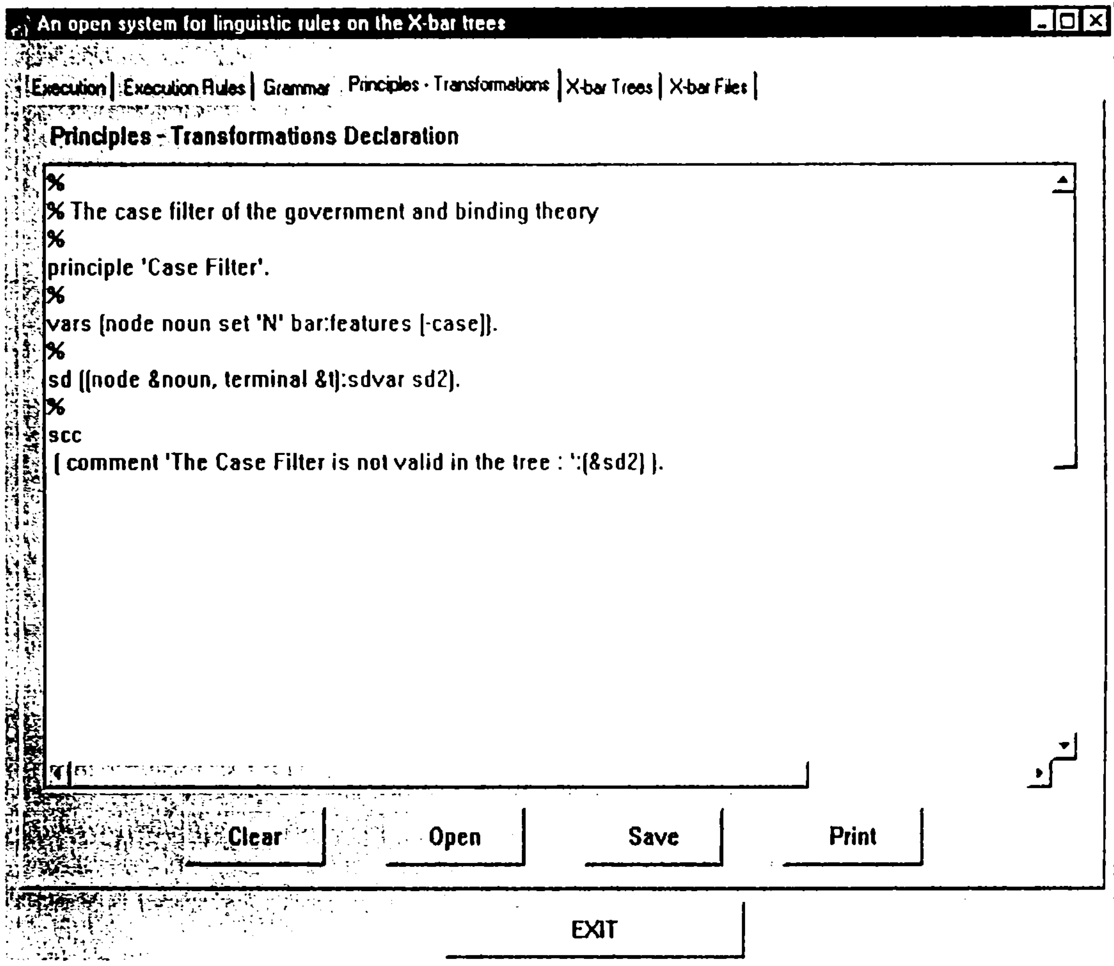
The Clear button deletes the whole content of this tab's text editor.

The Open button opens a dialogue window in which the user finds and downloads the desired file.

The Save button saves in the desired file the content of this tab's text editor.

The Print button prints the content of this tab's text editor.

Next we shall see the following tab in which we describe the several rules that constitute our theory.



In this tab we have a text editor that enables us to write the grammars we want.

In this tab we have a series of buttons that give us the following abilities:

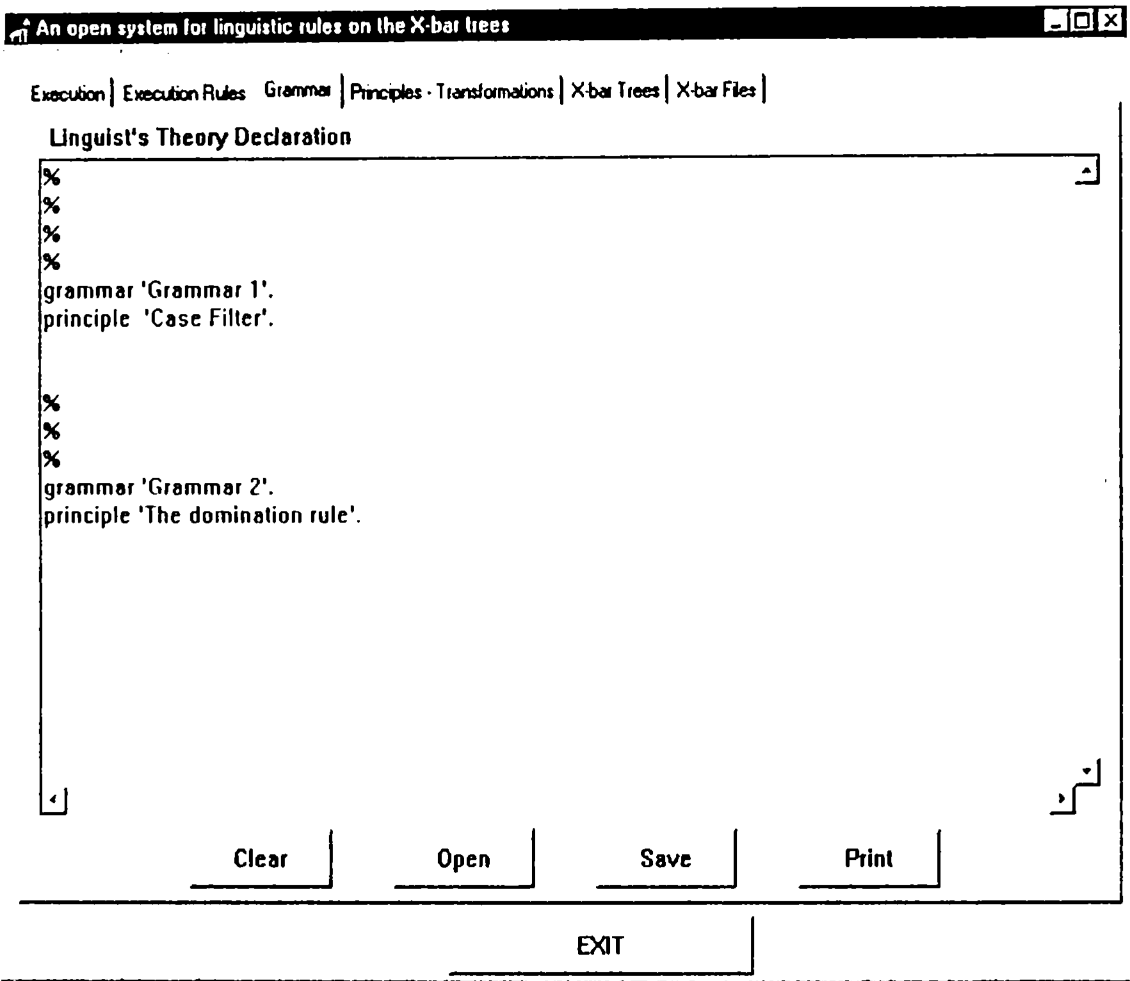
The Clear button deletes the whole content of this tab's text editor.

The Open button opens a dialogue window in which the user finds and downloads the desired file.

The Save button saves in the desired file the content of this tab's text editor.

The Print button prints the content of this tab's text editor.

Next we shall see the following tab in which we write the several principles and transformations.



In this tab we have a text editor that enables us to write the desired principles and transformations.

In this tab we have a series of buttons that give us the following abilities:

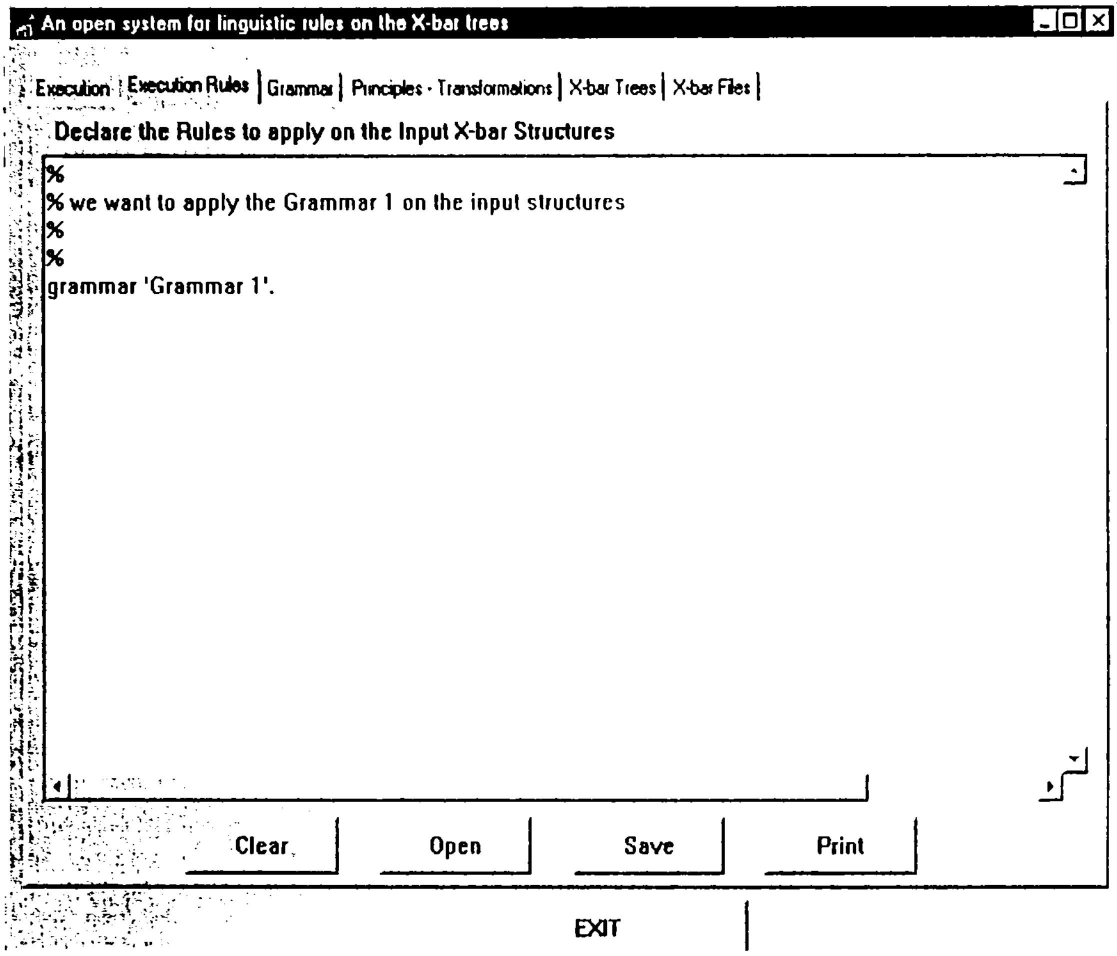
The Clear button deletes the whole content of this tab's text editor.

The Open button opens a dialogue window, in which the user finds and downloads the desired file.

The Save button saves in the desired file the content of this tab's text editor.

The Print button prints the content of this tab's text editor.

Next we shall see the following tab in which we write those grammars, principles and transformations that we wish to apply to the input trees.



In this tab we have a text editor that enables us to write the desired grammars, principles and transformations to apply on the input trees.

In this tab we have a series of buttons that give us the following abilities:

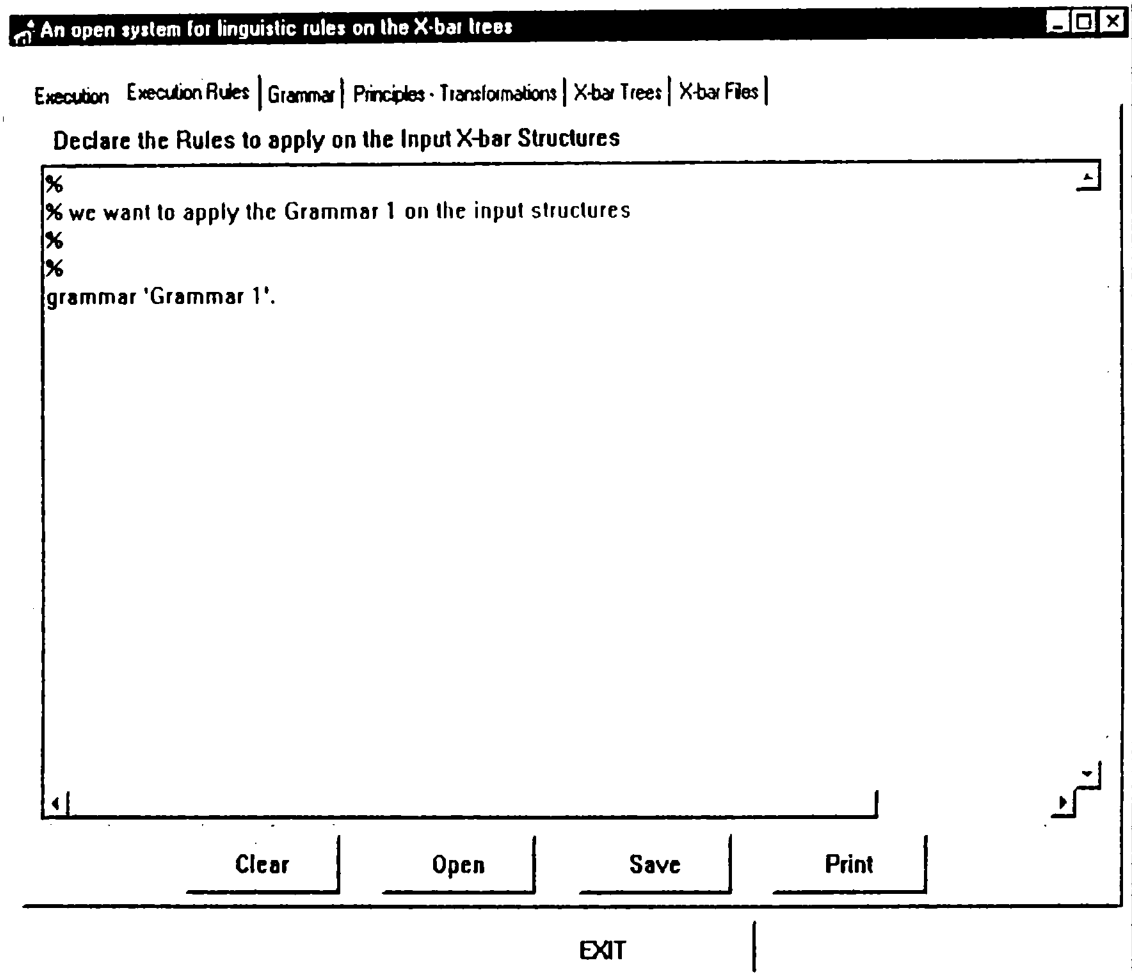
The Clear button deletes the whole content of this tab's text editor.

The Open button opens a dialogue window, in which the user finds and downloads the desired file.

The Save button saves in the desired file the content of this tab's text editor.

The Print button prints the content of this tab's text editor.

In the following tab we specify the input and output files that we wish the system to use and then we press the Execution button. The result is the production of a file in the subdirectory that we had selected the last time. This file has the name exec.ari and we download it in the prolog in order for the system to run.



This tab in its left part has a selector in which we can select the subdirectory that contains the desired file. Below there is the ability to select the appropriate filter for the files that we will see in the scrolling list that lies below.

In the right part there is a series of buttons. Each of them corresponds to a file. Thus, we select every time a file with the mouse and then we press the respective button to write it next to it. In this way we read all five files of the system. After doing this, we press the execution button that produces the file that the SWI-prolog needs to execute the linguistic system.

4. Conclusions

4.1 The existing computational methodologies

A. *The phrase-structure grammars*

They were presented mainly by Chomsky in 1957. They have the general form of x -> y, where x, y can be any combination of terminal and no-terminal elements.

The different categories of the phrase-structure grammars are the following:

- regular grammars:
    - left-linear grammars
    - right-linear grammars
- context-free grammars
- context sensitive grammars
- unrestriced grammars

These grammars are used in computational systems with different kinds of enhancements in order to produce or recognize natural language phrases. They are not restricted to specific tree structures and it is difficult to maintain and extend an application that uses this type of grammars. However the advantage of these grammar is that they have a very simple general format.

B. *Transition networks*

They are represented as finite states automatons. They are directed graphs with arcs noted by terminal elements. One node of the graph is denoted as starting point and another one as ending point. A sentence is accepted by the system if there is a path from the starting point to the ending point and its arcs contain the words of this sentence.

There are different kinds of transition networks :

- (STN) simple transition networks
- (RTN) recursive transition networks that are the same with the STNs but they additionally permit at their arcs phrasal categories except the lexical categories and recursions.
- (ATN) augmented transition networks that are RTNs with a set of registers for each network.

The disadvantages of these networks are:

- It is not possible to describe every grammar.
- The networks are very complicated.

- It is not possible to describe general rules for the different phrase categories in one network(Crystal, 1982). Usually, they are spread in many different networks.
- The check, the maintainance and the extension of these networks is very difficult.

The main advantage is that they have a simple general formalism that is possible to be implemented easily in prolog.

C. *Lexical functional grammar*

The basic characheristic of this grammar type is that the lexical records are declared as predicate structures with arguments. These structures are independent from the phrase structures and they are a form of functional comments for the lexical records. Also, there is the functional information of the phrase structures. This information is combined with the functional information of the lexical records and the final functional structure of a phrase is produced. The disadvantage of this theory is the only two functional equations between the functional information of the phrase structures and the lexical records. This sets restrictions on the declaration of rules.

D. *Generalized Phrase Structure Grammar*

This grammar type emphasizes on the information that the syntactic categories have. The internal structure of the syntactic categories is recognized.

The corresponding theory suggests the separation of the rules of syntactic structures in two categories:

- Rules of immediate dominace
- Rules of linear precedence

The first type refers to the hierarchical relation between different categories. The second type refers to the position that the different categories have in a sentence. This type of grammar is better for free order languages but it has been substituted by the newer model, the HPSGs. They do not support a specific tree structure and it is more difficult to extent an application or to declare reusable and general rules.

E. *Head-driven Phrase Structure Grammar*

This grammar type requires the existence of detailed morphological, syntactical and semantical information for every word. It requires more detailed information than the lexical functional grammars. This grammar is not a syntactical grammar but it combines both syntax and semantics. It organizes the linguistic knowledge as features structures. These features are sorted according to the specialization of them. Also, there is the possibility for paths that define the relation between them. The biggest difference between this theory and the previous ones is the way for the manipulation of the lexical records. Every representation requires very complicated information ànd there are very big problems for the maintainance of this huge information. Additionally, there is not any specific format and it is possible to have arbitrary different structures. It is not a good formalism for translation systems since the sing of the source and the destination language are not possible to be determined. That is way it is necessary for another semantic representation at this kind of systems.

4.2 The presented methodology

A computational system that implements the presented methodology is possible to be used as a tool by researchers. They can define rules and they can apply them on a set of X-bar trees. Additionally, it is possible to combine this with another software system that produces the X-bar trees. That system can use a set of very simple rewriting rules (even only lexical rules that produce all the X0 -> Terminal subtrees of the words of the phrases) for the production of the initial X-bar structures. The rules can be based only on general phrase structure information and they are the rules that were described in the corresponding section of the X-bar trees. The software that implements the presented methodology can be used in any natural language processing software system.

The main characteristics of the presented methodology are the followings:

- Is is an artificial computer language with variables, operators, if-then-else structures and repetitions-recursions dedicated in the natural language processing.

- It provides a mechanism for the declaration of rules that:
    - examines X-bar structures and rejects invalid ones.
    - transforms X-bar structures and produces new ones by permitting multiply simultaneous transformations on every X-bar structure.

- It manipulates the syntactic, semantic and pragmatic information of the X-bar structures. Additionally, it supports the checking of the accepted rate at a rule application and permits the evolutionary changing of the manipulated X-bar structures (Fouskakis, 2004c, 2005b).

- The syntactic and semantic information has simpler structure than the HPSG. The relation between the elements is determined and by the structure of the X-bar scheme. The variables have much stronger functionality with hierachical way of declaration that can change dynamically in the presented language which is better than in the unification grammars like the HSPG (Fouskakis, 2005a, 2005c) (the most interesting computational linguistics approach).

- The features of the nodes of X-bar structures can be changed dynamically by using transformations.

- It is more flexible than the TAGs (Fouskakis, 2005b).

- It is possible to define general rules that are applicable in many different X-bar trees since they are produced from the same general scheme and have the same structure and the same way of linguistic treatment (Fouskakis, 2004b, 2005b).

- It is according to the Chomsky ideas of the universal grammar theory, it combines his ideas in more general and abstract new approach (Fouskakis, 2004c) and it is unic in this sense (Fouskakis, 2000, 2005b).

- It is a different approach than the classical parsers that implement a version of the Chomsky's theory (GB-government and binding or Minimalistic Program).

- It is an artificial language that permits the declaration of natural language rules and its main characteristics are simplicity and generality.

- It supports anaphoric connections inside or outside of an x-bar structure.

- A better and simpler covering of the ambiguity problems of the phrases of natural languages by supporting more that one structures in the **structureDescription** field of the **principles** and **transformations** connected by the **and**, **or** operators and by using the variables (Fouskakis, 2005b).

- It integrates ideas from different theories.

- The simplicity, flexibility and generality facilitates the implementation, the maintenance and extension of the corresponding applications.

- It is better for embedded applications since the defined and produced structures are simpler and smaller and it is not necessary to have large memory size and strong processor.

- It facilitates the man-machine communication for the execution of commands and the retrieving of the required information that is expressed by natural language phrases. Possible applications can be in the domain of railway, airway or tourist information software systems. Also, it is possible to be used in the automotive domain to facilitate the communication with the today complicate information systems.

5. Bibliography

* Abercrombie D., 1967: *Elements of General Phonetics*, Ediburgh University Press.
* Alshawi H., Arnold D., Backofen R., Carter D., Lindop J., Netter K., Pulman S., Tsujii J., Uszkoreit H. and edited by Pulman G., 1992: *EUROTRA Rule Formalism and Virtual Machine Design Study-Final Report*, Cambridge, Cambridge Computer Science Department – Commission of the European Communities.
* Ananiadou S., Antona M., 1990: "Linguistic Opinions about the Multi-Language Translation from the Greek Language". Studies For The Greek Language proceedings of the 11 meeting of the linguistic department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Babiniotis G., 1980: *Theoretic Linguistics – An Introduction in Modern Linguistics*, University of Athens.
* Babiniotis G., 1985: *Introduction in Semantics*, University of Athens.
* Babko-Malaya O., 2004: "LTAG Semantics of Focus", Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms, Vancouver.
* Belletti A., Rizzi L., 2002: *Noam Chomsky On Nature and Language*, Cambridge: University Press.
* Campos H., 1987: "Passives in Modern Greek", Lingua, 73, 301-312.
* Cann R., 1993: *Formal Semantics An Introduction*, Cambridge University Press.
* Paris C., Swartout W., Mann W., 1991: *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, Kluwer Academic Publishers.
* Chomsky N., 1957: *Syntactic Structures*, The Hague: Mouton.
* Chomsky N., 1965: *Aspects of the Theory of Syntax*, MIT Press: Cambridge Mass.
* Chomsky N., 1968: *Language and Mind*, New York: Harcourt-Brace.
* Chomsky N., Halle M., 1968: *The Sound Pattern of English*, New York : Harper and Row.
* Chomsky N., 1970: "Remarks on Nominalisation", Jacobs R. - Rosenbaum P. 184-221.

* Chomsky N., 1972: *Studies in Semantics in Generative Grammar*, The Hague: Mouton.
* Chomsky N., 1976: *Reflections on Language*, London Fontana.
* Chomsky N., 1981: *Lectures on Government and Binding*, Dordrecht Foris.
* Chomsky N., 1982: *Some Concepts and Consequences of the Theory of Government and Binding*, Cambridge Mass: MIT Press.
* Chomsky N., 1986a: *Knowledge of Language. Its Nature, Origin and Use*, New York: Praeger.
* Chomsky N., 1986b: *Barries*, Cambridge: Mass MIT Press, Linguistic Inquiry Monograph 13.
* Chomsky N., 1988: "Some Notes on Economy of Derivation and Representation", MIT Working Papers 10.
* Chomsky N., 1995: *The Minimalist Program*, MIT Press.
* Chomsky N., 2000: *New Horizons in the Study of Language and Mind*, Cambridge: University Press.
* Copestake A., 2002: *Implementing Typed Feature Structure Grammars*, Standford: CSLIS.
* Cruse A., 2004: *Meaning in Language: An Introduction to Semantics and Pragmatics*, Oxford University Press.
* Crystal D., 1982: *Linguistic Controversies*, London Arnold.
* Drachman G., 1984: *Introduction to Greek Case*, University of Salzburg.
* Durand J., 1990: *Generative and Non-linear Phonology*, London: Longman.
* Dumitrescu D., 2002: *Principiile Inteligentei Artificiale*, Cluj Napoca:Editura Albastra.
* Efthimiou E., 1991: "Structural Substitution of Anaphoric Elements – a Case for Processing of Greek Language from the Computer", Studies For The Greek Language proceedings of the 12 meeting of the linguistic department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Fodor J., Katz J., 1964: *The Structure of Language. Readings in the Philosophy of Language*, Englewood cliffs: Prentice Hall.
* Fong S., 2000: "The Pappi System: Lexical Semantics and Morpho-Syntax", 38th Annual Meeting of the Association for Computational Linguistics, Hong-Kong.

* Fong S., 2005: "Computation with Probes and Goals: A Parsing Perspective", In UG and External Systems, Eds. Di Sciullo, A. M., John Benjamins Publishing Company.
* Fouskakis C., Halatsis C., 1997: "An Open System for Language Constraints on the X-Bar Trees", Working Papers in Natural Language Processing, An International Workshop, Athens Greece.
* Fouskakis C. 2000: "An Open System for Linguistics Rules on the X-Bar Trees", Ukrainian Journal of Computational Linguistics, Lviv Ukraine.
* Fouskakis C. 2004a: "A Computational Methodology for Linguistic Rules", Romanian-Hungarian International Conference SACI2004 on Applied Computational Intelligence, Timisoara Romania.
* Fouskakis C. 2004b: "An Overview of a Computational Approach for Linguistic Rules on the X-bar Trees", Development and Application Systems DAS2004 International Conference, Suceava Romania.
* Fouskakis C. 2004c: "The Organization of the Linguistic Knowledge in a Computational Methodology as Computer Language for Linguistic Rules", Symbolic and Numeric Algorithms for Scientific Computing SYNASC04 International Conference (University of West Timisoara and Reasearch Institute for Symbolic Computation from the Johannas-Kepler University of Linz-Austria), Timisoara Romania.
* Fouskakis C., 2005a: "A Computational Methodology as an Artificial Language for Natural Language Rules and The Unification Based Approach", Romanian-Hungarian International Conference SACI2005 on Applied Computational Intelligence, Timisoara Romania.
* Fouskakis C., 2005b: "The Basic Notions of the Tree Adjoining Grammars and a Methodology as Artificial Language about Linguistic Rules", Intelligent Linguistic Technologies Conference – World Academy of Science, Las Vegas Nevada USA.
* Fouskakis C., 2005c: "The Variables in the Computational Methodology as an Artificial Language for Linguistic Rules", 8th Intenational Conference – Computer Science Session, Oradea , Romania.
* Gabriilidou M., Lambropoylou P., Ronioth S., 1990: "Design and Commentary of a Greek Texts Corpus", Studies For The Greek Language proceedings of the 11

Meeting of the Linguistic Department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.

* Gazdar G., Klein E., Pullum G., Sag I., 1985: *Generalised Phrase Structure Grammar*, Oxford: Blackwell.
* Gilbert K., 1991: *Computer Processing of Natural Language*, Prentice Hall, Englewood Cliffs: New Jersey 07632.
* Grishman R., 1989: *Computational Linguistics an Introduction*, Cambridge: University Press.
* Haegeman L., 1990: *Introduction to Government and Binding*, Oxford: Blackwell.
* Haegeman L., 1995: *Introduction to Government & Binding Theory 2nd edition*, Blackwell.
* Halle M., Clements G., 1984: *Problem Book in Phonology*, Cambridge, Massachusetts and London, England: The MIT Press.
* Halatsis C., Stamatopoulos P., Karali I., Mourlas C., Gouskos D., Margaritis D., Fouskakis C., Kolokouris A., Xinos P., Reeve M., Veron A., Schuerman K., Li L.L., 1994: "MATOURA: Multi-Agent TOUrist Advisor", Proceedings ENTER'94, Insbruck.
* Halatsis C., Stamatopoulos P., Karali I., Bitsikas T., Fesakis G., Schizas A., Sfakianakis S., Fouskakis C., Coukoumpetsos Th., Papageorgiou D., 1996: "Crew Scheduling Based on Constraint Programming: The PARACHUTE Experience" , Proc. HERMIS'96, Athens.
* Haspelmath M., 2002: *Understanding Morphology*, Arnold Publishers.
* Horrocks G., Stavrou M., 1987: "Bounding Theory and Greek Syntax. Evidence for Wh-movement in NP", Journal of Linguistics, 23, 79-108.
* Jackendoff R., 1977: *The X-bar Syntax*, MIT Press.
* Jacobs R., Rosenbaum P., 1970: *Readings in English Transformational Grammar*, Xerox College.
* Wirth J., 1985: *Beyond the Sentence : Discourse and Sentential Form*, Karoma Publishers, Inc.
* Joseph B., Philippaki-Warbuton I., 1987: *Modern Greek*, London Routledge.
* Joshi A., Levy L., Takahashi M., 1975: "Tree Adjunct Grammars", Journal of Computer and System Sciences, 10(1), 136-163.
* Mckeown K., 1985: *Text Generation*, Cambridge University Press.

* Kosma D., Stratou S., Lolou A., 1988: "Analytic Grammar of the New Greek Language", Publications 2002 series Linguistic Library.
* Leech G., 1983: *Principles of pragmatics*, London: Longman.
* Lyons J., 1981: *Language, Meaning and Context*, London: Fontana.
* Mackridge P., 1985: *The Modern Greek Language*, Oxford University Press.
* Malikouti-Drachman A., Drachman G., 1988: "Accentuation in Greek", Studies for the Greek language, Thesaloniki Kiriakidis 127-144.
* Millett R., Lonsdale D., 2004: "Expanding Tree Adjoining Grammar to create Junction Grammars trees", Seventh International Workshop on Tree Adjoining Grammar and Related Formalisms, Vancouver.
* Noble H., 1988: *Natural Language Processing*, Blackwell Scientific Publications.
* Pedersen M. 2000: *Usability Evaluation of Grammar Formalisms for Free Word Order Natural Language Processing*, Queensland, University of Queensland:Thesis of Doctor of Philosophy.
* Philippaki-Warbuton I., 1970: *On the Verb in Modern Greek*, Bloomington Indiana University.
* Philippaki-Warbuton I., 1971: "Rules of Accentuation in Classical and Modern Greek", Glotta, 48, 107-121.
* Philippaki-Warbuton I., 1973: "Modern Greek Verb Conjugation. Inflectional Morphology in Transformational Grammar", Lingua, 32, 193-226.
* Philippaki-Warbuton I., 1975: "Passive in English and Greek", Foundations of Language, 13, 563-578.
* Philippaki-Warbuton I., 1976: "On the Boundaries of Phonology and Morphology. A Case Study from Modern Greek", Journal of Linguistics, 12, 259-78.
* Philippaki-Warbuton I., 1982: "Constraints on Rules of Grammar as Universals", Crystal D. 95-107.
* Philippaki-Warbuton I., 1985: "Word Order in Modern Greek", Transactions of Philological Society, 113-143.
* Philippaki-Warbuton I., 1987: "The Theory of Empty Categories and the PRO-drop Parameter in Modern Greek", Journal of Linguistics, 23, 289-318.
* Philippaki-Warbuton I., 1989: "Subject in English and Greek", Proceedings of the 3rd Symposium on the Description and/or Comparison of English and Greek, Thessaloniki Aristotle University School of English.

* Philippaki-Warbuton I., 1990: "Analysis of the Verb Set in New Greek", Studies For The Greek Language Proceedings of the 11 Meeting of the Linguistic Department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Philippaki-Warbuton I., 1992: *An Introduction in Theoretical Linguistics*, Nefeli, Athens.
* Photopoulou A., 1990: "Analysis of the Components of the Stereotype Sentences – Comments about their Classification", Studies For The Greek Language Proceedings of the 11 Meeting of the Linguistic Department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Radford A., 1981: *Transformational Syntax*, Cambridge University Press.
* Radford A., 1988: *Transformational Grammar*, Cambridge University Press.
* Radford A., 1997: *Syntax a minimalist Introduction*, Cambridge University Press.
* Ralli A., 1990a: "Lexical Phrase : Object of Morphological Interest", Studies For The Greek Language Proceedings of the 11 Meeting of the Linguistic Department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Ralli A., Touratzidis L., 1990b: "Computational Processing of Accents at New Greek Language", Studies For The Greek Language Proceedings of the 11 Meeting of the Linguistic Department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Ralli A., 1992: "The Theory of Features and the Inflectional Words Structure at New Greek Language", Studies For The Greek Language Proceedings of the 13 Meeting of the Linguistic Department of the Faculty of Philosophy of The Aristotle University of Thessaloniki.
* Roca I., Johnson W., 1999: *A Course in Phonology*, Blackwell Publishers.
* Samuel D., Norbert H., 1999: *Working Minimalism*, Cambridge: MIT Press.
* Schabes Y., Abeille A., Joshi A. K., 1988: "New Parsing Strategies for Tree Adjoining Grammars." In Proceedings, 12th International Conference on Computational Linguistics, 578-583.
* Schnelle H., Pierrot A., Hellwig P., Wiegand H., Naught J. Atkins B., Gross M., Calzolari N., Uszkoreit H., Schutz J., Caroli F., Devillers C., Rohrer C., Al B., Martin W. And Heid U., 1992: EUROTRA: *Feasibility and Project Definition Study an the Reusability of Lexical and Terminological Resources in*

Computerized Applications:Final Report, Stuttgart, University of Stuttgart Germany:EUROTRA study.

* Selkirk E., 1982: *The Syntax of Words*, MIT Press.
* Shaban M., 1994: A Minimal GB Parser, Boston, Boston University.
* Spencer A., 1991: *Morphological Theory. An Introduction to Word Structure in Generative Grammar*, Oxford: Blackwell.
* Staurou M., Philippaki-Warbuton I., 1987: "The Parameter of Reconciliation and the Independent Anaphoric Sentences", Studies for the Greek language, 311-322, Thesaloniki: Kiriakis.
* Tatar D., 2001: *Inteligenta Artificiala – Demonstrare Automata De Teoreme, Prelucrarea Limbajului Natural*, Cluj-Napoca: Editura Albastra.
* Tatar D., 2003: *Inteligenta Artificiala - Aplicatii In Prelucrarea Limbajului Natural*, Cluj-Napoca: Editura Albastra.
* Theofanopoulou D., 1986: "Structures for the Removal of the Object of a Complement", Studies for the Greek language, 87-108, Thesaloniki: Kiriakidis.
* Theofanopoulou D., 1989a: *Transformation Syntax from the Theory to Practice*, Kardamitsas: Athens.
* Theofanopoulou D., 1989b: "Compound Structures of NP and the Movement in Greek language", Studies for the Greek language, 337-354, Kiriakidis: Thesaloniki.
* Theofanopoulou D., 1994: *Transformation Syntax from the Theory to Practice II*, linguistics department of the Philosophy Faculty of the University of Athens.
* Vijay-Shanker K., Joshi A., 1988: "Feature Structure Based Tree Adjoining Grammars", in the proceedings of the 12th International Conference on Computational Linguistics, 714-719.
* Winston P., 1992: *Artificial Intelligence Third Edition*, Addison-Wesley publishing company.

INDEX

I

L

M

N

O

P

R

S

T

U

V

X