DOCTORAL THESIS IN COMPUTER SCIENCE

# AN EVALUATION SYSTEM FOR PARSING AND GENERATION ALGORITHMS

by

**Miroslav Martinović**

under the Guidance of

**Dr. Tomek Strzalkowski**
Courant Institute of Mathematical Sciences
New York University

Presented to
Mathematical Faculty
Faculty of Sciences and Mathematics

BELGRADE UNIVERSITY
October, 1992

# AN EVALUATION SYSTEM FOR PARSING AND GENERATION ALGORITHMS

Approved by Supervisory Committee:

------------------------------------

------------------------------------

------------------------------------

Copyright

by

Miroslav Martinović

1992

*To my Mother*

## ABSTRACT

This thesis addresses a challenging and significant problem of evaluating and ranking of different grammar processing algorithms (parsers or generators). Recently, with the advent of logic programming, which has been widely accepted as a paradigm for parser and generator design, it became possible to compare various competing approaches to the problem. The need for their evaluation has been felt strongly in both Linguistics and Computational Linguistics, since both have been thus far predominantly empirical, and it became difficult to measure the actual progress. This work sets up a formal apparatus for ranking grammar evaluation algorithms with respect to the following criteria: completeness, soundness, efficiency, optimality and reversibility. The method is based on the general principle of traversal of derivation trees, and is therefore independent of a particular grammar or execution strategy. It is also demonstrated how this formalism can be applied to evaluate specific algorithms, using as an example two well-known recent natural language generation algorithms. This work also rigorously defines the concept of logic grammars and a number of other related notions. Concepts lacking formal definitions, while used informally by many researchers, are formally and uniformly defined. Also, the equivalence of Definite Clause Grammars (DCG's) and type 0 Chomsky-an grammars, and DCG's and Turing machines is proven, using an original constructive method.

Approved by:

Tomek Strzalkowski

**Dr. Tomek Strzalkowski**

Thesis Advisor

Courant Institute of Mathematical Sciences

New York University

October, 1992

# PREFACE

The natural language processing (NLP) field has become one of the fastest growing and most exciting areas of applied Computer Science. Its challenges are numerous and various, and for a long time they were considered although extremely interesting, too complex to cope with in a satisfactory manner. Only recently with a number of new results, they became more approachable, which in turn, energized the research in the field, and showed potential new ways forward.

Phrase-structure grammars were dominant tools for description of languages, and their hierarchy, established by Noam Chomsky, was a framework for every research in the area, until late 70's and early 80's. Although theoretically complete, even for the class of artificial programming languages that had formal definitions, these grammars were either simple to implement and evaluate, but insufficient to describe all relevant features of a language (like context-free grammars), or, if they were sufficiently descriptive, they were hard to implement efficiently (as context-sensitive and type 0 grammars). Dealing with diversities and ambiguities of a natural language, for which it is often very difficult to propose a satisfactory mathematical definition, was a tedious and practically infeasible job within these formalisms, unless restricted only to a narrow subset of the language in question.

In parallel with the hierarchy of phrase-structure grammars and corresponding languages, a hierarchy of automata was introduced for recognizing these languages. Their relation toward the phrase-structure grammars was established as well, and one-to-one correspondence between classes of grammars and classes of automata was proven.

With the introduction of unification-based grammars, a new descriptive formalism was created that can as conveniently as previous ones be used for theoretical studies of language. However, unlike the phrase structure grammars and automata, this new formalism allowed various simple proof procedures to be used for the evaluation of the grammars, all due to the powerful mechanism of unification. The language description in it is independent and neutral toward the implementation. The descriptiveness of the unification-based grammars did not necessarily introduce high complexity to their implementation.

Although the equivalence of the new formalism in its generative power with the traditional ones (Turing machines, and type 0 phrase structure grammars) was proven, it was not easy to switch from one to another when needed. The reason for that was a lack of a constructive bridge between the traditional formalisms and the new one, which would make the transition from a description of a language in one to a description of the same language in another one automatic, and easy and natural to come up with.

This thesis contains two new formal proofs that definite clause grammars (DCGs) are equivalent in their generative power to Turing machines, and also type 0 grammars. Their significance is that they are both constructive, and therefore actually describing a procedure for transferring from a language description in one of the mentioned traditional formalisms to a DCG description. Taking into consideration the amount of work that was done using traditional formalisms, this thesis provides a method for their direct transfer and application to DCG's. This is, to the best of my knowledge a new and original contribution, and the only constructive bridge between Turing machines and DCGs, and type 0 grammars and DCGs. It is also simple enough to provide the people accustomed to describing formal languages through traditional formalisms, with a quick and natural switch to the unification-based ones.

In this work, DCG formalism is rigorously founded in a strict mathematical manner, and in that respect, this study represents a continuation of the work by V. Dahl and H. Abramson.

This thesis also introduces a formal system for evaluation of grammar-based linguistic algorithms. The criteria investigated here are, in my opinion, the most relevant when the algorithms are compared and waged one against another. They include generality of a grammar, and completeness, soundness,

efficiency, reversibility and finiteness for algorithms that process the grammar. The term "processing a grammar" is throughout this work used in the sense that it covers both, parsing and generation process for a given grammar.

The study first rigorously defines the forementioned criteria, offering sometimes different ways in which they can be understood and treated, and then provides the means for judging algorithms with respect to these criteria.

The judging of different algorithms is achieved here through analyzing the ways they traverse analysis trees produced by grammars. The notions like *tree*, *traversal of a tree*, etc. are rigorously defined, and then a relation (STAS) among different traversals is introduced. After proving important property that STAS relation is an equivalence relation, it is shown how it can be used to measure the completeness and reversibility of grammar-based linguistic algorithms.

For the efficiency criterion, this study offers another metric based on the number of edges traversed during the process of discovering an analysis tree by an algorithm. It proves an interesting property of tree-traversal algorithms that the average case analysis and worst case analysis are consistent (better in average case is better in worst case and vice versa).

Regarding finiteness criterion, we analyze the *guides' approach* by M. Dymetman, and we show that it can be viewed as a special case of the universal guides approach that is introduced in this work.

An example of how these criteria can be applied and used for a comparison between two algorithms is then presented. A semantic-head-driven generation algorithm developed by S.M. Shieber and others is compared with essential arguments algorithm described by T. Strzalkowski.

This work presents a new and unique formal system for evaluation of grammar-based linguistic algorithms. It also provides a foundation for a theoretical study of logic grammars and especially DCGs, that are rigorously defined here, and constructively connected to the traditional formalisms of Turing machines and type 0 phrase-structure grammars. It also presents a formal and complete framework for working with logic grammar processing algorithms.

# CONTENTS

# 1. PRELIMINARIES

## 1.1. Statement of Problem

The ongoing research on Natural Language Processing (NLP) involves the following four significant and distinct, but closely related areas of study:

(1) Investigation of the psychological processes involved in human language understanding;

(2) Building the computational systems for analyzing natural language input (and/or producing natural language output);

(3) Development of theories of natural language structures; and

(4) Determining the mathematical properties of grammar formalisms.

(1) and (2) build on and contribute to the work in area (3) and area (4) specifies rigorously the relations among different models of natural language processing. In this work, the relationship between issues in (2), (3) and (4) is studied, using recent results obtained from the research on reversible grammars and bidirectional natural language processing systems.

The idea behind the reversibility or bidirectionality is that the grammars are transducers between strings of words and some abstract meaning representations. Ideally, grammars should work in both directions: for analysis and for generation, but such bidirectionality has been until recently difficult to come by in practice. Recent significant results include parsing-generation algorithms (eg. [K90], [K84], [N89], [PS90], [S90a], [S90b], [S91], [SNMP89], [SNMP90], [W88]), as well as rigorous theoretical characterizations for symmetrical approach to parsing and generation (eg. [D90a], [D90b], [DI88], [DIP90]). With the new interest in the area, a need was recognized for developing and devising the means for ranking the proposed algorithms.

The purpose of this work is to

(i) establish rigorous formal criteria for comparisons of reversible grammar algorithms,

(ii) establish their hierarchy with respect to these criteria and

(iii) specify the conditions under which the algorithms will operate in an optimal fashion.

The relevant criteria, particularly significant in judging parsing and generation algorithms are: efficiency, generality, completeness, soundness, finiteness and directionality (or reversibility). In this thesis, these criteria are handled by analyzing the traversals of the analysis trees and counting the number of edges traversed by each algorithm. The directionality criterion primarily answers the question whether a particular algorithm can or cannot run an unification grammar in both, parsing and generation direction. For some of the algorithms (as it will be demonstrated for Essential Arguments Algorithm, by T. Strzalkowski), there might be even more than two directions in which they could be run. The application of our evaluation criteria is demonstrated in comparing Essential Arguments Algorithm (abbreviated here as EAA and introduced through [PS90], [S90a], [S90b], [S91]) with Semantic-Head-Driven Generation Algorithm (referred to as SHDGA and described in [SNMP89], [SNMP90]).

Parsing and generation algorithms can also be evaluated with respect to their ability to produce finite search spaces, in particular when enumerating all alternative solutions. Very often, a positive answer to these questions will depend on the fulfillment of some additional conditions imposed on the grammar in question. Depending on how limiting the conditions are, another hierarchy of the algorithms can be established.

## 1.2. Prior and Related Work

There is a natural appeal to the attempt to characterize parsing and generation in a symmetrical way, as well as to be able to use the same grammar for both. Although, the idea of reversibility is not new, a major attention to it has been paid only recently. The problem of reversibility can be described in the following manner: Given a grammar-like description of a language, that specifies both, its syntax and semantics, the aim is to obtain, by a fully automatic process, two possibly different programs: a parser and a generator. The parser would translate well-formed expressions of the source language into expressions of the language of semantic representation and the generator would accept well-formed expressions of the semantic representation language and produce corresponding expressions in the source natural language. The demand for practical algorithms that would run a grammar for both parsing and generation and the resulting software products is rapidly growing within computational linguistics, especially in the areas of machine translation and natural language communication with various information retrieval systems. A significant number of papers published and presented at major conferences addresses these problems.

### 1.2.1. Historical Overview

Only recently, a more serious attention started to be paid to the idea that a generator for a language might be constructed as an inverse of the parser for the same language. Although this idea has been around for some time, more concrete and more successful approaches to the problem coincided with the growing acceptance of applicative-style, machine-independent programming, most often as Lisp and Prolog programming ([PS87] and [GM89]). Classical imperative programming was based upon the use of machine-level side effects. In contrast, applicative-style programs have the character of formal specifications, which can be both executed and manipulated by other programs in order to change some of their mathematical properties. One of the important features of a program is the direction of computation.

In early eighties and before, the bidirectionality of natural language understanding process, although being desirable from a practical viewpoint, still remained hopelessly complicated within chiefly procedural systems, such as Robinson's DIAGRAM, which dominated the scene of natural language processing at that time ([R82]). Vastly superior in this respect are applicative systems, most notably those based on unification which, being an associative and commutative operation, lends itself naturally to inversion. Some better known formalisms for writing unification-based natural language systems are definite clause grammars (DCG), PATR and various others having their common ancestor in Colmerauer's metamorphosis grammars. These formalisms were introduced through [PW80], [S84] and [S86] and [C78], respectively.

#### *1.2.1.1. Applicative vs. Procedural Formalisms*

To make a comparison between applicative and procedural formalisms, in this study we will take as most representative examples of their respective classes, definite clause grammars (DCG's) and augmented transition networks (ATN's). A comparison between an applicative formalism (here DCG's), with a procedural one (here the formalism of augmented transition networks (ATN) from [B78]), often starts by relating their generative power. DCG's and ATN's as formal computational systems both have the power of a Turing machine and in that sense are as general as they could be. (The adequacy of DCG's for programming any computable task, without "coding" of data, is proved by Andreka and Nemeti (1976) in [AN76]).

In an ATN, it is impractical to build structures which do not closely mirror the recursive analysis of the string produced by the PUSH/POP mechanism. This is because a POP arc can only return a single structure and all of the subcomponents of this structure must be known at the time this POP is evaluated (as explained in [B78]). In an unification grammar, as in a DCG for instance, a non-terminal may return more than one structure as its result and these structures may contain variables which only later get a value. Thus, the structures generated in a DCG as the result of the analysis of a phrase may depend on items in the sentence which are outside the phrase concerned and possibly, not yet encountered in the parsing.

DCG's are more general and more flexible than ATN's because they can be used in a wider variety of ways. An ATN is a machine for parsing a language top-down, left-to-right, whereas a DCG is primarily a language description, neutral towards implementation. DCG's could be used by applying very different proof procedures (e.g., breadth-first rather than depth-first, bottom-up rather than top-down, etc.). Also, input to a DCG need not in principle be simple string consisting of atomic symbols. Symbols can be generalized to arbitrary tree structures (possibly with variables) and, more interestingly, instead of a simple list of symbols one can have a tree structure as an alternative in the input. In generation for example, semantics is represented by some kind of tree representation and objective is to discover a corresponding string for a given semantics.

Regarding the efficiency of both formalisms, let us note first that executing a DCG in Prolog gives a parsing mechanism, which can be described as "top-down, left-to-right, depth-first" and that is precisely the parsing mechanism used in the majority of ATN applications. DCG is expressed directly in a general purpose applicative programming language, Prolog. Therefore, DCG efficiency is equivalent to Prolog efficiency.

Warren and Pereira (1980) have described how Prolog can be compiled directly into efficient machine code. The speed of the produced code is comparable with that for more conventional high-level languages like Lisp. They also argued that pattern matching encourages better implementation of operations on structured data than the conventional use of selector and constructor functions (such as "car", "cdr" and "cons"). A practical implementation exists for the DECsystem-10 machine, as well as for many other systems and the actual timing data supports these conclusions ([PW80]). On the other hand, an ATN needs a special interpreter or compiler which generates a Lisp code. Thus, compilation of an ATN has two levels involving an intermediate high level language (Lisp). While ATN needs special access operations, in DCG's access to variable values is immediate and structure building is done by "structure sharing", at almost no extra cost. Automatic indexing provides for the immediate selection of appropriate alternatives in the DCG.

Unlike the ATN formalism, DCG can also be a useful formalism for theoretical studies of language. It could provide a bridge between the work of theoretical linguists and the work of those concerned with engineering practical natural language systems. That is because the theorists are usually concerned with describing what natural language is, in a clear and elegant way and details how language is recognized or generated may be of lesser importance to them. An ATN is always a description of a process for recognizing a language and only by extension a description of the language itself. DCG's could serve as both, as a description and also by virtue of the procedural interpretation of logic, as a process for analyzing the language.

Most of the characteristics of both, DCG's and ATN's, are shared by their respective classes of applicative and procedural formalisms. Therefore, on practical and philosophical grounds, applicative formalisms appear to represent significant advance over procedural ones.

### *1.2.1.2. Historical Overview of the Reversibility Problem*

The reversibility problem received some attention even before the applicative programming prevailed. The general problem of program inversion was addressed as early as 1956 by McCarthy ([M56]). Also, in 1983, Dijkstra considered the problem of permuted vectors and showed a two-way solution by manually deriving an inverse of the program that was written to solve the problem in one way only. No general case solution for the problem of invertibility was suggested by Dijkstra ([D83]). When logic programming emerged, Shoham and McDermott ([SM84]) discussed the invertibility of Prolog programs. A more recent work on directed predicates and data dependencies in logic programs is Debray ([De89]).

More recently, the problem of inverting a definite clause parser into a generator in the context of a machine translation system was addressed by Dymetman and Isabelle ([DI88]). They discussed two alternative solutions. The first one proposed a top-down interpreter with dynamic selection of AND goals (right-hand side goals). The interpreter is more flexible than, say, a left-to-right (LR) one. It can execute a given DCG in either direction depending only upon the binding status of arguments in the top-level literal. This approach, although conceptually quite general, proved far too expensive in practice, essentially because it is dynamic. The main source of overhead comes from employing the trick known as "goal freezing" (Colmerauer ([C82]), Naish ([N85] and [N86])). "Goal freezing" stops expansion of currently active AND goals until certain variables get instantiated. At the same time, other goals may become active, which could eventually lead to producing the required bindings and resuming of the "frozen" goals. Naish discussed a more advanced implementation in which the control information, in the form of implicit "wait" statements, could often be automatically generated from the static program specification. Similar techniques were also proposed by other authors, most notably in generation, by Wedekind 1988 ([W88]). Wedekind proposed reordering of AND goals by first generating nodes that are "connected", that is, those whose semantics is instantiated. Since the method is dynamic, the ordering of AND goals can, in principle at least, be different for different uses of the same rule. As Shieber, et al. in [SNPM89] pointed out, the inherently top-down character of goal-freezing interpreters might occasionally cause serious troubles during the execution of certain types of recursive goals, when there might be no bound on the size of a subcategorization list involved in the analysis, if the instantiation of it is essential for the resumption of the computation.

Shieber, et al. (1989) proposed another algorithm (semantic-head-driven generation (SHDGA)), that do away with the dynamic reordering of AND goals and replace it by a mixed top-down (TD), bottom-up (BU) interpretation. This algorithm will be analyzed in details here. In this algorithm, certain goals, whose expansion is defined by the so called "chain rules", are not expanded during the TD phase of the interpreter. Thus after generating the root of the AND/OR tree (analysis tree), the interpreter may skip a large portion of it, before generating a next node, somewhere down the tree. In the BU phase, the missing part of the tree will be filled in by applying the chain rules in a near backward manner. This technique proves to be less general and less efficient than EAA for instance, as it will be shown later in this study during the comparison of EAA (the essential arguments algorithm by T. Strzalkowski) and SHDGA. Its inefficiency is due to its indeterminism ([MS92]).

In another paper, Dymetman, Isabelle and Perrault ([DIP90]) characterized parsing and generation in a symmetrical way by introducing the notion of guides. The notion is applicable to both parsing and generation, but the instantiation is different in each case. Guides add redundancy to a definite clause program (in a form of new variables) that could be exploited for tighter control of computational process. After the guides are introduced for such programs and left recursion is eliminated by performing the usual transformation, a set of conditions can be specified that guarantees a finite search tree for any given query. These conditions are: guide consumption condition (GCC) and no-chain condition (NCC). It is shown that if both, GCC and NCC hold, then a finite search tree is guaranteed. These conditions are imposed on the lexical level predicates, for both parsing and generation. The main result is stated for a quite general class of definite clause programs. If we accept the following abbreviations:

| | |
|---|---|
| Program implementing the algorithm from [DIP90] | = MP |
| Definite Clause (Prolog) Program Implementing a Grammar | = DCPIG |
| DCPIG - input version (before processed by an algorithm) | = DCPIGi |
| DCPIG - output version (after processed by an algorithm) | = DCPIGo |
| Conservative Addition of Guide Unit | = CAGU |
| (unit from MP that adds guides to a grammar program) | |
| Left Recursion Elimination Unit | = LREU |
| (unit in MP that does away with left recursion), | |

the effect of the algorithm can be described by the following scheme:

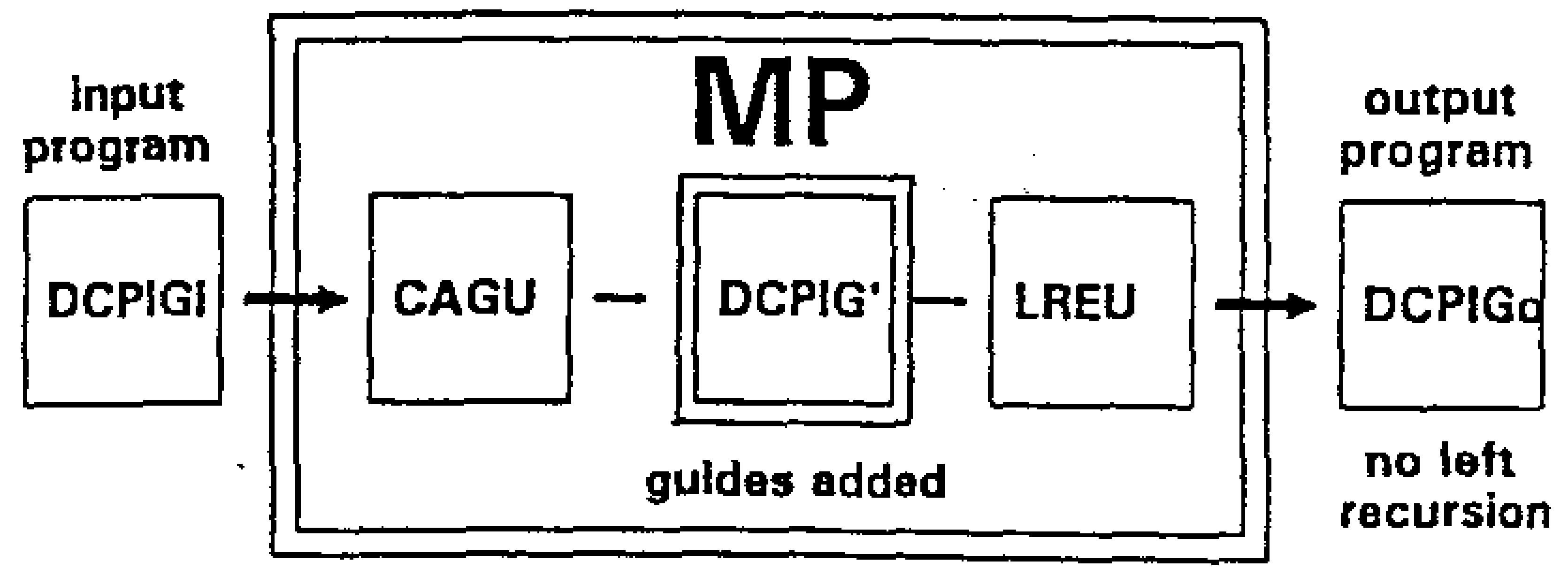


**Figure 1.2.1.2.1.:Implementation of Dymetman, et al.'s Algorithm**

Obviously, MP = CAGU + LREU. The formulation of the main theorem stated conditions to be fulfilled for DCPIG' in order for DCPIGo to have a finite search tree.

A similar characterization can be given to Strzalkowski's essential arguments algorithm (EAA) that is described in [S90b]. Let us assume the following abbreviations:

| | |
|---|---|
| EAA's program | = EAAP |
| Normalization Unit | = NU |
| (unit from EAAP doing away with left recursion) | |
| Efficiency Enhancing Unit | = EEU |
| (unit from EAAP performing lifting transformation and production splitting in EAAP) | |
| Reversing Unit | = RU |
| (unit from EAAP executing EAA) | |

EAAP = NU + EEU + RU,

The program implementing EAA algorithm (EAAP) can be described by the following scheme:

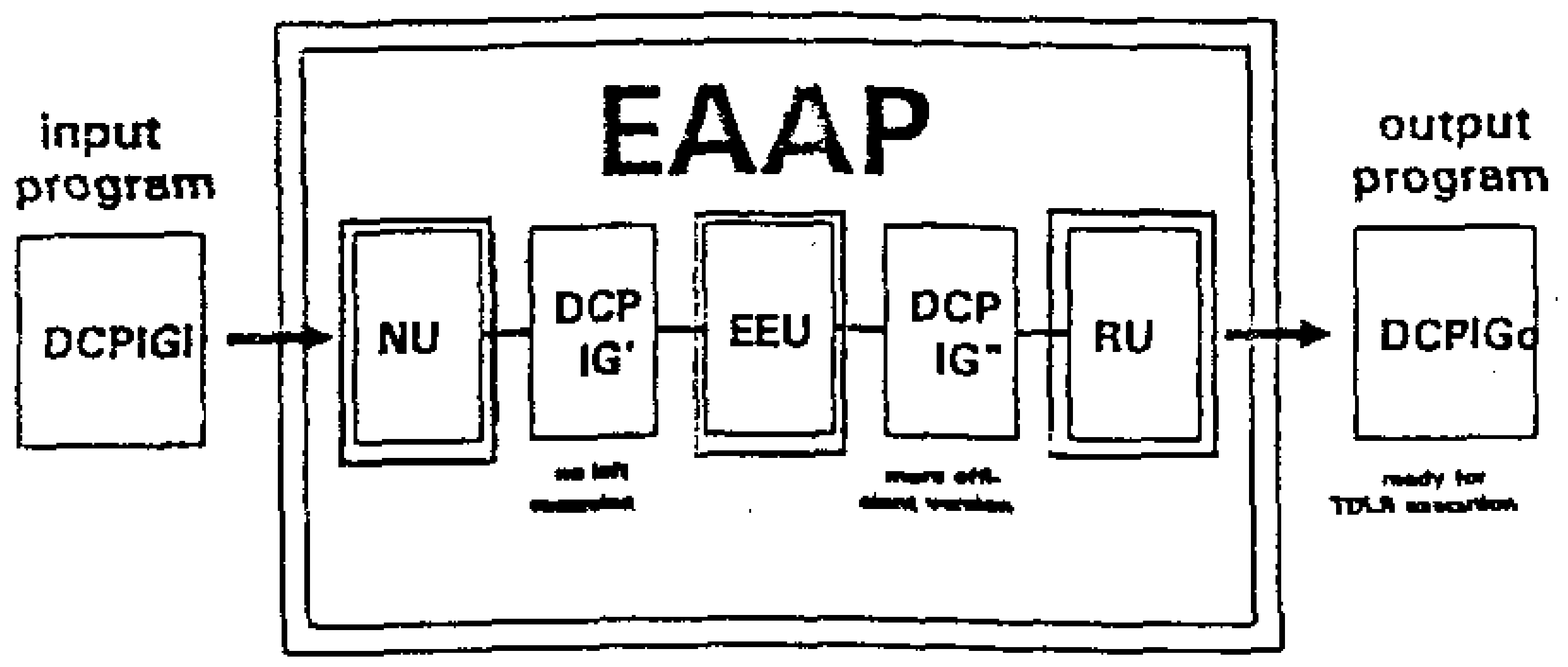


**Figure 1.2.1.2.2.:Implementation of Essential Arguments Algorithm**

Thus, both algorithms, do away with the left recursion. After a DCPIGi was processed by EAAP, there are no left recursive rules left in the output DCPIGo.

The EAA stops either after all initially uninstantiated variables in the main predicate of the query get their values (SUCCESS), or when that cannot be done (FAILURE). As it will be shown in this study, the proper guides for EAAP are lists of all initially uninstantiated variables. As the execution of EAA progresses, their number is being decreased (consumption of the guide). Unlike in [DIP90], the guide here can be consumed at all levels, not only at the lexical level. Both ideas are theoretically applicable to any possible direction of a computation, not only to parsing and generation. The study by Dymetman, Isabelle and Perrault however provided the powerful idea of guides, that, as it will be shown in this study, can be generalized and exploited for the purpose of establishing hierarchies of parsing-generation algorithms (with respect to the level of restrictiveness of the conditions on guides guarantying finiteness of their respective search trees).

### 1.2.2. Advantages of Bidirectional Natural Language Processing Systems

Practical advantages that we may expect from a bidirectional system beyond its mathematical properties are numerous. Perhaps the best summary of the arguments for adopting the bidirectional natural language processing (NLP) systems were given by Strzalkowski in [S90b]:

"(a) A "de facto" bidirectional NLP system, or a system whose inverse can be derived by a fully automated compile-time process, greatly reduces the effort required for system development, since only one program or specification is needed instead of two. This is especially true if creating the single specification does not require a substantial extra effort as compared to one-directional design. The actual amount of savings ultimately depends upon the extent to which the NLP system is made bidirectional (for example, how much of a parser can be inverted for generation).

(b) Using a single specification (a grammar) underlying both the analysis and the synthesis process leads to more accurate capturing of the language. Although no NLP grammar is ever complete, the grammars used in parsing tend to be too "loose", or unsound, in that they frequently accept various ill-formed strings as legitimate sentences.

This becomes immediately visible when the parser is run in the reverse on the representations it produces. The grammars used for generation, on the other hand, are usually made too "tight", as a result of limiting their output to the best surface forms. It is clear that such grammars are dramatically incomplete for parsing. A reversible system for both parsing and generation requires a finely balanced grammar which is sound and as complete as possible. However, the one-to-one correspondence between natural language expressions and internal representation formulae, argued for by some authors, is not necessary or even expected. Writing a balanced grammar puts more pressure upon the linguist, but, it should also be noted, a reversible grammar can serve as a powerful debugging tool in building such systems.

(c) A reversible grammar (or language specification, in general) provides, by design, a match between a system's analysis and generation capabilities, which is especially important in interactive systems. A discrepancy in this capacity may mislead the user, who tends to assume that what is generated as output is also acceptable as input and vice versa (this latter is especially true in machine translation). While this match can often be achieved in nonreversible systems where the parser and the generator are constructed independently of each other, it comes at the cost of a substantial effort.

(d) Finally from the computational viewpoint, as pointed out by Kay in 1984 (...)" (here, [K84]) "(...), a bidirectional system can be expected to be more robust, certainly easier to maintain and modify and altogether more perspicuous." (pp. 146).

### 1.2.3. Interpretations of the Problem of Bidirectionality

There are different ways that the problem of bidirectionality is understood and defined. Perhaps most widely shared approaches are the following three implementations of bidirectionality:

(a) The language specification (most often a grammar) is compiled into two separate programs of a parser and a generator, which do not share the same evaluation environment, that is, each requires a specific evaluation strategy that may not be applicable to the other.

(b) The parser and the generator are separate programs but they must be executable using the same evaluation strategy.

(c) There is only one program implementing both the parser and the generator and the evaluation strategy can handle it by running it in both directions.

As pointed out earlier, the applicative systems, especially those based on unification, proved to be very suitable for language processing. Logic programming (also known as definite clause programming) proved to be very suitable for NLP, with relatively simple evaluation strategies and powerful unification-based computation. The programming language Prolog, offers an elegant implementation of logic programming using a simple depth-first, left-to-right (DFLR) evaluation strategy. Now, the statements (a), (b) and (c) can be understood in a narrower sense. In (a), we translate the language specification into two separate logic programs that may require different evaluation strategy in execution. In (b), we require that both the parser and the generator are executable by the same evaluation strategy. In particular, the strategy might be DFLR and thus that both are correct Prolog programs. The difference between (b) and (c) is that in (b) we adjust our programs to a specific evaluation strategy and try to adjust it so as to accommodate perhaps conflicting evaluation requirements when the same program is used once as a parser and another time as a generator. It may be noted that (b) and (c) are not so much distinct approaches as just two variants of the same general paradigm located near the opposite ends of an entire spectrum of midway options. The option closer to (b) assume existence of a cheap, but usually restrictive evaluation strategy, such as DFLR strategy employed by Prolog, which restricts the set of bi- and multi-directional programs to a small, uninteresting set. Thus, separate programs are often needed for computation in reverse. As we move away from (b) towards (c), the restrictions upon the evaluation strategy are relaxed and the class of bidirectional programs grows, but so does the cost of the computation. If the parser for a language is written in Prolog, then the question becomes: can the Prolog program of the parser be reversed to become a generator? That is exactly the question taken up by T. Strzalkowski in [S90a]. This same question can be posed at the level of the grammar, so long as all relevant information is available about the evaluation strategy to be used. This is usually true for the DCG's supplied with most Prolog implementations, as well as certain other variations of them ([C78], [PW80], [DA84], among others).

### 1.3. Justification for and Significance of the Study

There is no doubt that the field of NLP is one of the fastest growing and most exciting areas of applied Computer Science.

Logic programming and the logic programming language Prolog have proven to be especially appropriate for addressing different problems in NLP. A wide variety of unification formalisms have been developed for various specific uses, including building language analyzers, compiler writing, modelling of linguistic theories, etc., etc.

This study provides a foundation for a theoretical study of logic grammars and especially DCG's. DCG's are rigorously defined and constructively connected to the traditional formalisms of Turing machines and type 0 phrase-structure grammars. This makes all work done within the traditional formalisms automatically transferable to the formalism of logic grammars and bridges the gap between them.

Many recently published algorithms (including the forementioned EAA and SHDGA) address the problem of automated parsing and automated generation of natural language expressions from a structured representation of meaning. They are very often parts of some ongoing machine translation (MT) projects. The development of reversible grammar (RG) systems is considered desirable and important in MT systems, because the direction of the computation in them changes very frequently. RG systems can be immediately used for both parsing and generation and they reduce the development and maintenance effort. Their soundness and completeness of linguistic coverage, as well as their match between the analysis and synthesis capabilities are very significant in any linguistic system and particularly in MT. Bringing these algorithms together and being able to judge their efficiency, generality, directionality and optimality for a given evaluation strategy in an uniform and methodical way, proves to be a very reasonable and also a challenging task. Although motivated by analysis for the existing algorithms, these criteria will influence the creation of the new still better methods.

Bringing principled theoretical approaches to bear effectively on practical applications regarding a natural language in all its "spoken" version's width and diversity remains another great challenge. It is nevertheless believed that the more recent works in this area show ways forward.

## 1.4. Methodology

Major components of this thesis would be establishing rigorous formal criteria for comparisons of parsing-generation algorithms, establishing their hierarchy with respect to these criteria and specifying the sets of conditions under which, the algorithms will operate in an optimal fashion with respect to the given criteria. In particular, these criteria are applied on two recent promising generation algorithms: EAA and SHDGA. It is envisaged that the metrics introduced in this work, will be usable for future work in this field, both to judge and promote the creation of new improved algorithms of the kind.

Therefore, this work essentially comprises of:

(I) Specifying

* the efficiency criterion based on the analysis of traversals of derivation trees for different algorithms. The criterion takes into consideration:

(i) optimality of traversals (whether or not the algorithm finds the optimal path during the search),

(ii) number of edges traversed in order to find the optimal path (although maybe finding the optimal path, the algorithm might do it by traversing a lot of edges, very few of them, or sometimes no wrong edges whatsoever). This criterion for DFLR search strategy is equivalent to the amount of backtracking;

** the generality criterion based on the analysis of traversals of derivation trees for different algorithms. This criterion considers:

(i) completeness of the algorithm, or number of solutions that a particular algorithm generates (an algorithm can generate only one, some but not all, or possibly all solutions),

(ii) soundness of the algorithm (the algorithm might accept (or generate) some incorrect language constructs as legal).

(II) Establishing the algorithms' hierarchy with respect to * and **, above.

(III) Specifying

*** the finiteness criterion based on the analysis of traversals of derivation trees for different algorithms.

Here, the task is to find a set of conditions sufficient and (possibly) necessary, under which a given algorithm will produce a finite search tree and therefore will not run into an infinite loop. The set of conditions is established relatively to an evaluation strategy under which the algorithm will be executed.

(IV) Establishing the algorithms' hierarchy with respect to ***.

This comparison is based on the restrictiveness of the conditions in ***.

(V) Demonstrating the application of (I) to (IV) to EAA and SHDGA.

## 1.5. Organization of the Thesis

This study presents a formal system for evaluation of grammar-based linguistic algorithms. The criteria considered here are generality of the underlying grammars and completeness, soundness, efficiency, reversibility and finiteness of the algorithms for evaluation of grammars.

Chapter 1 outlines historical prospective by giving a survey of significant and relevant results from the field.

Chapter 2 provides a rigorous formal founding of the field of logic grammars and related notions discussed later in the following chapters. It presents a unique mathematical system providing a groundwork for putting the whole theory into a more formal frame, subjecting it directly to the powerful apparatus of mathematics.

Chapter 3 contains two original constructive proofs for the equivalence of definite clause grammars (DCG's) and Turing machines and DCG's and type 0 grammars. The equivalence of these formalisms was known before, however, the study provides two algorithms for moving from a DCG to a Turing machine or a type 0 grammar and in that respect, represents an original contribution.

Further chapters talk about criteria for the estimation of different grammar algorithms. They introduce a unique and original system intended for this purpose and demonstrate it on a comparison of two grammar-based algorithms.

Chapter 4 introduces formalization of most important criteria for estimation of logic grammars and grammar processing algorithms: generality, completeness, soundness, efficiency, reversibility and finiteness, explaining along the way, some other notions that are included in these, or include them.

Chapter 5 introduces STAS relation, a relation applicable to the elements from the set of traversals of derivation trees. It proceeds with two important theorems that justify the introduction of the relation and prepare it for its use for the evaluation of the criteria from the previous chapter.

In Chapter 6 we argue the suitability of the STAS relation for the evaluation of grammar processing algorithms with respect to the completeness and reversibility criteria and suggests how it can be used in this context.

In Chapter 7 we discuss soundness criterion. We present a theorem that shows how a general definition as the one given in this study can be used in practice to evaluate the soundness of a grammar processing algorithm.

In Chapter 8 we discuss the issue of efficiency of grammar processing algorithms. Consistency between worst and average case analysis is proven for logic grammar processing algorithms and quantities WCAN, ACAN and BCAN as efficiency estimates are advocated.

Finiteness criterion is being dealt with in Chapter 9. Apart from the so called "guides approach" to this very complex issue, the notion of universal guides is introduced as an alternative approach to this criterion judgment.

Chapter 10 presents a comparison of two recent and very promising algorithms. Their properties with respect to the previously mentioned criteria are being analyzed and they are compared and ranked.

# 2. BASIC CONCEPTS

This chapter introduces a number of definitions that will be used throughout the rest of this work. The main motivation is the lack of rigorous formal definitions for a significant number of these notions, even though some of them are frequently used. By formalizing these notions, the powerful apparatus of mathematics can be used for proving facts about them. Some of the notions have been already known and some of them were even rigorously defined in some previous work. However, we believe that this system of definitions is unique and new as a system and provides a groundwork for putting the whole theory for evaluation of grammar-based parsing and generation algorithms into a more formal frame, making the math apparatus directly applicable to it.

Before we move to the more specific sections, we try to make clear certain terminology that we use and that should make connection between the terminology used by Chomsky for phrase structure grammars and terminology from the theory of logic grammars.

DEFINITION 2.1. [based on [DW83]] (ALPHABET)

A set $A=\{a_1,\ldots,a_i,\ldots\}$ is an alphabet, iff for any element $a_j$ of A the following holds: there is no sequence $a_{k1},\ldots,a_{km}$ of elements from A such that $a_j$ is obtained by concatenation of $a_{k1},\ldots,a_{km}$∎

In other words, no element from an alphabet A can be obtained as a sequence of some other elements from the same alphabet. The elements of an alphabet are sometimes referred to as *letters.*

DEFINITION 2.2. [based on [DW83]] (WORD)

Let $A=\{a_1,\ldots,a_i,\ldots\}$ be an alphabet. A word (or a string) with respect to the alphabet A is any sequence $a_{k1}\ldots a_{km}$ of elements from A∎

A special word consisting of zero letters is introduced and it is called *empty word.* The usual notation for empty word is $\varepsilon$. Set of all words over an alphabet A (including $\varepsilon$) is denoted by $A^*$ and set of all non-empty words over A by $A^+$.

DEFINITION 2.3. [based on [DW83]] (LANGUAGE)

Let $A=\{a_1,...,a_i,...\}$ be an alphabet. A language over the alphabet A is any set L such that $L \subseteq A^*$ ∎

The remainder part of present chapter consists of sections on Turing machines and type 0 grammars, two traditional and well founded formalisms. The following section then introduces logic grammars, derivation trees and related notions. We will also attempt to establish the theory of logic grammars in a fashion similar to the theory of Turing machines and Chomsky's non-logic grammars.

## 2.1. Turing Machines

The definitions from this and next section are based on work on Turing machines and Chomsky's hierarchy of phrase structure grammars from [DW83] and [HU79].

DEFINITION 2.1.1. [based on [HU79] and [DW83]] (TURING MACHINE)

Turing machine $\mu$ is an ordered quintuple ( **A**, **Q**, **P**, $q_1$, $q_0$). $\mathbf{A} = \{a_1,...,a_m\}$, is an *external alphabet* (whose elements $a_i$ we usually refer to as *letters*), $\mathbf{Q} = \{q_0,q_1,...,q_n\}$ is an *internal alphabet* (whose elements $q_j$ we call *internal states* of the machine). $\mathbf{P} = \{I_1,...,I_p\}$ is a finite set called the *program* of the machine. Its elements $I_r$ ($1 \leq r \leq p$) are ordered quintuples ($q_i$, $a_j$, $a_k$, M, $q_l$), where $q_i$ and $q_l$ are internal states, $a_j$ and $a_k$ are letters or blank symbols and M is an element from the set { N, L, R }. We refer to the elements of **P** as *instructions*. $q_1$ and $q_0$ are two distinguished elements from **Q**, called *initial* and *final state*, respectively ∎

Set { N, L, R } is chosen as it is, in order to associate to "No move", "Left move" and "Right move", in accordance with the most common interpretation of Turing machines, that we explain next.

The most common interpretation of a Turing machine is a tape infinite in both directions, with cells of the same size such that each can hold a character from **A**, or a blank symbol. Only a finite number of cells contain a nonblank character, at any moment. Symbols can be read or written on the tape with a movable "head" that can access one cell at a time. We say that the head is *observing* or *over* a cell. Such a device is also at any moment at one of the internal states q ($\in$ **Q**). The following picture should be helpful:

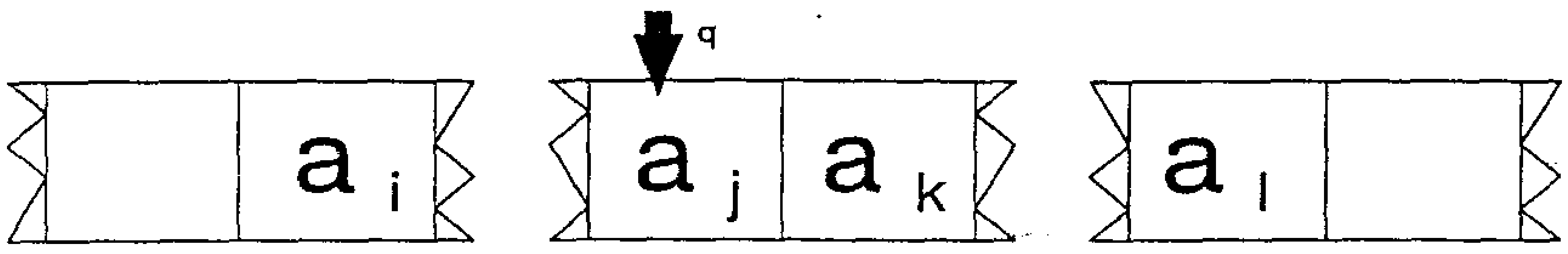


**Figure 2.1.1.: An Interpretation of Turing Machines**

A Turing machine is considered to start processing a word **w** = $a_i a_j..a_k$, if it starts from the following initial configuration:

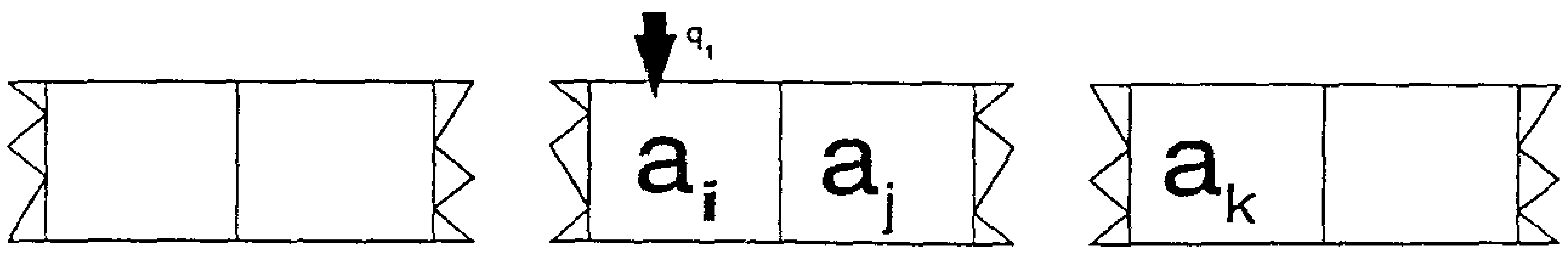


**Figure 2.1.2.: Initial Configuration for a Turing Machine**

Therefore in the initial state, the letters of the word w are in consecutive tape cells, the remaining cells being blank and the head is *observing* (*over*) the cell that contains the first letter of w. Accordingly, the first instruction to be executed is the one that starts with $q_1a_i$. This instruction is of the form $q_1a_i$-->$a_lMq_k$. If M was R the configuration of the machine after applying the instruction to the machine would be the following:

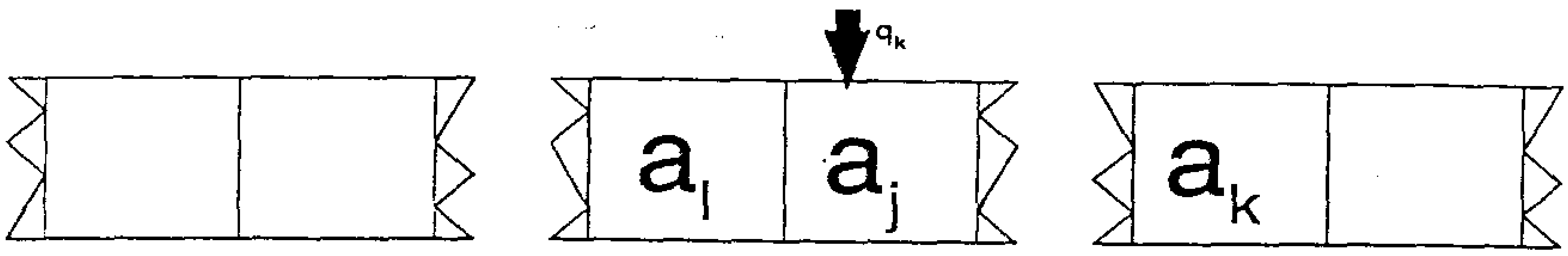


**Figure 2.1.3.: A Possible Next Configuration (Right Move)**

If M was L, the following configuration would emerge:

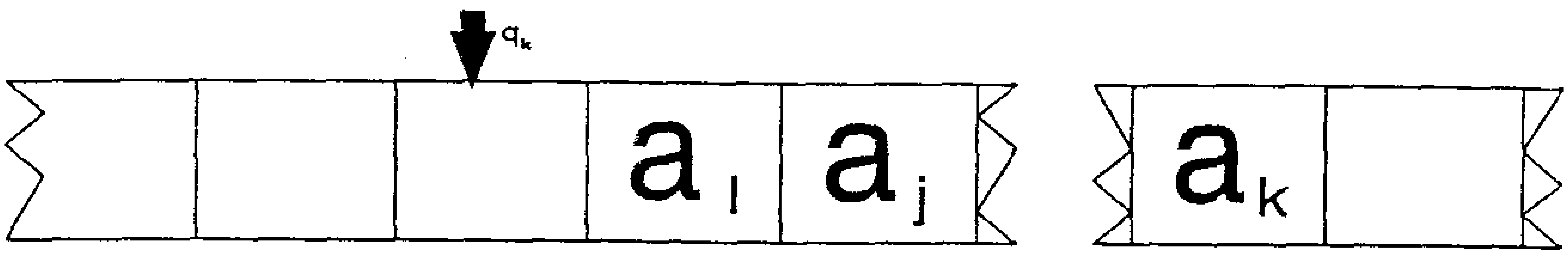


**Figure 2.1.4.: Another Possible Next Configuration (Left Move)**

And finally, if M was N, the following configuration would emerge:

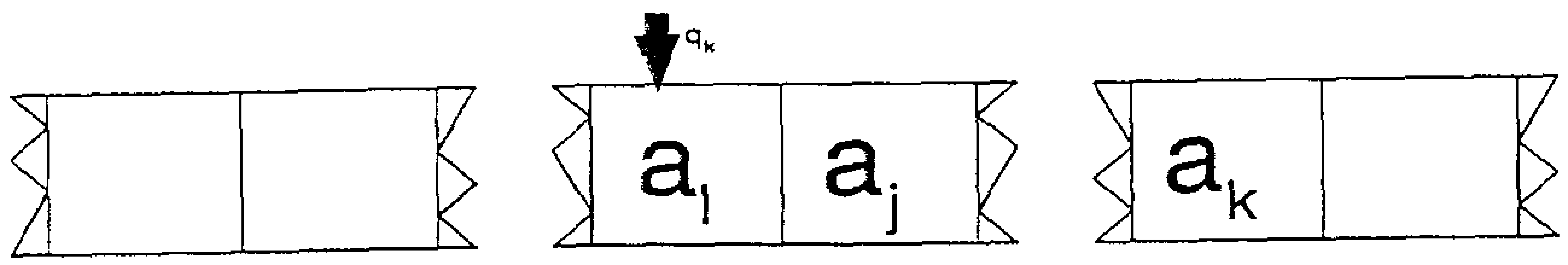


**Figure 2.1.5.: Another Possible Next Configuration (No Move)**

L, R, N stand for left, right and no move, respectively. Depending on the configuration of the machine (which state and *observing* which symbol), the next instruction for execution can be chosen. If there is no instruction beginning with the current state and currently observed symbol, the machine stops its execution and it is said to **reject** the word **w** that was initially on the tape. If at any point during the execution the machine enters the final state $q_0$, it is said to be **accepting** the word **w**, no matter what the content of the tape is and what the position of the head is.

Our next step is to formally define *a computation* of a Turing machine.

DEFINITION 2.1.2. [based on [HU79] and [DW83]] (INSTANTANEOUS DESCRIPTION)

Let $\mu$ = ( **A**, **Q**, **P**, $q_1$, $q_0$ ) be a Turing machine and *A* and ***A*** the following two alphabets: *A* = **A** $\cup$ {  } and ***A*** = *A* $\cup$ **Q** $\cup$ { $^*$ } (Here    stands for a blank symbol). An instantaneous description for the Turing machine $\mu$ is a word of the form $^*A^*qA^{**}$, where $q \in$ **Q**.

*A*$^*$ is the set of all words made up of zero or more *letters* from *A*.

An instantaneous description will be used for the description of a *current configuration* of a Turing machine in the following manner: The position of the state q in the word simulates the position of the head of the machine. Whatever follows q and precedes the asterix symbol, is considered to be to the right of the head, plus infinitely many blanks to the right of those symbols. An exception is the first letter after q, which is considered to be exactly in the currently *observed* cell. What is on the left of q is considered to be left of the head and also infinitely many blanks, on the left from the first symbol after the left asterix. For instance, the following configuration:

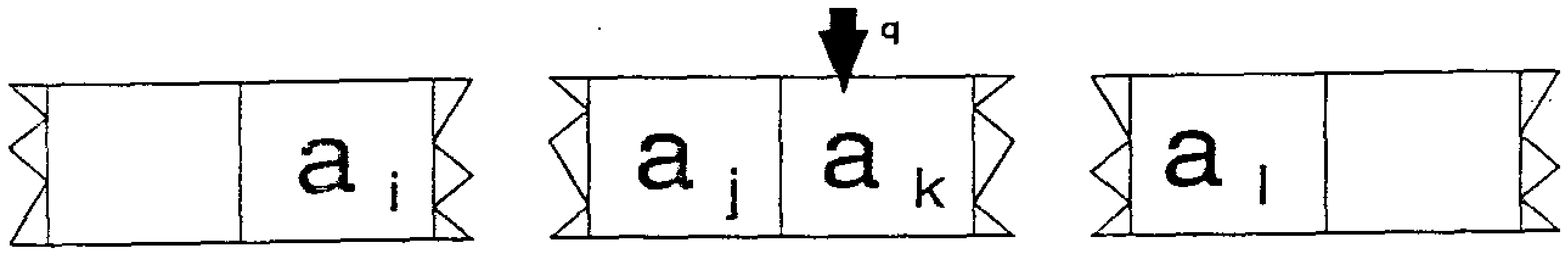


**Figure 2.1.6.: A Configuration and an Instantaneous Description**

would be represented by the instantaneous description $^*a_i..a_jqa_k..a_l{}^*$. This provides a necessary formalism for defining a *Turing machine computation.*

DEFINITION 2.1.3. [based on [HU79] and [DW83]] (TURING MACHINE TRANSFORMATION)

Let $I_j$ = $(q_i,a_j,a_k,M,q_l)$ be an instruction of the Turing machine $\mu$ = ( **A**, **Q**, **P**, $q_1$, $q_0$ ), (where **A** = $\{a_0,..,a_m\}$, **Q** = { $q_0$, $q_1$,.., $q_n$ } and **P** = { $I_1$,..,$I_p$ }). The following transformations of instantaneous descriptions are corresponding (are assigned) to $I_j$ depending on the value of M :

(i) If M=L, $T_i$ = $^*q_ia_j$ |--> $^*q_l\Box a_k$ and

(ii) $T_{ii}$ = $a_mq_ia_j$ |--> $q_la_ma_k$.

(iii) If M=N, $T_{iii} = q_ia_j \mid\!\!\dashrightarrow q_la_k$.

(iv) If M=R, $T_{iv} = q_ia_j{*} \mid\!\!\dashrightarrow a_kq_l\square{*}$ and

(v) $T_v = q_ia_ja_m \mid\!\!\dashrightarrow a_kq_la_m$∎

An execution of an instruction could therefore be seen as an application of one of the corresponding transformations. The portion of each transformation to the left of symbol I--> is usually referred to as *left side of the transformation* and the portion on the right of I--> as *right side of the transformation.*

DEFINITION 2.1.4. [based on [HU79] and [DW83]] (DERIVATION BY A TURING INSTRUCTION)

Let $\alpha$ and $\beta$ be two instantaneous descriptions for a Turing machine $\mu$ = ( **A**, **Q**, **P**, $q_1$, $q_0$ ). Let I be an instruction from **P** and S set of its corresponding instantaneous description transformations (S={$T_i$,$T_{ii}$}, or S={$T_{iii}$}, or S={$T_{iv}$,$T_v$}). We say that $\beta$ can be derived from $\alpha$ by using Turing instruction I and write $\alpha$ -I-> $\beta$, iff

(a) $\alpha$ contains left side of a transformation from S (${*}q_ia_j$ or $a_mq_ia_j$, when S={$T_i$,$T_{ii}$}; $q_ia_j$, when S={$T_{iii}$}; or $q_ia_j{*}$ or $q_ia_ja_m$, when S={$T_{iv}$,$T_v$}) and

(b) $\beta$ is the same as $\alpha$ except that the mentioned left side of the transformation in question is rewritten in $\beta$ as its corresponding right side from the same transformation (${*}q_l\square a_k$, or $q_la_ma_k$ when S={$T_i$,$T_{ii}$}; or $q_la_k$ when S={$T_{iii}$}; or $a_kq_l\square{*}$ or $a_kq_la_m$ when S={$T_{iv}$,$T_v$})∎

DEFINITION 2.1.5. [based on [HU79] and [DW83]] (DERIVATION BY TURING MACHINE)

Let $\alpha$ and $\beta$ be two instantaneous descriptions for a Turing machine $\mu$ = ( **A**, **Q**, **P**, $q_1$, $q_0$ ). We say that $\beta$ can be derived from $\alpha$ by using Turing machine $\mu$ and write $\alpha$ -$\mu^*$-> $\beta$, iff either

(a) $\beta$ can be derived from $\alpha$ by using a Turing instruction I ($\alpha$ -I-> $\beta$), or

(b) There exists an instantaneous description $\gamma$ such that $\gamma$ can be derived from $\alpha$ by using a Turing instruction I ($\alpha$-I->$\gamma$) and $\beta$ can be derived from $\gamma$ by using Turing machine $\mu$ ($\gamma$-$\mu^*$->$\beta$)∎

In other words, there exist instantaneous descriptions $v_1,..,v_{p+1}$, such that $v_1=\alpha$, $v_{p+1}=\beta$ and $v_i$-$l_i$->$v_{i+1}$, for each i=1,..,p and corresponding $l_i$ (possibly different for different i's) from the program of the Turing machine $\mu$.

DEFINITION 2.1.6. [based on [HU79] and [DW83]] (WORD ACCEPTED BY A TURING MACHINE)

A word w ($\in A^*$) is said to be accepted (or recognized) by a Turing machine $\mu$ iff $*q_1w*$ -$\mu^*$-> $\sigma$ ($q_1$ is initial state) and the instantaneous description $\sigma$ contains the final state $q_0$ ($\sigma=*\alpha q_0\beta*$ & $\alpha, \beta \in A^*$)∎

DEFINITION 2.1.7. [based on [HU79] and [DW83]] (LANGUAGE ACCEPTED BY A TURING MACHINE)

Set of all words from $A^*$ recognized by the Turing machine $\mu$ is called language accepted (or recognized) by $\mu$ and it is denoted as **L(μ)**∎

Thus, L($\mu$) = { w$\in$**A**| $*q_1w$-$\mu^*$->$*\alpha q_0\beta*$ & $\alpha, \beta \in A^*$ }.

## 2.2. Type 0 Grammars

Type 0 phrase structure grammars (in Chomsky's hierarchy) are the next formalism to be presented.

DEFINITION 2.2.1. [based on [HU79] and [DW83]] (PHRASE STRUCTURE GRAMMAR)

Grammar G is an ordered quadruple ( N, T, **P**, S ), where N $\cup$ T = ***A*** is the alphabet ***A*** of a language, **P** is set of the ordered pairs of words from $\boldsymbol{A}^*$, called production rules and S is a distinguished element from N, called the starting symbol of the grammar∎[1]

[1] The difference between type 0 grammars and better known context-free grammars (CFG's), which are also known as type 2 grammars in Chomsky's hierarchy, is in an additional restriction imposed on the form of the production rules of CFG's. In CFG's, a left-hand side of a production rule must be a non-terminal symbol (element of N), while in type 0 grammars, it can be any symbol from $A^*$.

DEFINITION 2.2.2. [based on [HU79] and [DW83]] (DERIVATION BY A PRODUCTION RULE)

Let G=( N, T, **P**, S ) be a type 0 grammar and $N \cup T = \boldsymbol{A}$ be alphabet of a language. The word $\beta \in \boldsymbol{A}^*$ is said to be derived from the word $\alpha \in \boldsymbol{A}^*$, by the use of the production rule $\gamma$-->$\delta$ from **P**, iff $\alpha=\xi_1\gamma\xi_2$ and $\beta=\xi_1\delta\xi_2$, where, $\xi_1$ and $\xi_2$ are the words from $\boldsymbol{A}^*$.∎

The usual notation for this is $\alpha$-$^{G}$->$\beta$.

DEFINITION 2.2.3. [based on [HU79] and [DW83]] (DERIVATION IN A GRAMMAR)

Let G=( N, T, **P**, S ) be a type 0 grammar and $N \cup T = \boldsymbol{A}$ be alphabet of a language. The word $\beta \in \boldsymbol{A}^*$ is said to be derived from the word $\alpha \in \boldsymbol{A}^*$, in the grammar G (written as $\alpha$-$^{G^*}$->$\beta$), iff, either

(a) word $\beta \in \boldsymbol{A}^*$ is derived from the word $\alpha \in \boldsymbol{A}^*$ by the use of a production rule $\gamma$-->$\delta$ from **P** ($\alpha$-$^{G}$->$\beta$), or

(b) word $\gamma \in \boldsymbol{A}^*$ is derived from the word $\alpha \in \boldsymbol{A}^*$ by the use of a production rule from **P** ($\alpha$-$^{G}$->$\gamma$) and word $\beta \in \boldsymbol{A}^*$ is derived from the word $\gamma \in \boldsymbol{A}^*$ in the grammar G ($\gamma$-$^{G^*}$->$\beta$).∎

In other words, there exist words $\delta_1$, $\delta_2$,...,$\delta_n$ such that: $\delta_1$-$^{G}$->$\delta_2$-$^{G}$->..-$^{G}$->$\delta_n$, and $\delta_1=\alpha$ and $\delta_n=\beta$.

DEFINITION 2.2.4. [based on [HU79] and [DW83]] (LANGUAGE RECOGNIZED BY A GRAMMAR)

Grammar G=(N,T,S,P) is said to recognize or generate language L iff L = { $w \in T^*$| S-$^{G^*}$->w }.∎

The set { $w \in T^*$| S-$^{G^*}$->w } is usually denoted as L(G).

Usual notation for grammar symbols uses words that start with a lower case letter for terminals, words that start with a capital letter for non-terminals, greek letters for words that consist of both, terminals and non-terminals and last six lower case letters (u,v,w,x,y,z) for words made up of terminals, only.

## 2.3. Logic Grammars and Derivation Trees

In this section we present basic concepts of the theory of logic grammars. We start with our definition of *tree*, that is actually a combination of several known definitions of the same notion. We consider this definition very suitable for our later introduction of *derivation trees.*

DEFINITION 2.3.1. (TREE)

Let V be a set of elements called vertices (or nodes) and $E \subseteq \{(r_1,r_2) | r_1,r_2 \in V\}$ a set of ordered pairs of elements from V, called *edges.* Tree **T** is an ordered quadruple of sets (V,E,Root,Subtrees), where $Root \subseteq V$ and either:

(a) $V=\emptyset$, $E=\emptyset$, $Root=\emptyset$, $Subtrees=\emptyset$ (case "empty tree"), or

(b) $V=\{r\}$, $E=\emptyset$, $Root=\{r\}$, $Subtrees=\emptyset$ (case "one-vertex tree"), or

(c) $Root=\{r\}$ and $Subtrees=\{(S_t^1,...,S_t^n)\}$, where $S_t^1,...,S_t^n$ are trees, such that:

(i) For $1 \leq i \leq n$, if $S_t^i=(V^i,E^i,\{r^i\},Subtrees^i)$, then $V^i \subset V$, $E^i \subset E$ and $(r,r^i) \in E$ and

(ii) For $1 \leq i,j \leq n$, if $S_t^i=(V^i,E^i,\{r^i\},Subtrees^i)$, $S_t^j=(V^j,E^j,\{r^j\},Subtrees^j)$ and $i \neq j$, then $V^i \cap V^j=\emptyset$ and $E^i \cap E^j=\emptyset$■

The element r (part (b) and (c) from the previous definition) is called *root* of the tree T. Accordingly, the elements $r^i$, $r^j$ are *roots* of $T^i$ and $T^j$, respectively.

The point (i) in the previous definition actually ensures that there are edges from the root r of the entire tree to the roots of the subtrees and point (ii) that the subtrees are disjunct.

If there are infinitely many vertices and edges in the tree, tree is called *infinite*, otherwise it is called *finite.*

The following concept of logic grammar symbols and the related concepts that precede it, are based on work done by V. Dahl and H. Abramson in [DA89] and they present an effort to make the concepts more formal and along the way, add some missing parts.

DEFINITION 2.3.2. [based on DA89] (TERM)

Let $F=\{f_1,\ldots,f_m\}$ and $V=\{V_1,\ldots,V_i,\ldots\}$ be respectively a set of functional symbols of some non-negative arity (finite number of them) and a set of variables. Term t is either

(a) a functional symbol $f_s$ from F of arity 0 (called *constant*), or

(b) a *variable* $V_t$ from V, or

(c) $f_p(t_1,\ldots,t_s)$, where $f_p$ is a functional symbol of arity s from the set F and the arguments $t_i$ are terms. Such $f_p(t_1,\ldots,t_s)$ is usually referred to as a *compound term*∎

The definition 2.3.2. is recursive and as long as F has at least one element of positive arity and V at least one variable inside, the number of terms that can be created is infinite and recursively enumerable (e.g. f(f(f(...f(X)...)))).

DEFINITION 2.3.3. [based on DA89] (LOGIC GRAMMAR SYMBOL)

Let $P=\{p_1,\ldots,p_n,[]\}$ be a set of some symbols called predicate symbols, all of some non-negative arity and one of them being "[]", the so called list symbol. Let $M_V=\{M_1,\ldots,M_r\}$ be a set of elements called meta-variables. Logic Grammar Symbol (LG symbol) L is either

(a) a *terminal* symbol t, iff $t=p_i$ and $p_i$ is a predicate symbol of arity 0, or

(b) a *meta-variable* symbol $M_i$ iff $M_i \in M_V$, or

(c) $p_k(t_1,\ldots,t_v)$, where $p_k$ is a predicate symbol of arity v from the set P and $t_1,\ldots,t_v$ are terms, or logic grammar symbols. This kind of logic grammar symbols are called *atoms* or *proper predicates*∎

To make the difference between the usage of terms and logic grammar symbols clear, let us present a production rule from a logic grammar that we analyze in depth in Chapter 10. We talk at this point only about the notions of terms and logic grammar symbols.

The production rule in question is: ***s(Form,Sem) --> Subj, vp(Form,[Subj],Sem).*** Here, *Form* and *Sem* are variables, *Subj* is a meta-variable and *s, vp* and *[]* are predicate symbols used to form the corresponding logic grammar symbols: *s(Form,Sem), vp(Form,[Subj],Sem)* and *[Subj]*, respectively. The arguments of *s(Form,Sem)* are the terms (in the form of variables) *Form* and *Sem*, respectively. *Subj*, in its first appearance in the production rule, immediately after the symbol "-->", is a logic grammar symbol, in the form of a meta-variable. The arguments of *vp(Form,[Subj],Sem)* are terms *(variables) Form* and *Sem* and another logic grammar symbol *[Subj]*. *[Subj]* consists of its predicate symbol *[]* and its arguments: a logic grammar symbol (meta-variable) *Subj*. Here, predicate *[]*, for a non-empty list, has its usual representation as having two arguments, *head* and *tail*. Tail is either another logic grammar symbol having *[]* as its predicate symbol, or it is the terminal symbol *[]*, that stands for an empty word (or empty list). Thus, here we have an effect known as *overloading* of the symbol *[]* with *[]* (non-empty list symbol which is a LG symbol with two arguments) and *[]* (terminal LG symbol for an empty list).

Let us emphasize that variables can be assigned values of constants (functional symbols of arity 0, that are available for building terms), or any other non-constant terms (variables and compound terms). Meta-variables can be assigned values of terminal logic grammar symbols, or non-terminal logic grammar symbols (meta-variables and proper predicates). Thus, no variable can be assigned the same value as a meta-variable and vice versa.

Borrowing from Prolog, the usual notation for logic grammar symbols, as seen in the previous example, uses words starting with a lower case letter for constants, functional symbols and predicate symbols and words starting with a capital letter for variables and meta-variables. In logic grammars in general, non-terminal symbols are distinguished from terminal symbols by the use of the square brackets for terminals, because they are both represented by words that start with a lower case letter. LG symbols are sometimes also referred to as trees because, by their above definition, they could be represented as trees.

DEFINITION 2.3.4. [based on DA89] (SUBSTITUTION)

Let V be a variable (meta-variable) and let $\alpha$ be a term (logic grammar symbol) different from V. Assigning $\alpha$ to V is called a substitution of variable (meta-variable) V with the term (logic grammar symbol) $\alpha$ and it is written as $V=\alpha$∎

DEFINITION 2.3.5. [based on DA89] (TERM UNIFICATION)

Let $t_1$ and $t_2$ be two terms. Terms unification is a procedure in which two terms $t_1$ and $t_2$ become equal by assigning certain values to some of the variables in them, according to the following rules:

(a) If $t_1=V_1$ and $t_2=V_2$, where $V_1$ and $V_2$ are <u>two variables</u>, then we say that $t_1$ and $t_2$ unify into V, if $V_1$ and $V_2$ both become V, where V is also a variable. We say then that $V_1$ and $V_2$ *share a value.*

(b) If $t_1=V$, where V is a <u>variable</u> and $t_2=f(t^1,...,t^n)$, where $f(t^1,...,t^n)$ is a <u>compound term</u>, such that none of the $t^1,...,t^n$ is, or contains variable V, <u>or</u> $t_2=f$, where f is <u>constant</u> (or a functional symbol of arity 0), then $t_1$ and $t_2$ are said to unify by the substitution $V=f(t^1,...,t^n)$ (or $V=f$) into $f(t^1,...,t^n)$ (or f). If variable V is present in $f(t^1,...,t^n)$, then prior to the substitution, all occurrences of V in $f(t^1,...,t^n)$ are to be renamed into a new variable V', that is not present in $f(t^1,...,t^n)$.

(c) If $t_1=f_1(t^1,...,t^n)$ and $t_2=f_2(u^1,...,u^p)$, where $f_1(t^1,...,t^n)$ and $f_2(u^1,...,u^p)$ stand for <u>compound terms or constants</u>, then:

(c') if $f_1=f_2$, $n=p$ and $t^i$ (if they are present) can unify with $u^i$, by some set of substitutions $S_i$ $(1\leq i\leq n)$ and substitutions from different $S_i$'s do not contradict each other (are consistent), we say that $t_1$ unify with $t_2$ by a set of substitutions $S=S_1\cup...\cup S_n$ and (c") if the previous is not the case, $t_1$ and $t_2$ are ununifiable (cannot unify).∎

For instance, the terms f(X,g(a,X)) and f(b,g(Y,Z)) unify into f(b,g(a,b)), with {X=b, Y=a, Z=b} (Z's value is b because it unifies to X, which in turn unifies with the value b). Another example would be when f(X,g(a,X)) and f(h(b,W),g(Y,Z)) unify into f(h(b,W),g(a,h(b,W))), with the substitution {X=h(b,W), Y=a, Z(=X)=h(b,W)}.

When the terms to be unified share any variable names, all occurrences of these must be renamed in one of the terms before attempting unification. For instance, for f(X,g(a,X)) and f(b,g(X,Z)), we should first obtain two terms with no variable names in common. We can rename X in the second term into a new variable name Y, so the second term becomes f(b,g(Y,Z)). Then the unification process yields f(b,g(a,b)), with the substitution {X=b, Y=a, Z=b}.

DEFINITION 2.3.6. [based on [DA89]] (LOGIC GRAMMAR SYMBOLS UNIFICATION)

Let $L_1$ and $L_2$ be two logic grammar symbols. Logic grammar symbols unification is a procedure in which two LG symbols $L_1$ and $L_2$ become equal by assigning certain values to some of the variables and meta-variables in them, according to the following rules:

(a) If $L_1=V_1$ and $L_2=V_2$, where $V_1$ and $V_2$ are some meta-variables, then we say that $L_1$ and $L_2$ unify into V, if both $V_1$ and $V_2$ become V, where V is also a meta-variable. We say then that $V_1$ and $V_2$ *share a value*.

(b) If $L_1=V$ and either $L_2=p(t^1,...,t^n)$ or $L_2=q$, where V is a meta-variable, $t^1,...,t^n$ are terms or logic grammar symbols, $p(t^1,...,t^n)$ is a proper predicate such that none of the $t^1,...,t^n$ is, or contains meta-variable V, and q is a terminal, then $L_1$ and $L_2$ are said to unify by the substitution $V=p(t^1,...,t^n)$ (or V=q) into $f(t^1,...,t^n)$ (or q).

(c) If $L_1$ is either $p_1(t^1,...,t^n)$ (a proper predicate) or $p_1$ (a logic grammar terminal, such that the arity n of its predicate symbol $p_1$ is 0) and $L_2$ is either $p_2(u^1,...,u^p)$ (a proper predicate)

<u>or</u> $p_2$ (a logic grammar <u>terminal</u>, such that the arity p of its predicate symbol $p_2$ is 0), where $t^1,\ldots,t^n$ and $u^1,\ldots,u^p$ are terms or logic grammar symbols, then:

(c') if $p_1=p_2$, $n=p$ and $t^i$ can unify as term with $u^i$, if they are both terms, or $t^i$ can unify as logic grammar symbol with $u^i$, if they are both logic grammar symbols, by some set of substitutions $S_i$ ($1 \le i \le n$), we say that $L_1$ unify with $L_2$ by a set of consistent (uncontradictory) substitutions $S=S_1 \cup \ldots \cup S_n$ and

(c") if the previous is not the case, $L_1$ and $L_2$ are ununifiable (cannot unify).■

To rephrase the definition in simple words, if $L_1$ and $L_2$ are two logic grammar symbols, possibly containing variables, terminals and meta-variables, unification finds values for those variables and meta-variables, (if such values exist), that make the two logical symbols identical. The values are actually either constants, or variables, or some compound terms for variables and meta-variables or proper predicates for LG symbols. The set of such assignments to the variables ({variable = value}, {variable_1 = variable_2}, or {variable = compound_term}) and meta-variables ({meta-variable=terminal}, {meta-variable_1=meta-variable_2} or {meta-variable=proper_predicate}) that makes two LG symbols equal is called a *substitution.*

Let $u(c_1, c_2, t) = c_3$ mean that $c_1$ and $c_2$ unify into $c_3$ by the substitution t.

DEFINITION 2.3.7. [based on [DA89]] (PRODUCTION RULE)

A production rule is an ordered pair $(\alpha_1,\alpha_2)$, where $\alpha_1$ and $\alpha_2$ are words made up of some logic grammar symbols.■

Production rules are usually written in the form $\alpha_1 \dashrightarrow \alpha_2$, which suggests that this is a kind of transformation.

DEFINITION 2.3.8. [based on [DA89]] (A PRODUCTION RULE APPLICATION IN A LOGIC GRAMMAR)

We say that the word $\beta$ is obtained from the word $\alpha$, by the application of the production rule $\gamma_1$--> $\gamma_2$, if the word $\alpha = \delta_1\gamma_1'\delta_2$ and the word $\beta = \delta_1''\gamma_2''\delta_2''$ and $\gamma_1'$ can be unified with $\gamma_1$ into $\gamma_1''$, with a substitution t and $\gamma_2$ becomes $\gamma_2''$, $\delta_1$ becomes $\delta_1''$ and $\delta_2$ becomes $\delta_2''$, all after being affected by the substitution t

The following figure gives visual explanation of the previous definition.

$$\alpha \text{ ---------------------------> } \beta$$

$$\delta_1\gamma_1'\delta_2 \text{---------------> } \delta_1''\gamma_2''\delta_2''$$

$$u(\gamma_1,\gamma_1',t) = \gamma_1''$$

$$u(\gamma_2,\gamma_2'',t) = \gamma_2''$$

$$u(\delta_1,\delta_1'',t) = \delta_1''$$

$$u(\delta_2,\delta_2'',t) = \delta_2''$$

$$\delta_1''\gamma_1''\delta_2'' \text{ ---------------> } \delta_1''\gamma_2''\delta_2''$$

**Figure 2.3.1.: An Application of a Production Rule in a Logic Grammar**

We use the notation $\alpha$ $\text{--}_G\text{-->}$ $\beta$, if $\beta$ can be derived from $\alpha$, by the application of a production rule from the grammar G.

It is obvious that if the logical symbols are restricted only to the unstructured LG symbols (terminals and meta-variables), the unification becomes a simple replacement, such as with ordinary phrase structure grammars.

The following simple example of an application of a production rule of a logic grammar should clarify the previous definition: Let $\alpha$ = h(X)f(a,b,X)d, $\beta$ = h(c)g(a,b)d, production rule p = f(X,Y,c) --> g(X,Y). Then, $\alpha$ $-_G\!\!>$ $\beta$, because u(f(X,Y,c),f(a,b,X),t) = f(a,b,c) with t = { X (from p) = a, Y = b, X (from h i $\alpha$, renamed first as Z) = Z = c }. Therefore, the whole $\alpha$ unifies with h(c)f(a,b,c)d, when the substitution t is applied to $\alpha$. When the substitution t is applied to the both sides of p, p becomes f(a,b,c) --> g(a,b). That implies also h(c)f(a,b,c)d $-_G\!\!>$ h(c)g(a,b)d, which completes the proof that $\alpha$ $-_G\!\!>$ $\beta$.

DEFINITION 2.3.9. [based on [DA89]] (DERIVATION IN A LOGIC GRAMMAR)

We say that word $\beta$ is obtained (derived) from word $\alpha$ in grammar G, iff either:

(a) word $\beta$ is obtained from the word $\alpha$, by the application of a production rule $\gamma_1$--> $\gamma_2$ from the set of the production rules of the grammar G, or

(b) word $\gamma$ is obtained from the word $\alpha$, by the application of a production rule $\gamma_1$--> $\gamma_2$ from the set of the production rules of the grammar G and word $\beta$ is obtained (derived) from the word $\gamma$ in grammar G∎

The previous recursive definition works in the following fashion: word $\beta$ is obtained (derived) from the word $\alpha$ in grammar G, if we can apply consecutively some p production rules $\gamma_{11}$ --> $\gamma_{21}$,..., $\gamma_{1p}$ --> $\gamma_{2p}$, in the given order. Also, there must exist words $\chi_1,\ldots,\chi_{p+1}$, such that $\chi_1 = \alpha$ and $\chi_{p+1} = \beta$ and $\chi_2$ is obtained from $\chi_1$, by the application of the production rule $\gamma_{11}$ --> $\gamma_{21}$ (in the sense of the definition 2.3.8.) and $\chi_3$ is obtained from $\chi_2$, by the application of the production rule $\gamma_{12}$ --> $\gamma_{22}$ and so on and $\chi_{p+1}$ is obtained from $\chi_p$, by the application of the production rule $\gamma_{1p}$ --> $\gamma_{2p}$. This process is also referred to as a definition of the *derivation of* $\beta$ *from* $\alpha$ *in grammar G by consecutively applying p production rules* $\gamma_{11}$ --> $\gamma_{21}$,..., $\gamma_{1p}$ --> $\gamma_{2p}$, *in the given order* and it is written as $\alpha$ $-\!\!-^{*}_{G}\!\!-\!\!>$ $\beta$. The derivation is said to be an *oracle derivation* if there is no variable that gets instantiated during the course of it (If there were uninstantiated variables initially, they remained uninstantiated). This kind of derivation assumes that all choices for variables appearing in the logic grammar symbols were made prior to it. We will need this term later on to associate it with the notion of *oracle derivation tree*. The derivation is said to be *full* (or *complete*), if there are no more uninstantiated variables in $\beta$ following the derivation and *partial* (or *incomplete*) otherwise. Also, if we allow $\beta$ to be an infinite word (consisting of an infinite number of non-terminals) and also allow application of infinitely many production rules instead of finite number of them, we then include the notion of *infinite derivations*, too. Otherwise, the derivation is called *finite*.

DEFINITION 2.3.10. [based on [DA89]] (LOGIC GRAMMAR)

$G = (N, T, S, P, F, V, P_s, M_v)$ is a logic grammar, iff

(a) N and T are sets of non-terminal and terminal LG symbols, respectively,

(b) S is a distinguished member of N (called starting symbol),

(c) P is a set of production rules, (consisting only of ordered pairs of words made up of the symbols from $N \cup T$),

(d) F is a finite set of functional symbols (used for creating terms),

(e) V is a set of variables (also, used for creating terms),

(f) $P_s$ is set of predicate symbols (used for the members for N and T) and

(g) $M_v$ is a set of meta-variables (also, used for N and T and during the derivation)$_\blacksquare$

DEFINITION 2.3.11. [based on [DA89]] (LANGUAGE DEFINED BY A LOGIC GRAMMAR)

Logic grammar G defines (describes) language L, iff $L = \{ w \mid w \in T^* \text{ \& } S \text{-}\text{-}^{*}_{G}\text{-}\text{-}{>} w \text{ is a finite derivation } \}_\blacksquare$

Set $\{ w \mid w \in T^* \text{ \& } S \text{-}\text{-}^{*}_{G}\text{-}\text{-}{>} w \text{ is a finite derivation } \}$ is usually denoted as *L(G)*.

DEFINITION 2.3.12. [based on [DA89]] (DEFINITE CLAUSE GRAMMARS)

Definite clause grammars (programs) are logic grammars such that its production rules are of the restrictive form $\alpha \text{ --> } \beta$, for which $\alpha \in N$ (left side is always a non-terminal, as with context-free phrase-structure grammars)$_\blacksquare$

The following two definitions are needed for explaining some intermediate notions, on our way to the definition for derivation tree.

DEFINITION 2.3.13. (TWO-LEVEL TREE (TLT))

Let $\alpha^1 \text{--}_G\text{-->} \beta^2$, in a definite clause program G, by the use of the production rule $\alpha\text{-->}\beta$ and a substitution that unifies $\alpha$ and $\alpha^1$ into $\alpha^2$ and $\beta$ into $\beta^2$. Let $\beta^2=S_1...S_m$, where $S_1,..,S_m$ are non-terminals and terminals from G. Two-level tree (TLT) corresponding to $\alpha^1\text{--}_G\text{-->}\beta^2$ is the tree $(\{\mathrm{pr\_sym}(\alpha^1),\mathrm{pr\_sym}(S_1),....,\mathrm{pr\_sym}(S_m)\},\{(\mathrm{pr\_sym}(\alpha^2),\mathrm{pr\_sym}(S_1)),...,(\mathrm{pr\_sym}(\alpha^1),\mathrm{pr\_sym}(S_m))\},\{\mathrm{pr\_sym}(\alpha^1)\},\{((\{\mathrm{pr\_sym}(S_1)\},\varnothing,\{\mathrm{pr\_sym}(S_1)\},\varnothing)...,(\{\mathrm{pr\_sym}(S_m)\},\varnothing,\{\mathrm{pr\_sym}(S_m)\},\varnothing))\}$, where $\mathrm{pr\_sym}(\alpha^1),\mathrm{pr\_sym}(S_1),...,\mathrm{pr\_sym}(S_m)$ denote main (outmost) predicate symbols of the logic grammar symbols $\alpha^2,S_1,...,S_m$, respectively

The following figure should be helpful in visualizing the above definition:

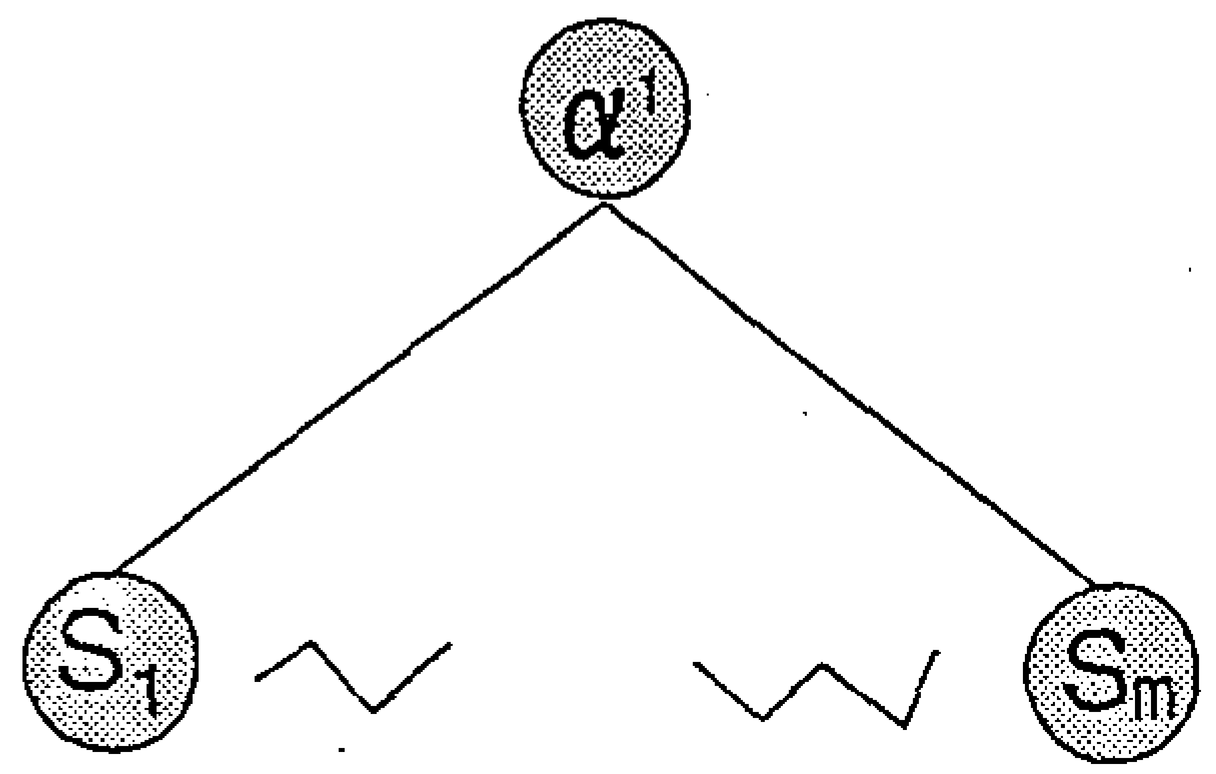


**Figure 2.3.2.: A TLT for the Derivation $\alpha^1 \text{--}_G\text{->} \beta^2$, by the Application of Production $\alpha\text{-->}\beta$.**

$\beta^2$ is obtained from $\alpha^1$, by the application of the production rule $\alpha \text{-->} \beta$. The root of the TLT is the node marked with the predicate symbol of $\alpha^1$ and the nodes at the level below the root are marked with the predicate symbols of $S_1,...,S_m$ and they appear in that exact order from left to right. There are also edges (one per a child) originating at the root to any of the nodes at the level below. We also say that the *TLT* *corresponds to* $\alpha \dashrightarrow \beta$ *production rule,* used *in a combination with the substitution* in question.

Let, from now on, UnI(L) and Inst(L) stand for the set of uninstantiated and instantiated or partially (outmost functional symbol or some internal variable is totally instantiated) instantiated variables in a logic grammar symbol L, respectively. Let $\mathrm{UnIn}(L)_{L'',L'}$ and $\mathrm{UnOut}(L)_{L'',L'}$ stand for sets of uninstantiated variables in L, before and after a production rule L" --> L' was used. In order L" --> L' to be applicable L" must be unifiable with L. If it is clear from the context what production rule is in question, we can omit the subscripts and just write UnIn(L), UnOut(L). If it is also obvious what logic grammar symbol is in question we can write only UnIn and UnOut.

DEFINITION 2.3.14. (TWO-LEVEL EXTENDED TREE)

Let $\alpha^1 \dashrightarrow_G \beta^2$, in a definite clause program G, using the production rule $\alpha \dashrightarrow \beta$ and substitution S that unifies $\alpha$ and $\alpha^1$ into $\alpha^2$ and $\beta$ $(=S_1...S_m)$ into $\beta^2$ $(=S_1^2...S_m^2)$. Here, $S_1,...,S_m$ and $S_1^2...S_m^2$ are logic non-terminals and terminals from G, unifiable by S ($S_1$ with $S_1^2,...,S_m$ with $S_m^2$). Let $\mathrm{UnI}(\alpha^1)$ and $\mathrm{UnI}(\alpha^2)$ be sets of all uninstantiated variables in logic nonterminals $\alpha^1$ and $\alpha^2$, respectively. Let $I(S_i)$ be set of variables from $\mathrm{UnI}(\alpha^1)$ that are instantiated in $S_i^2$ $(1 \leq i \leq m)$. Let mapping $P:\{1,...,m\} \dashrightarrow \{1,...,m\}$ be a permutation of the set $\{1,...,m\}$ and P' be the inverse mapping for P ($P(P'(i))=i$ and $P'(P(i))=i$). A two-level extended tree (TLET) corresponding to $\alpha^1 \dashrightarrow \beta^2$ derivation by the production rule $\alpha \dashrightarrow \beta$, is a TLT whose vertices are ordered triples (M,UnIn,UnOut), whose components satisfy the following points ((i), (ii) and (iii)):

(i) M=pr_sym($\alpha$), if M is the root of the TLT and M=pr_sym($S_i$), $(1 \leq i \leq m)$, for other nodes, or M is a meta-variable's name, if the corresponding node represents a meta-variable;

(ii) Let $\alpha_1^1$ be logic grammar symbol obtained by minimal unification of $\alpha$ and $\alpha^1$ (only instantiated variables in $\alpha$ or $\alpha^1$ are instantiated in $\alpha_1^1$, too). Then, UnIn[pr_sym($\alpha$)] is any set such that UnIn[pr_sym($\alpha$)]$\supseteq \mathrm{UnI}(\alpha_1^1)$ and UnOut[pr_sym($\alpha$)] =UnIn[pr_sym($\alpha$)]-($\mathrm{UnI}(\alpha^1)$-$\mathrm{UnI}(\alpha^2)$);

(iii) UnIn and UnOut for non-root grammar symbols $S_{P'(1)},\ldots,S_{P'(m)}$ ($S_1,\ldots,S_m$, taken in the order defined by the permutation P) are obtained by the following (quasi Pascal-like) non-deterministic procedure:

```
Begin
    Inst_So_Far:=Inst($\alpha_1{}^1$);
    UnIn[pr_sym($S_{P'(1)}$)]=UnIn[pr_sym($\alpha^1$)]$\cup$(UnI($S_{P'(1)}$)-
        Inst_So_Far);
    UnOut[pr_sym($S_{P'(1)}$)]:=Range_Set; {Range_Set is any set satisfying:
        UnIn[pr_sym($S_{P'(1)}$)]-I($S_{P'(1)}{}^2$)$\subseteq$Range_Set$\subseteq$UnIn[pr_sym($S_{P'(1)}$)]}
    Inst_So_Far := Inst_So_Far$\cup$(UnIn[pr_sym($S_{P'(1)}$)]-UnOut[pr_sym($S_{P'(1)}$)]);
    For j:=1 To m-2 Do Begin
        UnIn[pr_sym($S_{P'(j+1)}$)] :=
            UnOut[pr_sym($S_{P'(j)}$)]$\cup$(UnI($S_{P'(j+1)}$)-Inst_So_Far);
        UnOut[pr_sym($S_{P'(j+1)}$)] := Range_Set; {Range_Set is any set satisfying:
            UnIn[pr_sym($S_{P'(j+1)}$)]-I($S_{P'(j+1)}{}^2$)$\subseteq$Range_Set$\subseteq$UnIn[pr_sym($S_{P'(j+1)}$)]}
        Inst_So_Far := Inst_So_Far$\cup$(UnIn[pr_sym($S_{P'(j+1)}$)]-UnOut[pr_sym($S_{P'(j+1)}$)])
    End; {For}
    UnIn[pr_sym($S_{P'(m)}$)] :=
        UnOut[pr_sym($S_{P'(m-1)}$)]$\cup$(UnI($S_{P'(m-1)}$)-Inst_So_Far);
    UnOut[pr_sym($S_{P'(m)}$)]=UnIn[pr_sym($S_{P'(m)}$)]-I($S_{P'(m)}{}^2$);
    Inst_So_Far := Inst_So_Far$\cup$(UnIn[pr_sym($S_{P'(m)}$)]-UnOut[pr_sym($S_{P'(m)}$)])
End;■
```

We used UnIn[V] and UnOut[V] to denote UnIn and UnOut components of the vertex (V,UnIn,UnOut), respectively.

Thus, M is grammar symbol marking the vertex. UnIn component of the root is a set of uninstantiated variables from the logic symbol marking this node, before the rule represented by the tree was used and UnOut component of the root is a set of uninstantiated variables from the logic symbol marking this node, after the rule represented by this tree was used. For any other node of the tree (leaf pr_sym($S_i$)), UnIn and UnOut are two sets obtained by the forementioned computation.

The root will therefore be marked by pr_sym($\alpha$) and carrying the forementioned additional pieces of information: a list of uninstantiated variables from the logic symbol $\alpha^1$ marking this node before the rule represented by the TLET was used (before the unification took place and possibly some other variables from the root logic symbol became instantiated from the application of this rule) and a list of uninstantiated variables from the logic symbol marking this node after the rule represented by the TLET was used.

Let us illustrate the above definition by an example. Let the production rule be:

sen ( finite, SenSem ) -->

subj ( Num, SSem ), pred ( Num, SenSem, [ SSem, OSem ] ),

obj ( Num1, OSem ).

Thus, following the notation from the definition:

$\alpha$= sen(finite, SenSem),

$\beta$= subj(Num, SSem), pred(Num, SenSem, [SSem, OSem]), obj(Num1, OSem) and

$S_1$= subj(Num, SSem), $S_2$= pred(Num, SenSem, [SSem, OSem]) and $S_3$ = obj(Num1, OSem).

Let the actual derivation using the above rule be:

sen ( Form, SenSem ) $-\!-_G\!-\!->$

subj ( sing, john ), pred ( sing, make ( john, cakes ), [ john, cakes ] ), obj ( pl, cakes ).

Following the notation from the definition:

$\alpha^1$ = sen(Form, SenSem), $\alpha_1{}^1$ = sen(finite, SenSem), $\alpha^2$ = sen(finite, make(john, cakes)) and

$\beta^2$ = subj(sing, john), pred(sing, make(john, cakes), [john, cakes]), obj(pl, cakes), where,

$S_1^2$ = subj(sing, john), $S_2^2$ = pred(sing, make(john, cakes), [john, cakes]) and $S_3^2$ = obj(pl, cakes).

A TLET for this example can be formed by the following computation:

P(1)=2, P(2)=1, P(3)=3 (P'(1)=2, P'(2)=1, P'(3)=3) and thus

pr_sym($S_{P'(1)}$)=pr_sym($S_2$)=pred, pr_sym($S_{P'(2)}$)=pr_sym($S_1$)=subj, pr_sym($S_{P'(3)}$)=pr_sym($S_3$)=obj;

Inst_So_Far={Form} (indeed, =Inst($\alpha_1^1$));

UnIn[sen]={SenSem} (indeed, $\supseteq$UnI($\alpha_1^1$)={SenSem}),

UnOut[sen]=$\varnothing$ (indeed, =UnIn[sen]-(UnI($\alpha^1$)-UnI($\alpha^2$)=

{SenSem}-({Form,SenSem}-$\varnothing$));

UnIn[pred]={SenSem,Num,SSem,OSem} (indeed,

= {SenSem}$\cup$({Num,SenSem,SSem,OSem}-{Form})

= UnIn[sen]$\cup$(UnI(pred(Num,SenSem,[SSem,OSem]))-Inst_So_Far)

= UnIn[pr_sym($\alpha^1$)]$\cup$(UnI($S_{P'(1)}$)-Inst_So_Far),

UnOut[pred]={SSem,OSem} (indeed, satisfying:

{SenSem,Num,SSem,OSem}-{SenSem,Num,SSem,OSem}

$\subseteq${SSem,OSem}$\subseteq${SenSem,Num,SSem,OSem}, or equivalently

UnIn[pred]-I(pred(sing,make(john,cakes),[john,cakes]))$\subseteq${SSem,OSem}$\subseteq$UnIn(pred), or

UnIn[pr_sym($S_{P'(1)}$)]-I($S_{P'(1)}^2$)$\subseteq$Range_Set$\subseteq$UnIn[pr_sym($S_{P'(1)}$)]];

Inst_So_Far={Form,SenSem,Num} (indeed,

= {Form}$\cup$({SenSem,Num,SSem,OSem}-{SSem,OSem})

= Inst_So_Far$\cup$(UnIn[pred]-UnOut[pred])

= Inst_So_Far$\cup$(UnIn[$S_{P'(1)}$]-UnOut[$S_{P'(1)}$]);

UnIn[subj]={SSem,OSem} (indeed,

= {SSem,OSem}$\cup$({Num,SSem}-{Form,SenSem,Num})

= UnOut[pred]$\cup$(UnI(subj(Num,SSem))-Inst_So_Far

= UnOut[pr_sym($S_{P'(j)}$)]$\cup$(UnI($S_{P'(j+1)}$)-Inst_So_Far)),

UnOut[subj]={OSem} (indeed, satisfying:

{SSem,OSem}-{SSem}$\subseteq${OSem}$\subseteq${SSem,OSem}, or equivalently

UnIn[subj]-I(subj(sing,john))$\subseteq${OSem}$\subseteq$UnIn(subj), or

UnIn[pr_sym($S_{P'(2)}$)]-I($S_{P'(2)}{}^2$)$\subseteq$Range_Set$\subseteq$UnIn[pr_sym($S_{P'(2)}$)]};

Inst_So_Far={Form,SenSem,Num,SSem} (indeed,

= {Form,SenSem,Num}$\cup$({SSem,OSem}-{OSem})

= Inst_So_Far$\cup$(UnIn[subj]-UnOut[subj])

= Inst_So_Far$\cup$(UnIn[pr_sym($S_{P'(2)}$)]-UnOut[pr_sym($S_{P'(2)}$)]));

UnIn[obj]={OSem,Num1} (indeed,

= {OSem}$\cup$({Num1,OSem}-{Form,SenSem,Num,SSem})

= UnOut[subj]$\cup$(UnI(obj(Num1,OSem))-Inst_So_Far)

= UnOut[pr_sym($S_{P'(2)}$)]$\cup$(UnI($S_{P'(3)}$)-Inst_So_Far)),

UnOut[obj]=$\varnothing$ (indeed,

= {OSem,Num1}-{OSem,Num1}

= UnIn[obj]-I(obj(pl,cakes))

= UnIn[pr_sym($S_{P'(3)}$)]-I($S_{P'(3)}{}^2$);

Inst_So_Far={Form,SenSem,Num,SSem,OSem,Num1} (indeed,

= {Form,SenSem,Num}$\cup$({OSem,Num1}-$\varnothing$)

= Inst_So_Far$\cup$(UnIn[obj]-UnOut[obj])

= Inst_So_Far$\cup$(UnIn[pr_sym($S_{P'(3)}$)]-UnOut[pr_sym($S_{P'(3)}$]))).

It should be noticed here, that if we add to UnIn[sen] an outside uninstantiated variable, say X, in the beginning of the process, that variable would be present in UnIn's and UnOut's of all vertices. It will not, of course, get instantiated, because it does not participate in any way in the derivation process, except that is present and unchanged all the way. This property of TLET will be used when we define the notion of traversal, later in the study. Thus, it is obvious that for a derivation by a production rule there could be infinitely many TLET's, because we can add as many as necessary such "unparticipating" variables. Also, the presence of $\supseteq$ in the definition instead of =, enables another kind of flexibility, all in order to accommodate the requirements of the definition 2.3.16. of traversals. The notion of the oracle derivation trees (from Definition 2.3.15.) is also needed to get prepared for the definition of traversals.

The following figure illustrates the TLET introduced by the previous example:

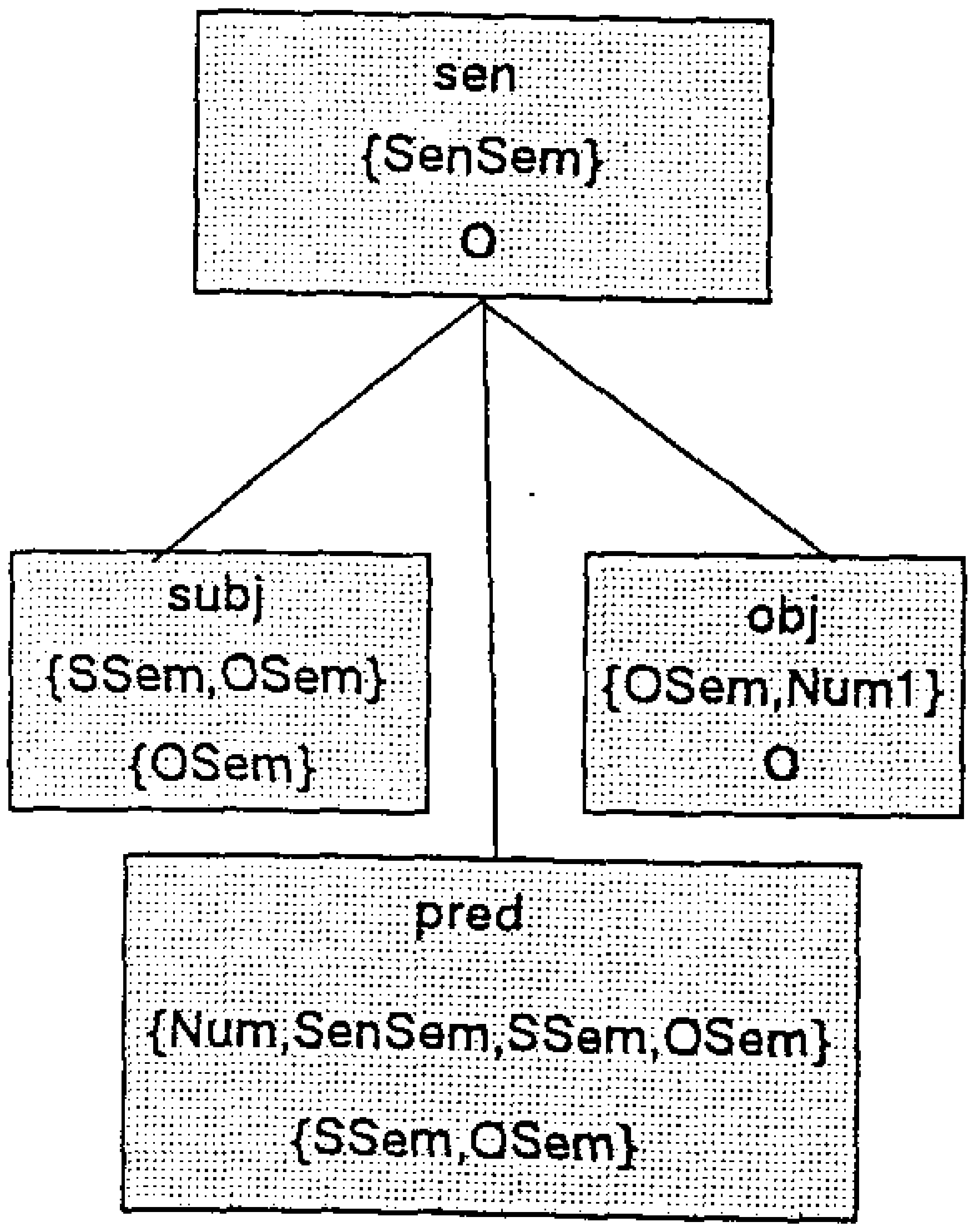


**Figure 2.3.3.: A Two-Level Extended Tree.**

DEFINITION 2.3.15. (ORACLE DERIVATION TREE)

An oracle derivation tree (ODT) for an oracle derivation $\alpha \dashrightarrow^{*}_{G} \beta$ in a logic grammar LG is a tree T such that:

(a) Its root is node marked by the predicate symbol of $\alpha$,

(b) If v is any non-leaf node of the tree and $\gamma_1,\ldots,\gamma_p$ all of its children nodes, then the structure consisting only of v, $\gamma_1,\ldots,\gamma_p$ and the edges from v to $\gamma_1,\ldots,\gamma_p$, is a TLT, corresponding to the appropriate production rule used in the combination with the corresponding substitution in the derivation $\alpha \dashrightarrow^{*}_{G} \beta$

It is clear that an infinite derivation tree will correspond to an infinite derivation and vice versa.

If no nodes in an oracle derivation tree contain uninstantiated variables, we call it a *full oracle derivation tree (FODT).* We adopted that name, because we can think of it as of a derivation tree corresponding to a derivation in the grammar where all values for variables were fully and correctly guessed before the derivation started and then used throughout the derivation. It should be noted here that even a FODT can be infinite, if for instance, the grammar contains a rule that duplicates the symbol on the left-hand side of the rule. A derivation corresponding to a FODT can be viewed as a proof that a string and a semantics correspond to each other in the given grammar and that both are correct language constructs, each in its corresponding class. It is much more often in practice of a NLP system that one of those two is known in the beginning and the derivation is simply a process of discovering the other. That establishes a need for the next definition.

DEFINITION 2.3.16. (TRAVERSAL (DERIVATION TREE))

Let $\alpha$ --$^{*}_{G}$--> $\beta$ be a derivation in a logic grammar G. Let also $V_{UnI}^{\alpha,\beta}$ be set of all variables that appear in $\alpha$ --$^{*}_{G}$--> $\beta$ initially uninstantiated and let $V_{UnO}^{\alpha,\beta}$ denote set of all variables left uninstantiated in $\beta$ after the $\alpha$ --$^{*}_{G}$--> $\beta$ derivation. Let S be set of all instantiations (substitutions) for variables from $V_{UnI}^{\alpha,\beta}$ made during the derivation. Let $\alpha'$ --$^{*}_{G}$--> $\beta$ be an oracle derivation corresponding to $\alpha$ --$^{*}_{G}$--> $\beta$ derivation, where $\alpha'$ is logic grammar symbol $\alpha$ after substitutions from S assigned values to corresponding variables from $\alpha$. Let T be an ODT corresponding to $\alpha'$ --$^{*}_{G}$--> $\beta$, $V=\{V_1,...,V_i,...\}$ set of all its vertices, $V_1$ its root and $I=\{1,...,i,...\}$ set of all indices corresponding to vertices from V. Let mapping P:I-->I be a permutation of elements from I and P' its inverse mapping (P(P'(j))=j and P'(P(k))=k). A traversal of T is a tree $T_t$ with the following properties:

Vertices are ordered quadruples (M,UnIn,UnOut,Ord), such that:

(i) M=pr_sym($V_j$), for some $V_j$ from V (*Marking Component of a Vertex*);

(ii) Ord=P(j), for an element $V_j$ from V and no two vertices have same component Ord (*Ordinal Component of a Vertex*);

(iii) UnIn, UnOut are sets of some variables (*Instantiation Status Components - UnIn and UnOut*) such that:

(*) For $V_{P'(1)}$ vertex, UnIn=$V_{UnI}^{\alpha,\beta}$, UnOut=$V_{UnO}^{\alpha,\beta}$,

(**) If P(i)=P(j)+1, for some vertices $V_i$ and $V_j$ from V, then:

(PCC) (a) if $V_i$ is parent node of $V_j$, then UnIn($V_i$)=UnOut($V_j$) and

(b) if $V_j$ is parent node of $V_i$ then UnIn($V_i$)=UnIn($V_j$) (*Parent Child Case*);

(NPCC) If $V_i$ and $V_j$ are not in parent-child relation, then UnIn($V_j$)=UnOut($V_i$);

(iv) If $V_k$ is a non-leaf vertex in the tree and $C_1,\ldots,C_m$ all of its children nodes, then ({($V_k$,UnIn,UnOut),($C_1$,UnIn,UnOut),...,($C_m$,UnIn,UnOut)}, {(($V_k$,UnIn,UnOut),($C_1$,UnIn,UnOut)),...,(($V_k$,UnIn,UnOut),($C_m$,UnIn,UnOut))},{($V_k$,UnIn,UnOut)},{(({($C_1$,UnIn,UnOut)},∅,{($C_1$,UnIn,UnOut)},∅),...,({($C_m$,UnIn,UnOut)},∅,{($C_m$,UnIn,UnOut)},∅))} is a TLET, corresponding to some production rule from the grammar G∎

The actual meaning of visiting a vertex is expanding the rule with the vertex's mark as the left-hand side of the rule. The instantiation status of the variables in the left-hand side logic grammar symbol of the rule is defined by the UnIn component of the vertex. Namely, all variables appearing in UnIn for the vertex that are also arguments of the logic grammar symbol, are considered uninstatiated before the rule is expanded. The UnOut component will contain only those variables from UnIn that did not get instantiated during the expansion of the production rule.

The following two figures illustrate (PCC) and (NPCC) from the above definition.

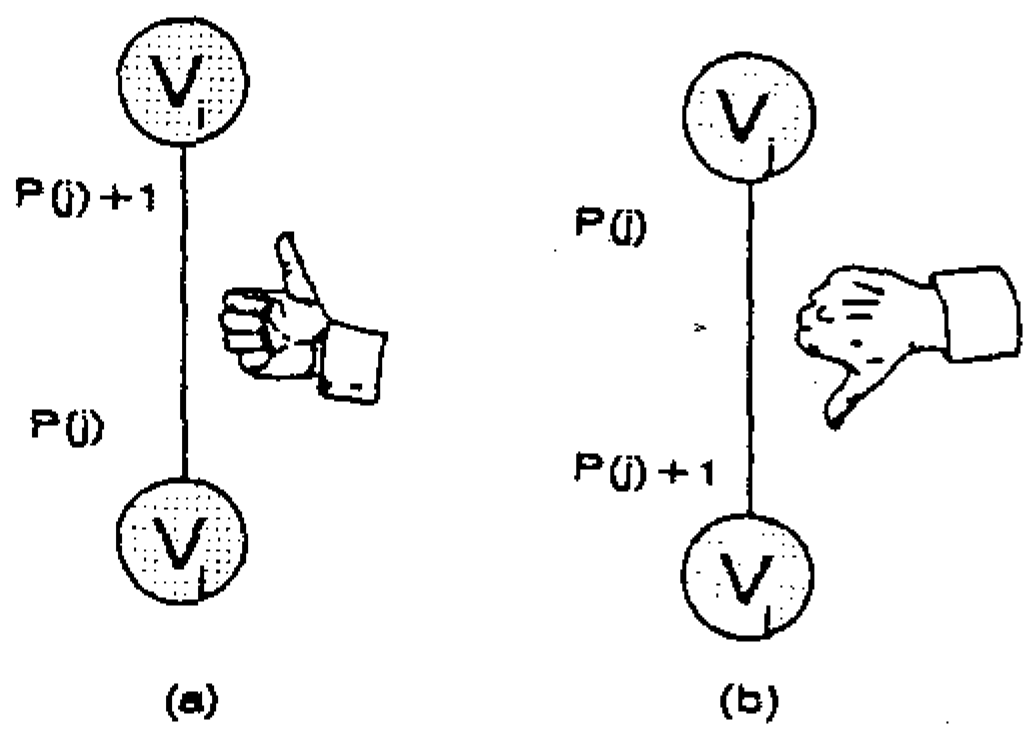


Figure 2.3.4.: Parent-Child Cases (a) and (b).

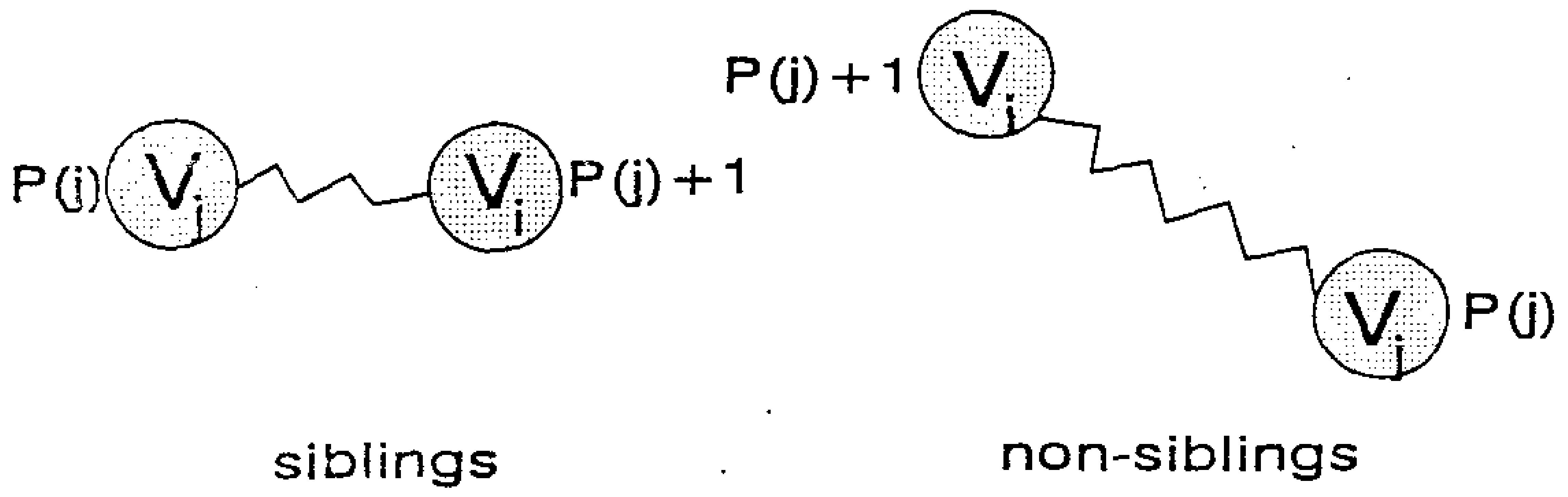


Figure 2.3.5.: Non-Parent-Child Case.

The following examples should be helpful in understanding the previous definition. Let the following be a toy grammar for *wh-questions:*

```
whques(WhSem) --> whsubj(Num,WhSubj),
            whpred(Num,Tense,[WhSubj,WhObj],WhSem),
            whobj(WhObj).                                        (1)
..........
whsubj(_,who) --> [who].                                         (2)
..........
whsubj(_,what) --> [what].                                       (3)
..........
whpred(sing,perf,[Subj,Obj],wrote(Subj,Obj)) --> [wrote].        (4)
..........
whobj(this) --> [this].                                          (5)
```

We first present an ODT corresponding to the derivation of the sentence *who wrote this* and then three different traversals of the tree. In the following graphs, rectangular regions represent vertices. Each vertex contains: the predicate symbol of the corresponding logic grammar symbol, the UnIn set, the UnOut set and its ordinal number indicating the visiting order.

The following figure represents the oracle derivation tree for the sentence *who wrote this.*

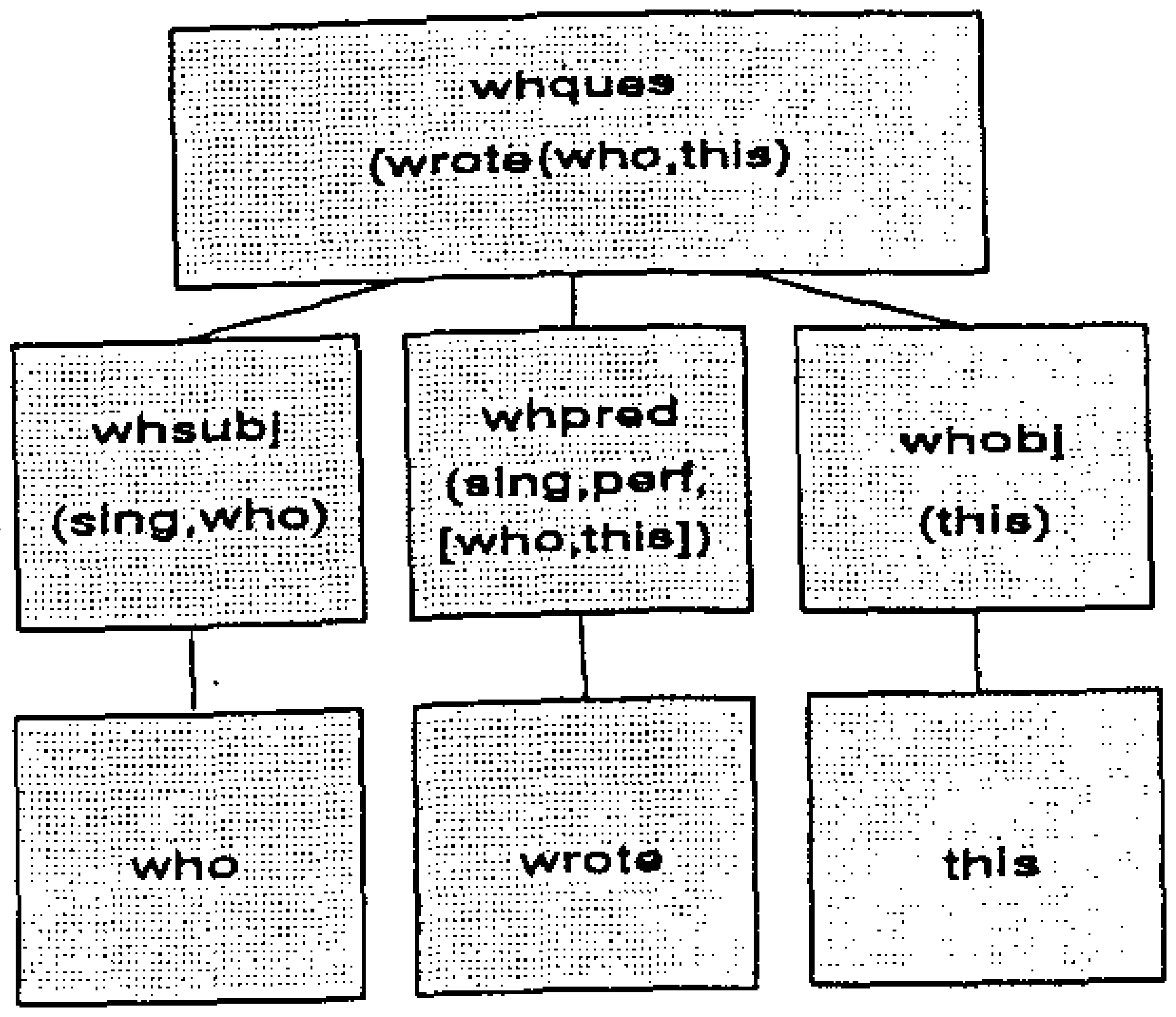


**Figure 2.3.6.: ODT for Sentence *who wrote this.***

Throughout this study, we will use term *derivation tree* for a derivation $\alpha \text{-}_G\text{->} \beta$ in a logic grammar G interchangably with the above defined notion of a traversal. The reason for that lies in the fact that the actual traversal could be viewed as a generalization of the notion of a derivation in a grammar. Such a generalized derivation allows skipping some parts of the classical derivation and doing them later.

Next we will present three different traversals of the ODT from Figure 2.3.6. in full details. As in the previous figure, rectangular regions represent vertices. They consist of the following components: predicate, UnIn and UnOut sets and ordinal number. Marking component is the corresponding predicate symbol, UnIn and UnOut are sets of uninstantiated variables when the vertex was entered and exited and ordinal number defines after how many other vertices was this vertex visited.

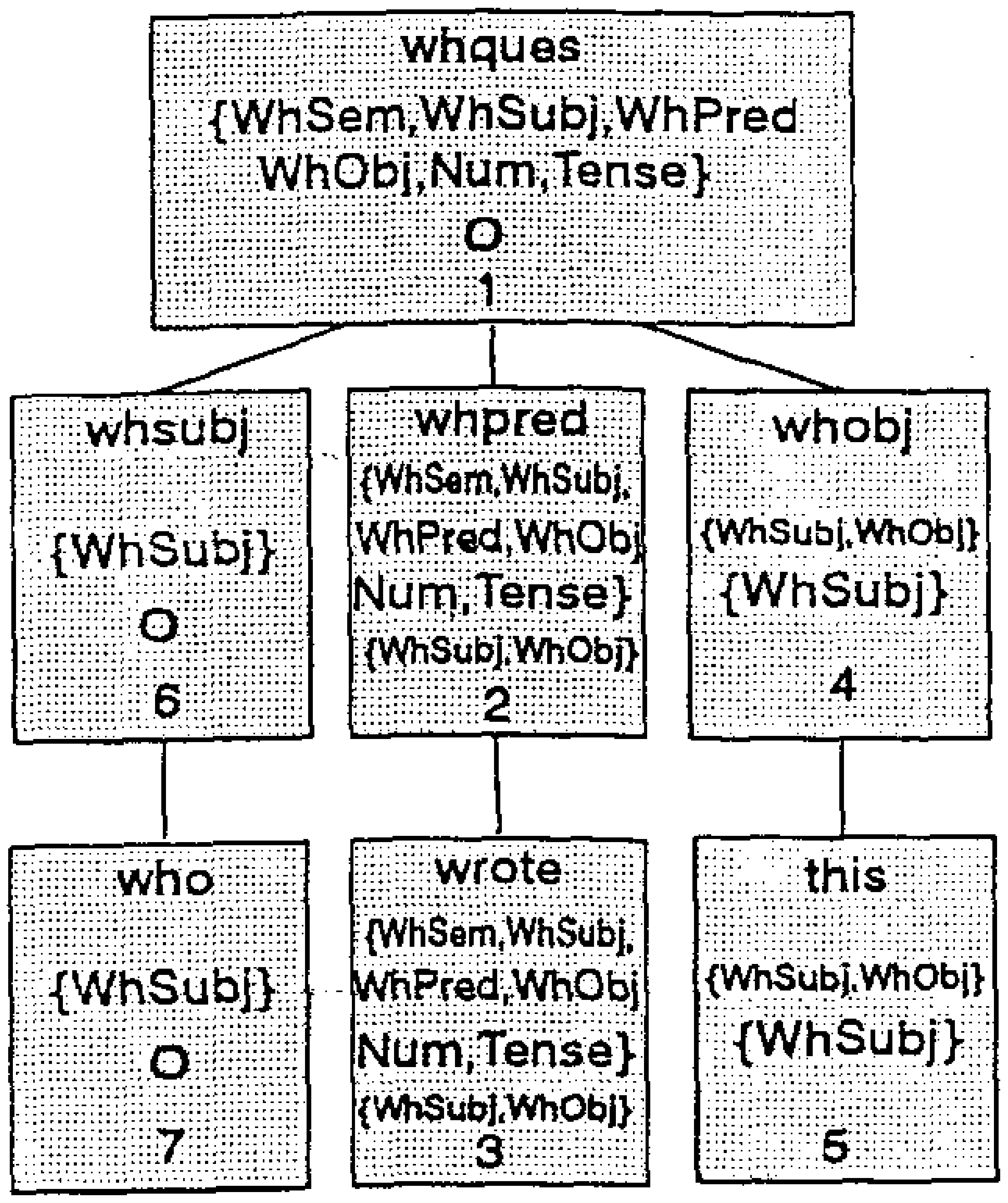


**Figure 2.3.7.: A Traversal of the ODT for sentence *who wrote this.***

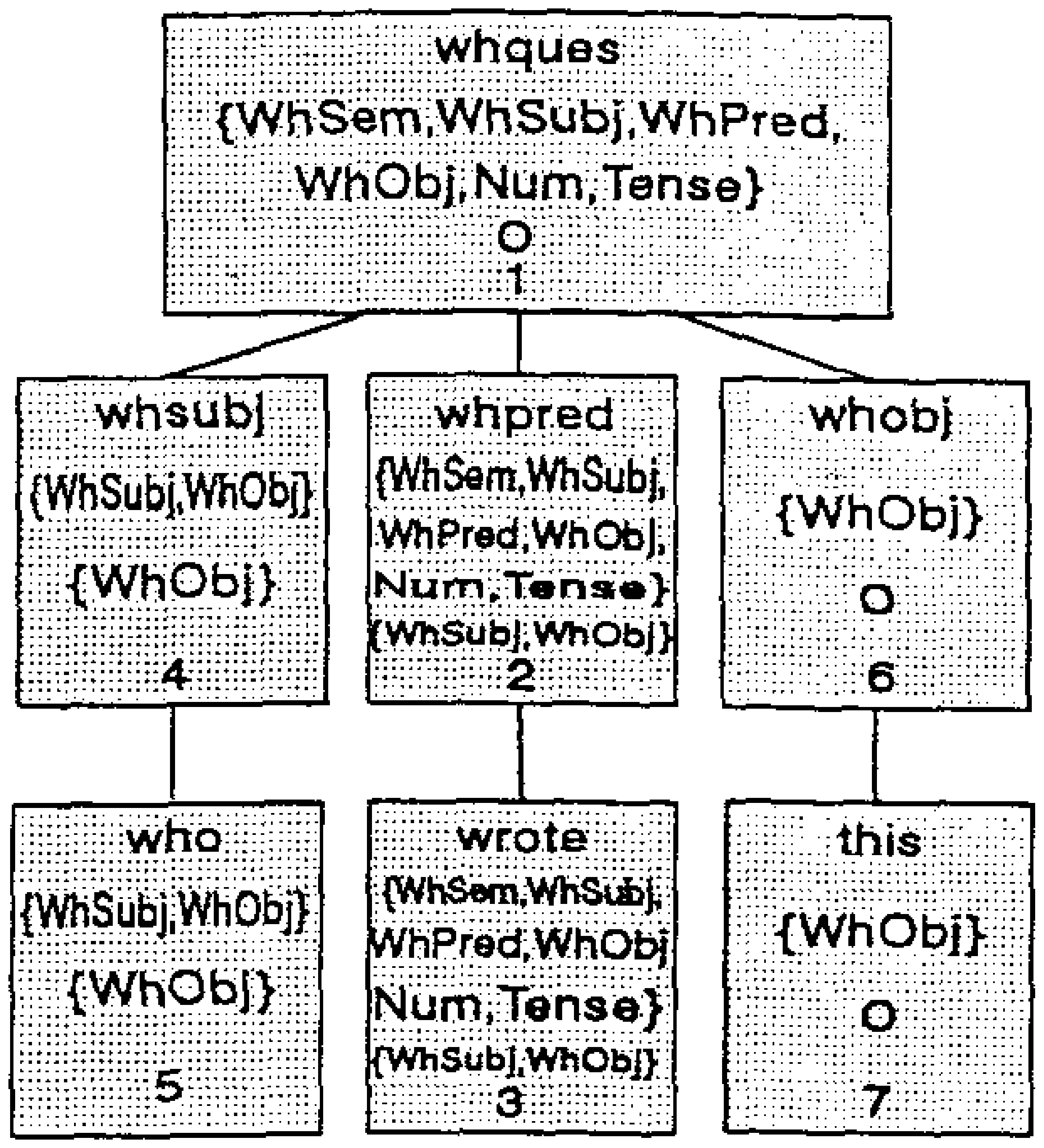


Figure 2.3.8.: A Traversal of the ODT for sentence *who wrote this.*

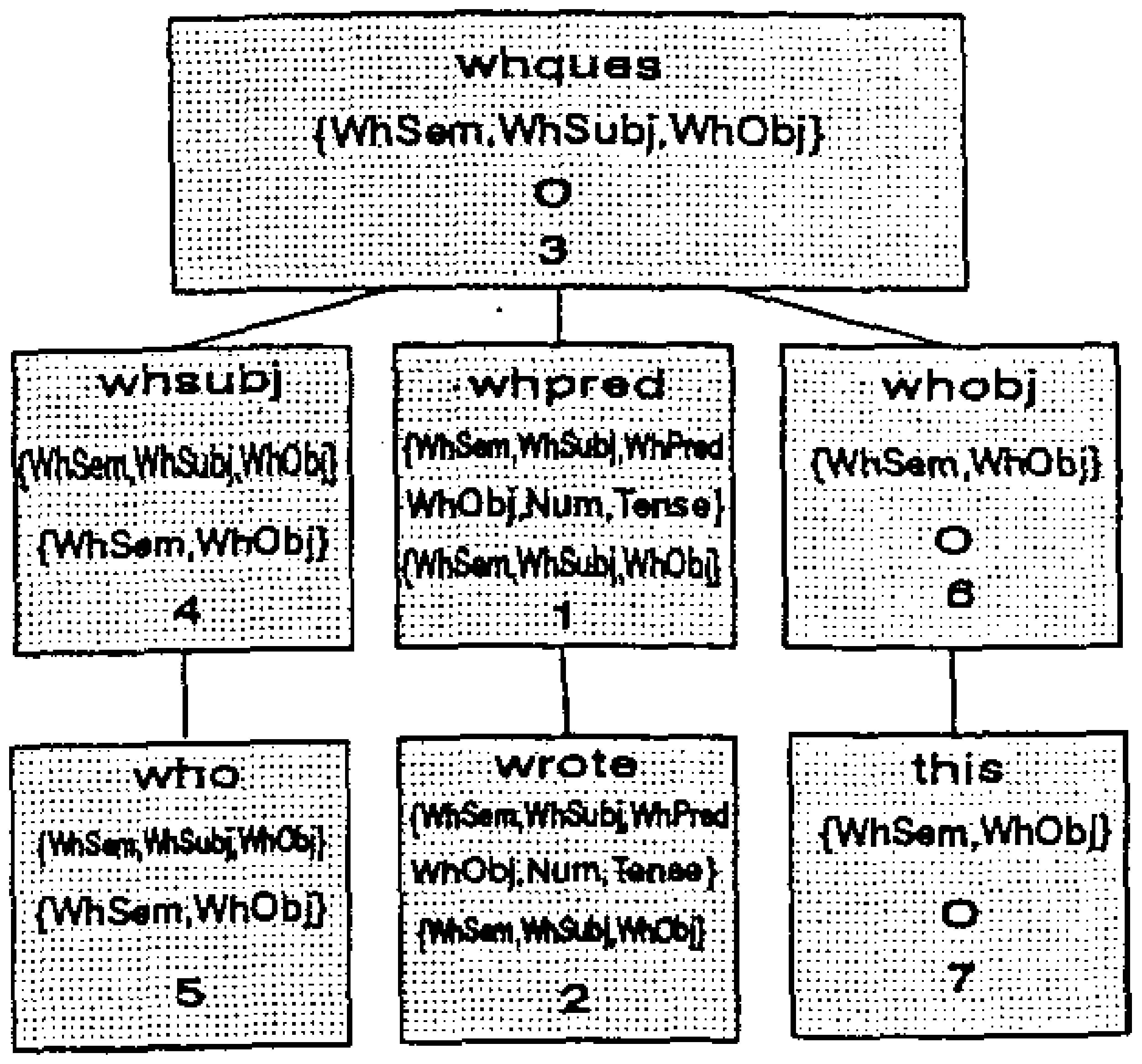


Figure 2.3.9.: Another Traversal of the ODT for sentence *who wrote this.*

Based on the definition 2.3.9. of the derivation in a logic grammar and the previous definition of the derivation tree, a derivation in a logic grammar will always have a corresponding *derivation tree.*

Intuitively, it is clear that a derivation tree T has the following properties: The root of the tree is marked by $\alpha'$ and contains set of variables uninstantiated in $\alpha$ and set of variables that remained uninstantiated after the derivation, plus an ordinal number equal to some index s, meaning that this was the s-th visited node in the tree. Vertex $V_{P'(1)}$ will have ordinal number 1, meaning that this was the first node in this tree that was visited. If v is any non-leaf vertex in the tree and $\gamma_1,\dots,\gamma_r$ all of its children, then the structure *Str* containing v, $\gamma_1,\dots,\gamma_r$ and edges from v to $\gamma_1,\dots,\gamma_r$, one edge per a child, is a TLET. Also, v is marked by the logic grammar symbol that it corresponds to. The node v also contains two sets of uninstantiated variables. First set consists of those variables that were uninstantiated before this rule was used in the derivation and the second one the uninstantiated variables after its expansion finished, if that ever happened. If the expansion never stops, instead of the second set, it contains the symbol $\infty$. The node also has a number assigned to it, actually an ordinal number indicating how many nodes in T were visited before this node. The leafs are similarly marked by the terminals they represent, plus two sets of uninstantiated variables, containing those variables that were uninstantiated before this rule was used in the derivation and after its expansion finished. They also have numbers indicating the order in which they were visited.

In the light of the definition 2.3.16., if the *Semantics* attribute of the root node *parser_generator(Semantics,String)* is instantiated, then the goal is to discover a finite derivation tree, with the root at *parser_generator(semantics,String)* whose leafs make up the list that becomes the value of *String* at the end of the derivation process. This corresponds to the generation process. On the other hand, if the *string* attribute representing a natural language sentence is given at the beginning, the goal is to discover a finite derivation tree, with its node root at *parser_generator(Semantics,string)*, such that some of its interior nodes will contribute pieces to the final value for the variable *Semantics*. How the pieces get composed into the final value for *Semantics* is defined by the production rules which are used in the derivation. This later derivation corresponds to the parsing process. Thus, both, parsing and generation can be viewed as processes of discovering their corresponding derivation trees.

DEFINITION 2.3.17. (INSTANTIATION OF (META-)VARIABLE BY DERIVATION TREE)

A derivation tree T instantiates a variable (meta-variable) V to a constant (terminal) v, if there is a node U in T such that V∈UnIn(U)-UnOut(U).∎

Thus, V appears among U's initially uninstantiated variables, but does not appear among uninstantiated variables, when the node is left.

DEFINITION 2.3.18. (CORRESPONDING STRING AND SEMANTICS IN A LOGIC GRAMMAR)

If given a value *string* of attribute String, there exists a derivation tree that instantiates variable *Semantics* to the value *semantics* and vice versa, if given a value *semantics* for the attribute *Semantics*, there is a derivation tree that instantiates variable *String* to the value *string*, we say that *string and semantics* correspond to each other in the logic grammar.∎

DEFINITION 2.3.19. (INVERTED STRING AND SEMANTICS VALUES)

If *str* and *sem* are two values representing a correct string and a correct semantics corresponding to each other in the logic grammar G and their derivation trees are two possibly different traversals of a same FODT, *str* and *sem* are said to be inverses of one another (or inverted string and semantics values).∎

In the sense of the above definitions, parsing and generation algorithms can be viewed as methods of discovering a finite derivation tree, or as methods for traversing a FODT (finite or infinite). Thus, they could be identified with how they define the order in which the nodes are visited and the instantiation status of variables before and after each node was visited.

DEFINITION 2.3.20. (GRAMMAR PROCESSING ALGORITHM AND TRAVERSALS PRODUCED BY IT)

Let A be a procedure that uniquely defines how derivation trees are constructed for a given logic grammar G. A is then called a grammar processing algorithm and the corresponding derivation trees are called traversals (derivation trees) produced by A∎

Algorithm A actually defines the order(s) in which nodes of each FODT (finite or infinite) in a logic grammar G are visited and the instantiation status of variables before and after each node was visited in accordance with the grammar rule used at the particular node. In other words, A prescribes traversals of FODT's or the way that derivations in G are performed.

The following definitions are merely introducing the terminology that we will use in this study and also formalizing the notions usually treated informally.

DEFINITION 2.3.21. (FULL PARSING (GENERATION) ALGORITHM)

Let A be a grammar G processing algorithm. If each *derivation tree T produced by A* with a starting logic grammar symbol *S* containing uninstantiated variable *Sem* (or *Str*) representing semantics (string) at its root instantiates variable *Sem* (*Str*) to some value *sem* (*str*), A is a full parsing algorithm, or FPA (full generation algorithm, or FGA)∎

If the previous is not true for each derivation tree, but for a significant number of them, A is then called just *parsing algorithm (generation algorithm)*, or *PA* (*GA*). If A is both PA and GA, it is then called *parser-generator*. The proportion of all derivation trees produced by A for which A is a PA is called *degree of parsibility (DP) of grammar G under the grammar processing algorithm A*. Similarly, the proportion of all derivation trees produced by A for which A is a GA is called *degree of generability (DP) of grammar G under the grammar processing algorithm A*. Finally, the proportion of all derivation trees produced by A for which A is both parsing and generation algorithm is called *degree of reversibility (DR) of grammar G under* *the grammar processing algorithm A.*

An ideal case would be to have a full parser-generator, but in practice that is usually beyond reach taking into consideration practical limitations. Thus, the problem in practice comes down to specifying a set of conditions on the underlaying grammar G (the less restrictive, the better), that will, if fulfilled, guarantee certain degree of reversibility of G under A.

With previous definitions at hand, we will be able to give a more precise and accurate description of major criteria concerning the logic grammars and the algorithms that process them: generality of logic grammars and completeness, soundness, reversibility, efficiency and finiteness of grammar processing algorithms. But first, we turn our attention to the relation between DCG formalism and formalisms of Turing machines and type 0 phrase structure grammars.

# 3. DEFINITE CLAUSE GRAMMARS (DCG's) AND SOME OTHER FORMALISMS

In this chapter the equivalence of the DCG formalism with the formalism of Turing machines and later on, type 0 grammars will be proven. First formalism to be introduced are Turing machines.

## 3.1. Equivalence of the Turing Machines and DCG's

We just quote theorem 3.1.1. (without giving its proof) as a direction of the equivalence between Turing machines and DCG's of lesser importance to us and then we proceed with the major result of this section, theorem 3.1.2..

**Theorem 3.1.1.** For any definite clause grammar G = ( N, T, S, **P**, **F**, **V** ) there exists a Turing machine $\mu$, such that $\mu$ accepts (recognizes) the language generated by G (L(G) = L($\mu$))∎

**Theorem 3.1.2.** For any Turing machine $\mu$ there exists a DC grammar G such that the grammar G generates the same language that is recognized by $\mu$ (L($\mu$) = L(G))∎

*Proof.*

Let the starting symbol for grammar G be $q_1$([],X) and productions for the grammar be the following ones:

0 $q_1$([],X) --> $q_f$([],X), X.

I For each of the transformations of type (i) (from the definition 2.1.3.) add a corresponding rule:

$q_i$([],[$a_j$|X]) --> $q_l$([],[□,$a_k$|X]).

II For each of the transformations of type (ii) add a corresponding rule:

$q_i([a_m|X],[a_j|Y])$ --> $q_l(X,[a_m,a_k|Y])$.

III For each of the transformations of type (iii) add a corresponding rule:

$q_i(X,[a_j|Y])$ --> $q_l(X,[a_l|Y])$.

IV For each of the transformations of type (iv) add a corresponding rule:

$q_i(X,[a_j])$ --> $q_l([a_k|X],[\square])$.

V For each of the transformations of type (v) add a corresponding rule:

$q_i(X,[a_j,a_m|Y])$ --> $q_l([a_k|X],[a_m|Y])$.

VI $q_0(X,Y)$ --> $\varepsilon$. ($\varepsilon$ is empty word).

VII [X|Y] --> X, Y.

VIII [] --> $\varepsilon$.

The first argument of each of the non-terminals (they actually correspond to the states of the Turing machine $\mu$) is *the left part* of the current instantaneous description in its reversed order (read from right to left). By *left part,* we mean the portion of the instantaneous description to the left from the current state. The instantaneous description describes a state where the machine is during the computation for the input word. The second argument represents *the right part* of the instantaneous description (read from left to right). The production 0 serves merely to start the computation and copy the input word (given in X). Production VI makes $q_0$ disappear after it was reached, leaving the input word as the only remaining content. The input word remained untouched during the derivation as the rightmost end of every intermediate word derived during this process. The input word is then said to be derived. The remaining production rules simulate the computation of $\mu$. The grammar G when run as a DCG in Prolog will actually simulate the work of the corresponding Turing machine. We represent *left part* of an instantaneous description in a right-to-left fashion, rather than in its natural left-to-right manner as it appears in the instantaneous description because the list mechanism provided in DCG's, offers a convenient and here very usable distinction between the first letter of a word and the rest of them. Since a move to the left by a Turing machine transfers the rightmost letter of *the left part* of the instantaneous description to *the right* *part*, the representation from right to left appears very appropriate in this situation.

The part $L(\mu)\subseteq L(G)$ of the proof is obvious from the previous explanation. The other direction ($L(\mu)\supseteq L(G)$) of the proof is also simple, because for each production used in a derivation of a word (except the first and the last one, that simply initialize and finalize a derivation), we can use its corresponding transformation and ultimately a corresponding Turing instruction. Also, for each instantaneous description transformation, there is an instruction that the transformation was assigned to. Thus, we can recall the whole sequence of instructions of $\mu$ used for the recognition of the word∎

The following example should be helpful for understanding a transition from a Turing machine to the corresponding DCG. The following Turing machine program recognizes the language $\{ a^{(n+1)}b \mid n\geq 0 \}$. So, $\mu=(\{a,b,\ \}, \{q_0,q_1,q_2,q_3,q_4,q_5\}, q_1, q_0, \mathbf{P})$ and the program **P** consists of the following instructions, to the right from which are the corresponding productions from the corresponding DC grammar:

| | |
|---|---|
| | $q_1([],X)\text{-->}q_1([],X),\ X$ |
| $q_1a \text{ --> } aRq_2$ | $q_1(Y,[a\mid X])\text{-->}q_2([a\mid Y],X)$ |
| $q_1b \text{ --> } bNq_1$ | $q_1(Y,[b\mid X])\text{-->}q_1([b\mid Y],X)$ |
| $q_1\Box\text{-->}\ \Box Nq_1$ | $q_1(Y,[\Box\mid X])\text{-->}q_1(Y,[\Box\mid X])$ |
| | $q_1(Y,[])\text{-->}q_1(Y,[])$ |
| $q_2a \text{ --> } aRq_2$ | $q_2(Y,[a\mid X])\text{-->}q_2([a\mid Y],X)$ |
| | $q_2(Y,[a])\text{-->}q_2([a\mid Y],[\Box])$ |
| $q_2\Box\text{-->}\ \Box Rq_3$ | $q_2(Y,[\Box\mid X])\text{-->}q_3([\Box\mid Y],X)$ |
| | $q_2(Y,[\Box])\text{-->}q_3([\Box\mid Y],[\Box])$ |
| $q_2b \text{ --> } bNq_2$ | $q_2(Y,[b\mid X])\text{-->}q_2(Y,[b\mid X])$ |
| $q_3b \text{ --> } bRq_4$ | $q_3(Y,[b\mid X])\text{-->}q_4([b\mid Y],X)$ |
| | $q_3(Y,[b])\text{-->}q_4([b\mid Y],[\Box])$ |

| | |
|---|---|
| $q_3$a --> aN$q_3$ | $q_3$(Y,[a\|X])-->$q_3$(Y,[a\|X]) |
| $q_3$□--> □N$q_3$ | $q_3$(Y,[□\|X])-->$q_3$(Y,[□\|X]) |
| $q_4$□--> □R$q_5$ | $q_4$(Y,[□\|X])-->$q_5$([□\|Y],X) |
| | $q_4$(Y,[□])-->$q_5$([□\|Y],[□]) |
| $q_4$a --> aN$q_4$ | $q_4$(Y,[a\|X])-->$q_4$(Y,[a\|X]) |
| $q_4$b --> bN$q_4$ | $q_4$(Y,[b\|X])-->$q_4$(Y,[b\|X]) |
| $q_5$□--> □N$q_0$ | $q_5$(Y,[□\|X])-->$q_0$(Y,[□\|X]) |
| $q_5$a --> aN$q_5$ | $q_5$(Y,[a\|X])-->$q_5$(Y,[a\|X]) |
| $q_5$b --> bN$q_5$ | $q_5$(Y,[b\|X])-->$q_5$(Y,[b\|X]) |
| | $q_0$(Y,X)-->ε |
| | [X\|Y]-->X, Y |
| | []-->ε. |

## 3.2. Equivalence of Type 0 Grammars and DCG's

As with Turing machines, we first quote the theorem 3.2.1., as a direction of the equivalence between DCG's and type 0 phrase-structure grammars of lesser significance to us at the moment and then we proceed with our major result in this section, theorem 3.2.2..

**Theorem 3.2.1.:** Let $G_L$ be a DCG and L($G_L$) language generated by $G_L$. There exists a type 0 phrase-structure grammar G that accepts the same language (L($G_L$)=L(G))∎

**Theorem 3.2.2.:** Let G = ( **N**, **T**, **P**, S ) is a type-0 grammar in Chomsky's hierarchy, where **P** = { $\alpha$-->$\beta$ | $\alpha, \beta \in$ ( **N** $\cup$ **T** )$^*$ }. Then, there exists a logic grammar $G_L$ such that L(G) = L($G_L$)∎

*Proof.*

We start building grammar $G_L$ by specifying its components.

Non-terminals are all and only the symbols whose main (outermost) functors are: **sym**, **merge-right** and list functor denoted with "[]".

Terminals are the same as the terminals for the grammar G (set **T**).

Starting symbol is **sym([],[S],[])**.

Set of production rules consists of the following productions:

(i) sym([],[S],[])-->sym([],$\alpha^{\wedge}$,[]), if S-->$\alpha$ was in **P**. Here, $\alpha^{\wedge}$ stands for a lists of all the symbols in $\alpha$.

(ii) sym( X, $\alpha^{\wedge}$, Y )-->sym( X, $\beta^{\wedge}$, Y ), if $\alpha$-->$\beta$ was from **P**. Here, again, $\alpha^{\wedge}$ and $\beta^{\wedge}$ stand for lists of all the symbols from $\alpha$ and $\beta$ respectively.

(iii) sym( X, Y, [Z|U] )-->merge_right( V, Y, Z ), sym( X, V, U ).

(iv) sym( [X|Y], Z, U )-->sym( Y, [X|Z], U ).

(v) sym( X, Y, Z )-->merge_right( Y, H, T ), sym( X, H, [T|Z] ).

(vi) sym( X, [Y|Z], U )-->sym( [Y|X], Z, U ).

(vii) sym( [], X, [] )-->X.

(viii) merge_right( [X], [], X )-->ε. Here, ε stands for *an empty word* (a word with no letters).

(ix) merge_right( [X|U], [X|Y], Z )-->merge_right( U, Y, Z ).

(x) merge_right( X, Y, Z )-->ε.

(xi) []-->ε.

(xii) [X|Y]-->X, Y.

The meaning of the parameters in **sym** is the following: Lists represented by the first, second and the third parameter when concatenated (the first parameter, when taken from right to left, then the second parameter, read from left to right and finally, the third parameter read from the left of the list to its right) represent always the symbols from the current word in the derivation from the grammar G.This derivation is to be *simulated* in grammar $G_L$. The first parameter contains the list of symbols *to the left* of the symbol currently being expanded by a production from **P**. The symbol being expanded is represented by the second parameter. The list for the first parameter is given in right-to-left order with respect to how its elements appear in its corresponding instantaneous description. We refer to the first, second and third parameter as *to the left, middle* and *to the right* part of the current word during the derivation. We decided on this representation rather than on some other because during a derivation in a grammar G, one has to be able to extract an arbitrary part of the current word that matches a left side of a production rule and then replace it by the right side of the same production rule. Of course, the *unextracted* pieces of the current word, to the left and right from the part just replaced remain the same and they should be just concatenated to the new piece, to its left and its right, in the usual manner. In order to achieve that kind of simulation in $G_L$, one must assure that, letter by letter of *to the left* and *to the right* part could be concatenated to the *middle* part, at any time. When talking about *to the left* part, letter by letter must be obtainable from its right to its left, as opposed to *the right* part, where this is done from its left to its right. Productions (iii) and (iv) take care of that.

Also, there must be a possibility for taking letters both from right and from left of *the middle part* of the current word and merging them to *the right part* and *to the left part* of the current word, respectively. This must be possible at any time of a derivation, for arbitrary number of letters from right and from left. Productions (v) and (vi) serve that purpose.

The meaning of the symbol **merge_right( L, B, Last )** is that its first parameter is always a list **L**, whose last element is the third parameter **Last** and the second parameter **B** is a list of symbols in their exact order as they appear in **L**, that precede the last element in **L**. After determining these parts for a list of symbols (by applying productions (viii) and (ix)), the symbol **merge_right** disappears (by applying the production (x)). After a string of terminals is reached, by applying the productions (iii), (iv), (v), (vi), (viii), (ix), (x), **sym( [], w, [] )** can be easily derived (w is the final list of terminals). Then the production (vii) gets rid of the symbol **sym** and productions (xi) and (xii) transform the list of terminals w into the corresponding string of the same terminals in the same order∎

An example may be helpful in explaining how a derivation from G is simulated in $G_L$. Language { $a^{(n)}b^{(n)}$ | n≥2 } can be obviously generated by the following grammar:

G = ( N, T, **P**, S ), where N={ S, A }, T={ a, b } and

**P**={ S-->aAb, A-->aAb, A-->ab }.

So, for the word aaabbb, a derivation would be:

**S**-->**aAb**==a**A**b-->a**aAb**b==aa**A**bb-->aa**ab**bb==aaabbb.

In the derivation, bold case letters are used always for the part of the current word that, either has just replaced a part of the previous word, or for the part of the current word just to be replaced by the use of a production rule from the grammar. In addition, italics are used for variables from $G_L$, in order to distinguish them from non-terminals from grammar G, for which also capital case letters were chosen. Operator == suggests how the same word is to be viewed, in order to apply the next production rule. For instance, from the starting symbol **S**, a first word to be derived is **aAb**. That word should then be viewed as a**A**b in order to emphasize that **A** is the part to be replaced by the right side of a production rule, whose left side is **A**. Whatever is on the left and right from **A** remains like it is.

The derivation in $G_L$ that simulates these steps in G, mentioned for aaabbb is as follows:

sym([a], [A], [b])-->sym([a], [a,A,b], [b]) (ii).

sym([a], [a,A,b], [b])-->sym([a,a], [A,b], [b]) (vi).

sym([a,a], [A,b], [b])-->merge_right([A,b], *H*, *T*), sym([a,a], *H*, [*T*|b]) (v).

Now, again, **merge_right([A,b],** ***H*****,** ***T*****)** unifies with the left side of the production rule (ix) and *H* is substituted by [A|*HT*].

**merge_right([A,b], [A|*HT*], *T*)**, sym([a,a], [A|*HT*], [*T*|b]) -->

**merge_right([b], *HT*, *T*)**, sym([a,a], [A|*HT*], [*T*|b]) (ix).

Again, here, *HT* will unify with [] and *T* with b, in order to apply the production rule (viii):

**merge_right([b], [], b)**, sym([a,a], [A], [b,b])-->sym([a,a], [A], [b,b]) (viii).

What follows is to move the final terminal string to the second argument of **sym**, in order to prepare for production rules (vii), (xi) and (xii) that will leave the string as the final result of the derivation in $G_L$.

sym([a,a], [A], [b,b])-->sym([a,a], [a,b], [b,b]) (ii).

sym([a,a], [a,b], [b,b])-->sym([a], [a,a,b], [b,b]) (iv).

sym([a], [a,a,b], [b,b])-->sym([], [a,a,a,b], [b,b]) (iv).

sym([], [a,a,a,b], [b,b])-->merge_right(*V*, [a,a,a,b], b), sym([], V, [b]) (iii).

Now, **merge_right(*V*, [a,a,a,b], b)** unifies with the left side of the production rule (ix) and is substituted by [a|*VT*]. Now:

**merge_right([a|*VT*], [a,a,a,b], b)**, sym([], [a|*VT*], [b])-->

  **merge_right(*VT*, [a,a,b], b)**, sym([], [a|*VT*], [b]) (ix).

In the same fashion, *VT* is substituted by [a|*VTT*] in:

**merge_right([a|*VTT*], [a,a,b], b)**, sym([], [a,a|*VTT*], [b])-->

  **merge_right(*VTT*, [a,b], b)**, sym([], [a,a|*VTT*], [b]) (ix).

Next, *VTT* is substituted by [a|*VTTT*] and then *VTTT* by [b|*VTTTT*]:

**merge_right([a|*VTTT*], [a,b], b)**, sym([], [a,a,a|*VTTT*], [b])-->

  **merge_right(*VTTT*, [b], b)**, sym([], [a,a,a|*VTTT*], [b]) (ix).

**merge_right([b|*VTTTT*], [b], b)**, sym([], [a,a,a,b|*VTTTT*], [b])-->

  **merge_right(*VTTTT*, [], b)**, sym([], [a,a,a,b|*VTTTT*], [b]) (ix).

Finally, **merge_right(*VTTTT*, [], b)** will unify with the left side of the production rule (viii) and *VTTTT* will be substituted by [b]:

**merge_right([b], [], b)**, sym([], [a,a,a,bl[b]], [b])

   -->ε, sym([], [a,a,a,b,b], [b]) (viii).

The last **b** will be moved to the second argument in the same fashion:

sym([], [a,a,a,b,b], [b])-->

   merge_right(, [a,a,a,b,b], b), sym([], V, []) (iii).

Now, **merge_right([], [a,a,a,b,b], b)** unifies with the left side of the production rule (ix) and is substituted by [al*VT*]. Now:

**merge_right([al*VT*], [a,a,a,b,b], b)**, sym([], [alVT], [])-->

   **merge_right(VT, [a,a,b,b], b)**, sym([], [alVT], []) (ix).

In the same fashion, *VT* is substituted by [al*VTT*] in:

**merge_right([al*VTT*], [a,a,b,b], b)**, sym([], [a,al*VTT*], [])-->

   **merge_right(*VTT*, [a,b,b], b)**, sym([], [a,al*VTT*], []) (ix).

Next, *VTT* is substituted by [al*VTTT*] and then *VTTT* by [bl*VTTTT*] and *VTTTT* by [bl*VTTTTT*]:

**merge_right([al*VTTT*], [a,b,b], b)**, sym([], [a,a,al*VTTT*], [])-->

   **merge_right(*VTTT*, [b,b], b)**, sym([], [a,a,al*VTTT*], []) (ix).

**merge_right([b|*VTTTT*], [b,b], b)**, sym([], [a,a,a,b|*VTTTT*],[])-->

**merge_right(*VTTTT*, [b], b)**, sym([], [a,a,a,b|*VTTTT*], []) (ix).

**merge_right([b|*VTTTTT*], [b], b)**, sym([], [a,a,a,b,b|*VTTTTT*], [])-->

**merge_right(*VTTTTT*, [], b)**, sym([], [a,a,a,b,b|*VTTTTT*], []) (ix).

Finally, **merge_right(*VTTTTT*, [], b)** will unify with the left side of the production rule (viii) and *VTTTTT* will be substituted by [b]:

**merge_right([b], [], b)**, sym([], [a,a,a,b,b|[b]], [])-->

ε, sym([], [a,a,a,b,b,b], []) (viii).

Next, **sym** will disappear:

sym([], [a,a,a,b,b,b], [])-->[a,a,a,b,b,b] (vii).

And finally, the list of terminals will turn into a sequence of the same terminals:

[a,a,a,b,b,b]-->a, [a,a,b,b,b] (xii).

a, **[a,a,b,b,b]**-->a, **a, [a,b,b,b]** (xii).

a, a, **[a,b,b,b]**-->a, a, **a, [b,b,b]** (xii).

a, a, a, **[b,b,b]**-->a, a, a, **b, [b,b]** (xii).

a, a, a, b, **[b,b]**-->a, a, a, b, **b, [b]** (xii).

a, a, a, b, b, **[b]**-->a, a, a, b, b, **b, []** (xii).

a, a, a, b, b, b, **[]**--> a, a, a, b, b, b (xi).

The previous example points out how $L(G) \subseteq L(G_L)$ can be proved.

The opposite direction ($L(G) \supseteq L(G_L)$) is the immediate consequence of the following facts: Each derivation of a string of terminals must end in exactly the same fashion as the derivation of aaabbb and that is, to end with a **sym ([], [a,a,a,b,b,b], [])**-like form with the list of letters from the string as the middle argument of **sym**. In order a string **w** to be derivable in $G_L$, there must be a way to end up with **sym ([], w^, [])** (**w^** is the list of letters from the string **w**). Since the derivation in $G_L$ up to such a form can be nothing more than the simulation of what is being done in a derivation of the same string in G, then the fact that **w** is derivable in $G_L$ also means that it is derivable in G.

# 4. CRITERIA FOR EVALUATION OF LOGIC GRAMMARS AND GRAMMAR PROCESSING ALGORITHMS

## 4.1 Generality of Logic Grammars

Intuitively, generality of a logic grammar is judged based on how large is the subset of a natural language described by the grammar.

DEFINITION 4.1.1. (GENERALITY COMPARISON OF LOGIC GRAMMARS)
Let L' and L" be subsets of the same natural language L described by grammars G' and G" respectively. If $L' \subset L''$ then it is said that the grammar G' is *less general* than the grammar G"∎

The hierarchy of grammars can be established with respect to this criterion, relying on the hierarchy of cardinal numbers for sets.

## 4.2. Completeness of Grammar Processing Algorithms

Intuitively, completeness of a grammar processing algorithm is judged based on how large is the subset L' of the language L defined by the grammar, for which the algorithm produces finite derivation trees for elements from L'.

DEFINITION 4.2.1. (COMPLETENESS COMPARISON OF GRAMMAR PROCESSING ALGORITHMS)
Let A1 and A2 be two grammar processing algorithms of the same kind (both parsing, both generation, or both parser-generators). Let L' and L" be subsets of the same language L defined by a grammar G, for whose elements there are finite derivation trees produced by A1 and A2 respectively. If $L' \subset L''$, then it is said

that the algorithm A2 is *more complete* than the algorithm A1∎

The last definition allows us to establish a hierarchy of algorithms with respect to the completeness criterion based on the cardinal numbers for the language subsets corresponding to different grammar processing algorithms. In fact, we can actually import the whole theory of cardinal numbers for sets that already exist.

## 4.3. Soundness of Grammar Processing Algorithms

Soundness is a notion that can be defined similarly for generation and for parsing algorithms. For parsing algorithms, it intuitively has the following meaning: if from an incorrect string the algorithm produced a meaningful and correct semantic structure, this is considered a sign of unsoundness of the parsing algorithm. Similarly for generation algorithms, if from an incorrect semantic structure, the algorithm produces a correct and meaningful string, this is considered a sign of unsoundness of this generation algorithm. The following schemes (Figure 4.3.1.) graphically describe these situations.

DEFINITION 4.3.1. (SOUNDNESS COMPARISON OF GRAMMAR PROCESSING ALGORITHMS)

Let A1 and A2 be two parsing (generation) algorithms. Let L1 and L2 be respectively sets of incorrect strings (semantic structures), parsed (generated) by A1 and A2 respectively, into some correct and meaningful semantic structures (strings) in a language L. If card(L1-L2) > card(L2-L1), then we say that A1 is *less sound* algorithm than A2∎

In practice, the soundness problem is usually known also as *overproducing* and is much more frequent in the context of parsing algorithms. It means that an algorithm accepts some irregular structures as they were regular ones, without reporting on their incorrectness. Thus, although the underlying grammar correctly specified that the construct in question was an incorrect one and should be rejected, the algorithm that process (implements) the grammar did not reject it.

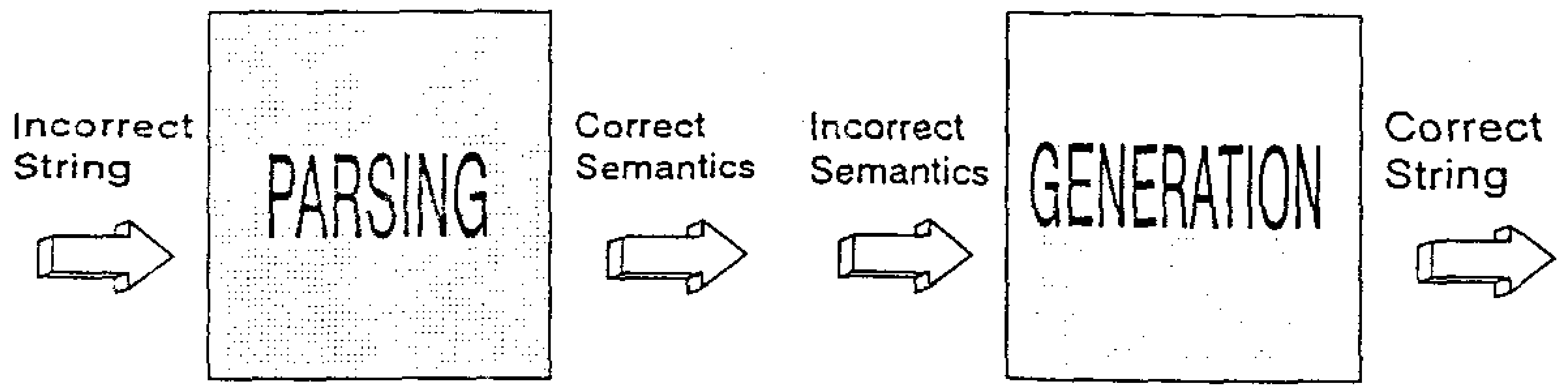


**Figure 4.3.1.: Unsoundness of Parsing and Generation.**

### 4.4. Finiteness of Grammar Processing Algorithms

It is a known fact that grammar processing algorithms can produce finite or infinite derivation trees for a given root. The criterion of finiteness is often treated in two different manners. One is when the amount of finiteness is actually amount of finite derivation trees produced by the algorithm. This actually coincides with the notion of completeness. The other, that we adopt here, is given in a more complex form. It deals with a set of conditions imposed on the underlying grammar, sufficient to guarantee only finite derivation trees. The more restrictive the conditions are, the lower the place of the algorithm in the hierarchy. These conditions are most often given in the form of sufficient, though not necessary conditions, which can only be viewed as a kind of "worst case" and not as an absolute estimate. The most notable attempts in treating this criterion for grammar processing algorithms are presented by M. Dymetman, et al. in [D90a], [D90b], [DI88] and [DIP90]. The previous discussion can be summarized by the following definition.

DEFINITION 4.4.1. (FINITENESS COMPARISON OF GRAMMAR PROCESSING ALGORITHMS)

Let A1 and A2 be two grammar processing algorithms of the same kind (either both parsing or both generation algorithms). Let S1 and S2 be two minimal sets of necessary and sufficient conditions, imposed on an underlaying grammar G, that guarantee only finite derivation trees for G, produced by A1 and A2 respectively. Let also GS1 be set of all grammars satisfying conditions from S1 and GS2 set of all grammars satisfying conditions from S2. If GS1$\supset$GS2, we say that the algorithm A1 is *more finite* than A2∎

## 4.5. Efficiency of Grammar Processing Algorithms

Efficiency criterion can also be interpreted in two different ways. One is given by the notion of optimal tree for a certain construct (string or semantics) and the other one through the notion of optimal discovery. Here, we adopt the second one, because it better captures the notion and quality of the actual derivation performed by a real algorithm and it is given relatively to a particular derivation and not in an "on average" fashion. The first one more compares the ideal case (oracle-provided) situations and has little practical significance. One reason for that is that it only determines if an optimal tree was found or not and not whether it was found in an optimal manner. Finding optimal tree can sometimes be done in a very inefficient manner, so that its discovery actually says nothing about the efficiency of the algorithm.

DEFINITION 4.5.1. (OPTIMAL TREE)

The optimal tree for a given construct (string or semantics) is a FODT corresponding to a derivation of that construct, which has the smallest number of nodes among all such FODT's∎

DEFINITION 4.5.2. (ON AVERAGE EFFICIENCY COMPARISON OF GRAMMAR PROCESSING ALGORITHMS)

Let A1 and A2 be two grammar processing algorithms of the same kind. Let L1 and L2 be sets of all optimal trees found and traversed in the corresponding manners by A1 and A2, respectively. If $L1 \supset L2$, then we say that A1 is *more efficient on average* than A2.▪

DEFINITION 4.5.3. (BEST DERIVATION NUMBER)

Let $T_1,..,T_i,..$ be all derivation trees corresponding to the derivation $\alpha -_G^* \!\!> \beta$ in a grammar G and produced by a grammar processing algorithm A in the given order ($T_1$ produced first,..., $T_i$ i-th, etc.). Let numbers of different production rules from grammar G, tried during these derivations of $T_1,...,T_i,..$ be $n_1,..,n_i,..$ respectively. The best derivation number relative to algorithm A for the derivation $\alpha -_G^* \!\!> \beta$ (abbreviated as *BDNRA*$(A,\alpha -_G^* \!\!> \beta)$) is $\min(n_1,..,n_i,..)$.▪

The last definition introduced a number that better describes the potential of an algorithm than its real performance. If BDNRA is not $n_1$, i.e. related to the first derivation tree produced by the algorithm, then it actually describes how the algorithm best performs, if it was rerun for an unlimited (and possibly infinite) number of times. Although BDNRA might point out if and in what direction the algorithm could be changed in order to improve its performance, it practically does not contribute much to the evaluation of the current form of the algorithm. The algorithm's present performance is best described by the results that first run of the algorithm returned, namely by number $n_1$. We formalize this discussion by the following definition.

DEFINITION 4.5.4. (DERIVATION NUMBER)

Let $T_1$ be first derivation tree corresponding to the derivation $\alpha -_G^* \!\!> \beta$ in a grammar G and produced by a grammar processing algorithm A. Let number of different production rules from grammar G, tried during this derivation be $n_1$. The number $n_1$ is then called derivation number relative to the algorithm A for the derivation $\alpha -_G^* \!\!> \beta$ (abbreviated as *DNRA*$(A,\alpha -_G^* \!\!> \beta)$).▪

DEFINITION 4.5.5. (EFFICIENCY COMPARISON RELATIVE TO A DERIVATION)

Let A1 and A2 be two grammar processing algorithms of the same kind. Let $\alpha \xrightarrow{*}_G \beta$ be a derivation in a grammar G and let D1 and D2 be DNRA's for $\alpha \xrightarrow{*}_G \beta$ relative for A1 and A2 respectively. If D1 < D2, we say that the algorithm A1 is *more efficient with respect to* $\alpha \xrightarrow{*}_G \beta$ than the algorithm A2∎

DEFINITION 4.5.6. (NUMBER OF CASES-BASED EFFICIENCY COMPARISON OF ALGORITHMS)

Let A1 and A2 be two grammar processing algorithms of the same kind (either both parsing or both generation algorithms). Let L1 and L2 be sets of constructs for which A1 and A2, respectively produce finite derivation trees. Let L be an intersection of L1 and L2. Let $L' \subseteq L$ be the set of all constructs for which A1 is more efficient than A2 and $L'' \subseteq L$ be the set of all constructs for which A2 is more efficient than A1. If card(L')>card(L"), we say that the algorithm A1 is *more efficient based on the number of cases* than the algorithm A2∎

Obviously, if A1 is more efficient than A2 with respect to any derivation (in the sense of the definition 4.5.5.), then A1 must be also more efficient than A2 in the sense of the definition 4.5.6..

Taking into consideration the nature of the crucial definition 4.5.3. ("...number of production rules tried during a derivation..."), efficiency criterion is actually measuring how deterministic an algorithm is. In fact, based on how many guesses are made during a derivation (no, few, a lot,...), a hierarchy can be established with respect to the efficiency criterion, too.

## 4.6. Reversibility of Grammar Processing Algorithms

Whether one grammar processing algorithm is more or less reversible than another can be judged based on the number of corresponding string and semantic structures that can be derived one from another using this algorithm.

DEFINITION 4.6.1. (REVERSIBILITY COMPARISON OF GRAMMAR PROCESSING ALGORITHMS)

Let A1 and A2 be two grammar processing algorithms and G a logic grammar defining a language L. Let L' be a set of all ordered pairs $(\sigma,\varsigma)$, where $\sigma$ is a string from L such that A1 can both parse $\sigma$ into semantic structure $\varsigma$ and generate $\sigma$ from $\varsigma$. Let L" be a set of all ordered pairs $(\sigma,\varsigma)$, where $\sigma$ is a string from L such that A2 can both parse $\sigma$ into semantic structure $\varsigma$ and generate $\sigma$ from the $\varsigma$. If $L' \subset L''$, it is said that A2 algorithm is *more reversible* than A1 algorithm∎

In the above definition we used set of ordered pairs of strings and corresponding semantic structures for L' and L", rather than just sets of *reversible* strings. The reason for that is that a string can be parsed into several different semantic structures and some of them may be reversible into the original string and some other may not. Thus, we really cannot say that such a string is reversible, neither we can say that it is not. To take this type of situation into account when judging reversibility and to measure the number of cases in which the reversal is possible, we opted for ordered pairs.

The following section will introduce a relation defined on set of all possible traversals for an ODT, that can be used to estimate the quality of a grammar processing algorithm based on the previous criteria.

# 5. STAS - A RELATION FOR ANALYSIS OF GRAMMAR PROCESSING ALGORITHMS

DEFINITION 5.1. (SAME-TO-A-SUBTREE (STAS) TRAVERSALS OF AN ODT)

Two traversals T' and T" of a ODT T corresponding to a derivation $\alpha \ -_G^{*}\!> \beta$ in a grammar G are said to be same-to-a-subtree (STAS), if the following condition holds: Let N be any node of the tree T and $S_1,\ldots,S_n$ all subtrees whose roots are immediate descendants of N. If the order of traversal for the subtrees of T' is $S_i^1,\ldots,S_i^n$ (without paying attention to the order in which the nodes within the subtrees will be visited) and the order of traversal for the subtrees of T" is $S_j^1,\ldots,S_j^n$, then $S_i^1=S_j^1,\ldots,S_i^n=S_j^n$ ($S_i^1,\ldots,S_i^n$ and $S_j^1,\ldots,S_j^n$ are some of the subtrees $S_1,\ldots,S_n$).■

STAS does not imply that the order in which the nodes are visited will necessarily be the same. In other words, two traversals can be STAS and their orderings of nodes quite different, as it will be shown on a couple of examples.

The following figure presents an example of two tree traversals which are STAS. The nodes are visited by the first algorithm in the order indicated by the arabic numbers (A, E, J, K, F, B, G, H, L, C, I, M, D). This traversal, as it will become clear later in the study, can be produced by semantic-head-first algorithm. The second algorithm, marked here by roman numbers, is depth-first search algorithm, performed in a top-down, left-right fashion, producing the following ordering: A, B, E, F, J, K, F, C, G, H, L, D, I, M. As it can be verified easily, the choice of the subtrees for the traversals at any node is same for both algorithms. For instance, at the node A, both algorithms first traverse the leftmost subtree, rooted at the node B, then the middle one (rooted at C) and at the end the rightmost (rooted at D). However, it should be observed that the order in which the nodes in the subtrees are visited by the two algorithms is not the same. Same type of verification can be done for the remaining nodes in the tree. Therefore, these two traversals are STAS, although the order in which the nodes are visited is different.

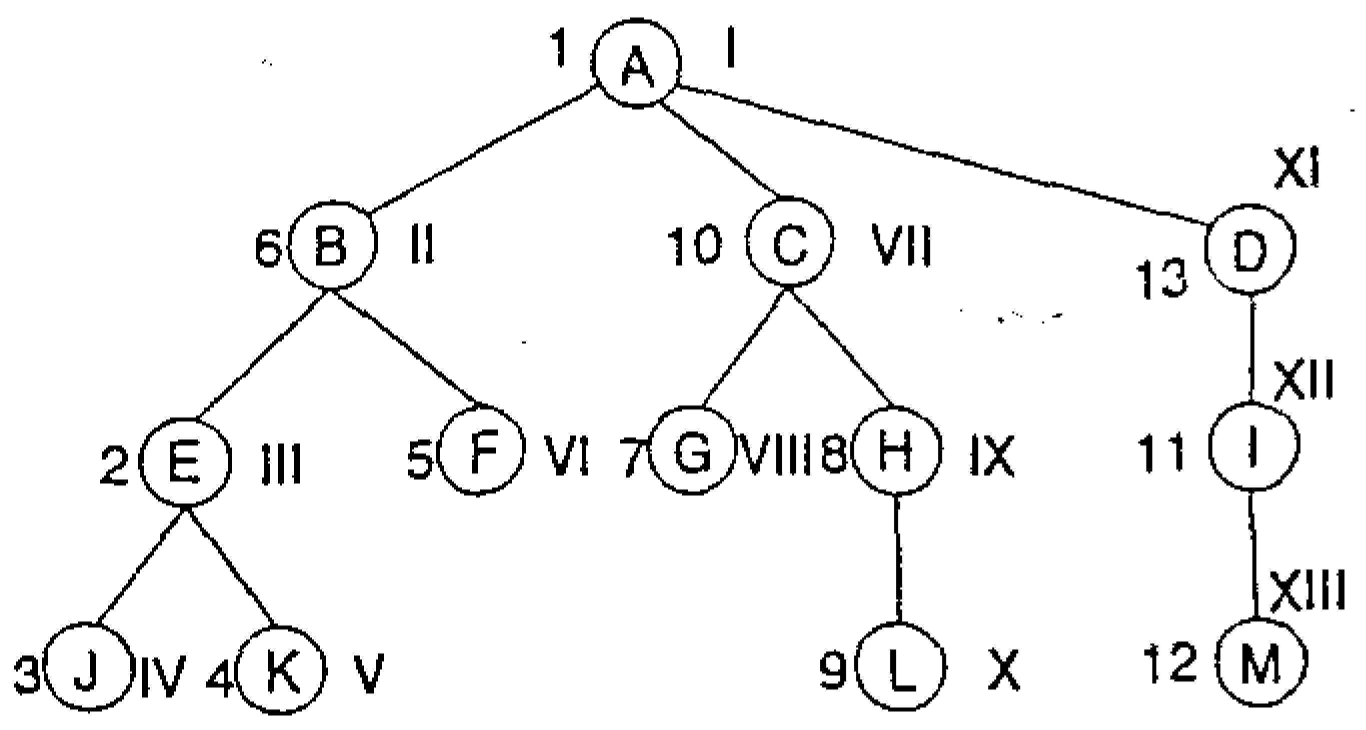


**Figure 5.1.1.: Two *STAS* Traversals of a Tree.**

**Theorem 5.1.** Relation STAS is an equivalence relation on the set of all possible traversals of an ODT

*Proof.*

In order to prove the previous theorem, one must prove that STAS is a reflexive (r), symmetrical (s) and transitive (t) relation.

(r) (∀T) STAS(T,T) holds trivially, by the definition of STAS relation.

(s) (∀T) (∀T') ( STAS(T,T') => STAS(T',T) ), again, is a trivial consequence of the definition of the relation STAS.

(t) (∀T) (∀T') (∀T") ( STAS(T,T') & STAS(T',T") => STAS(T,T") ). Let the order in which the subtrees will be taken on for the traversal by T be $S_i^1,\dots,S_i^n$, by T' $S_j^1,\dots,S_j^n$ and by T" $S_k^1,\dots,S_k^n$. Then by STAS(T,T'), we have that $S_i^1=S_j^1,\dots,S_i^n=S_j^n$. Also, by STAS(T',T"), we have that $S_j^1=S_k^1,\ldots,S_j^n=S_k^n$. Therefore, $S_i^1=S_k^1,\ldots,S_i^n=S_k^n$ and by the definition of STAS relation, we have also that STAS(T,T"). This completes the proof of the Theorem 5.1.1.

Since STAS is an equivalence relation on the set of all possible traversals of a FODT, it partitions the set onto its equivalence classes. Each of them contains all mutually STAS traversals and they have no common elements (they are disjunct). Figure 5.1.2. gives a graphical illustration of the effect of the relation STAS on the set of all possible traversals of a FODT.

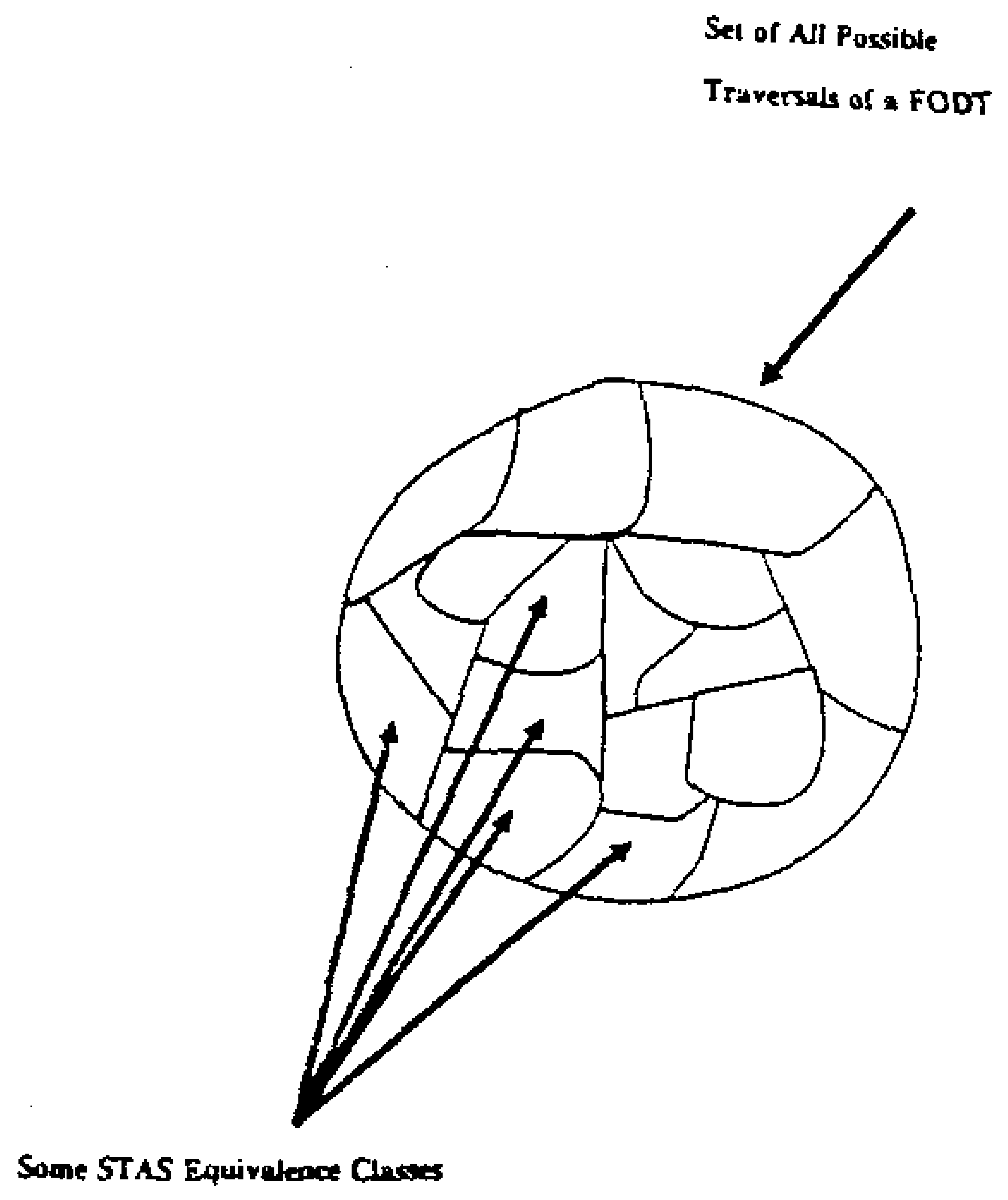


**Figure 5.1.2.:Partitioning and Equivalence Classes of STAS Relation.**

**Lemma 5.1.:** STAS traversals produce the same pair of corresponding constructs (a string and a corresponding semantics).

*Proof.*

This is a trivial consequence of the definition of the STAS relation. STAS traversals are traversals of the same ODT and therefore, just different ways of producing a same construct.■

The more representatives an algorithm has in different equivalence classes (the more classes it covers), the more of the desirable properties it possesses, as defined by the criteria from the previous sections. We will discuss some of them in more details with respect to the STAS relation later in this work.

To support the previous discussion with some "real life" examples and to illustrate how the STAS relation can be used for a comparison of grammar processing algorithms, we will introduce in Chapter 10 two well known parsing-generation algorithms and apply STAS relation to compare them.

# 6. STAS RELATION AND COMPLETENESS AND REVERSIBILITY CRITERIA FOR EVALUATION OF LOGIC GRAMMAR PROCESSING ALGORITHMS

## 6.1. STAS and Completeness Criterion

Completeness criterion as established in chapter 4, refers to the capacity of an algorithm to produce finite derivation trees for correct constructs of a language (strings or semantics). If one algorithm produces M finite derivation trees and the other one produces N such trees, then the relation between these two numbers (M and N can also be cardinal numbers for infinite sets, that are also comparable) determines which of the two algorithms is more complete.

**Theorem 6.1.1:** Let A1 and A2 be two grammar processing algorithms of the same kind and G an arbitrary logic grammar. Let also T be an arbitrary finite FODT corresponding to a derivation of some constructs in G (a string and a corresponding semantics, both fully instantiated, since it is a full ODT). If for each traversal T' produced by A1, there is a traversal T" produced by A2 and T' and T" are STAS, then A2 algorithm is at least as complete as A1 algorithm∎

*Proof.*

The claim of the theorem is an immediate consequence of the conditions that are imposed on A1 and A2 and the lemma 5.1. (the fact that two STAS traversals of a same FODT represent two derivations of a same construct)∎

This theorem claims that given two algorithms A1 and A2 that meet preconditions set forth in the theorem we can rank A1 and A2 with respect to the completeness criterion. Chapter 10 will present an example, when that is possible.

## 6.2. STAS and Reversibility Criterion

Reversibility criterion as presented earlier refers to the capacity in which an algorithm can be used to reproduce in the backward manner the original input from its normal output.

Intuitively, it is clear that the more STAS equivalence classes an algorithm covers, the better are the chances that some of the traversals will be usable in one and some in the other direction (namely, for parsing and generation). The following theorem states some conditions whose fulfillment establishes ranking of two algorithms with respect to the reversibility criterion.

**Theorem 6.2.1:** Let A1 and A2 be two grammar processing algorithms of the same kind and let G be an arbitrary logic grammar. Let also T be a finite FODT corresponding to a derivation of a string and a corresponding semantics (both fully instantiated) in G. If for each traversal T' produced by A1, there is a traversal T" produced by A2 and T' and T" are STAS, then A2 algorithm is at least as reversible as A1 algorithm∎

*Proof.*

Similarly as for theorem 6.1.1., this claim is an immediate consequence of the conditions that are imposed on A1 and A2 and the lemma 5.1. (fact that two STAS traversals of a same FODT represent two derivations of a same construct)∎

Thus, if by an analysis preconditions for theorem 6.2.1. can be guaranteed to hold, STAS relation can be used for ranking of two grammar processing algorithms with respect to the reversibility criterion.

# 7. SOUNDNESS CRITERION FOR RANKING OF LOGIC GRAMMAR PROCESSING ALGORITHMS

The soundness criterion as defined earlier in a rather general form, applies also to the situations when two algorithms overproduce from two possibly different or partially overlapping sets of incorrect constructs. The main metrics was the quantity of overproduced constructs by one, but not by the other algorithm. As a simple consequence of the soundness criterion definition, we have the following lemma that is sometimes used as a definition for the soundness criterion itself. The claim of the lemma assumes less generality than the definition 4.3.1. used here and its fulfillment implies the fulfillment of the conditions of the definition 4.3.1..

**Lemma 7.1.**: Let A1 and A2 be two parsing (or generation) algorithms. Let L1 and L2 be, respectively, sets of incorrect strings (or incorrect semantic structures) parsed (or generated) by A1 and A2 into some correct and meaningful semantic structures (or strings). If L1 $\supset$ L2, then the algorithm A1 is less sound than the algorithm A2$_\blacksquare$

*Proof.*

The condition $L1 \supset L2$ implies the condition *card(L1-L2)>card(L2-L1)* of the definition 4.3.1. and therefore by the definition A1 is less sound than A2$_\blacksquare$

# 8. EFFICIENCY CRITERION FOR RANKING OF LOGIC GRAMMAR PROCESSING ALGORITHMS

The efficiency criterion as established earlier in this study, refers to the number of production rules from the grammar, expanded (ended with success or failure) during a derivation produced by an algorithm. Intuitively, if for a moment we view a derivation tree as a dynamic process of building it (with trying edges, giving up on some and trying some other), the efficiency is measured by the number of edges tried (traversed) during this process. It is clear that the more deterministic choices an algorithm makes, less edges will be traversed in vain and therefore more efficient the algorithm will be.

For this criterion, the analysis of the manner in which a derivation tree was built is more important than how the actual tree looks like, i.e. how the corresponding FODT is traversed.

**Lemma 8.1.:** Let us assume that during the course of building of a derivation tree T' corresponding to a finite FODT T of size n, an algorithm A makes arbitrary non-deterministic choices for the next edge (production rule for expansion) among $p_i^1,\dots,p_i^m$ possibilities at nodes $v_i^1,\dots,v_i^m$, respectively. Taking into consideration only the nodes $v_i^1,\dots,v_i^m$, the number of edges traversed before the tree is completed, at worst is $p_i^1 * \dots * p_i^m$ (WCAN - worst case analysis number), on average it is $(p_i^1 * \dots * p_i^m)/2^m$ (ACAN - average case analysis number) and at best it is m (BCAN - best case analysis number)$_\blacksquare$

*Proof.*

WCAN and BCAN are trivially verified and in the on average case analysis we assume that the luck was only half on the side of the algorithm and half against it. That gave us $2^m$ factor$_\blacksquare$

We may note that the derivation number relative to the algorithm for the derivation in question (as defined in Chapter 4) is given as Q + n - m - 1, where $m \leq Q \leq WCAN$ (DNRA = Q + n - m - 1).

Obviously, if A1 shows better average case performance than A2, then A1 will perform better than A2 in the worst case as well. Also, vice versa, if A1 shows better worst case performance than A2, then A1 will perform better on average too. This is due to the fact that if ACAN1<ACAN2, then WCAN1<WCAN2 and vice versa, because WCAN = ACAN * $2^m$ and ACAN = WCAN / $2^m$.

This consistency between the worst and average case is an interesting property of algorithms for tree traversals and it is rarely present for other algorithms.

Also, WCAN, ACAN and BCAN are means that can be easily calculated and used for the analysis of a grammar processing algorithm efficiency-wise.

# 9. FINITENESS CRITERION FOR ESTIMATION OF LOGIC GRAMMAR PROCESSING ALGORITHMS

The finiteness criterion as we adopted it in Chapter 4, deals with the conditions, necessary and sufficient, that, when imposed on the grammar defining a language, will guarantee that a given algorithm will be producing only finite derivation trees for the grammar. Different algorithms will need different sets of necessary and sufficient conditions. These conditions imposed on a logic grammar, themselves define a class of grammars (and therefore also languages defined by the grammars) on which an algorithm is absolutely finite. The larger the corresponding class of grammars, the better the algorithm, finiteness-wise.

This criterion also affects the generality of the underlaying grammar. The grammars that fulfill conditions sufficient and necessary for an algorithm to be finite on them, might have to sacrifice their generality. They may no longer have enough capabilities for describing the entirety of a language and might be suitable only for a limited subset of it.

Because of the high complexity of the problem, only a limited number of results were reported on the matter. Most of these results were limited to stating sufficient conditions only. Dealing with only sufficient conditions unfortunately provides only a kind of worst case analysis of the problem. If certain sufficient conditions are fulfilled, the algorithm behaves in a proper manner producing only finite derivation trees. However, if these conditions are not fulfilled (and some other might be), the algorithm still might be able to produce only finite trees.

One of the most significant attempts with respect to this criterion are results published by Dymetman, Isabelle and Perrault in [DIP90].

## 9.1. The Approach with Guides

In the paper by Dymetman, Isabelle and Perrault ([DIP90]), the finiteness criterion was addressed by introduction of the notion of guides. Guides are variables added to a logic grammar as a piece of redundancy, that as we stated before, can be exploited for tighter control of the computational process.

The main result was stated for a quite general class of definite clause programs. These programs were understood to introduce terminals only at the lexical level (when the lexical predicates were called for expansion). First step, as previously shown by the figure 1.2.1.2.1. (page 9), was to introduce guides to the original program DCPIGi, creating its new version DCPIG'. DCPIG' was either left-recursion free, or it was subjected to a known transformation for the elimination of the left-recursion, in order to produce another program DCPIGo. Then, if the guide consumption condition (GCC) and no-chain condition (NCC) held for DCPIG', ordinary depth-first-search, left-to-right (Prolog) algorithm was guaranteed to produce only finite derivation trees, when applied to DCPIGo.

Guide consumption condition basically stated that the value for guide variables was initially finite and consumed each time a lexical predicate was called for the expansion. No-chain condition prohibited exclusive appearance of predicates like T=U on a right-hand side of a rule.

The paper presented application of the main results to a rather restrictive class of lexical grammars and achieved a symmetrical treatment of parsing and generation process.

The following improvements look desirable with respect to this result: (i) Allow consumption of guides at any level, not only lexical, (ii) Be more specific about the guides, instead of leaving the details to the moment when more information about algorithm and the underlaying grammar is available and (iii) State the main result (concerning finite derivation) not only with respect to the depth-first-search, left-to-right

(DFLR) algorithm for the evaluation of a grammar, then with respect to any grammar processing algorithm.

## 9.2. The Approach with Universal Guides

As we pointed out earlier in this study, a grammar processing algorithm is considered finite if it produces only finite derivation trees. Notion of producing or discovering a finite derivation tree could be viewed as discovering a finite set of all variables taking part in the derivation, uninstantiated initially. These are getting their values gradually during a finite derivation, by applying appropriate production rules and eventually, the set of uninstantiated variables is reduced to an empty set.

Thus, another view of the derivation process would be as of a discovering of the set of all variables taking part in the derivation, that are uninstantiated at the moment when they are introduced into the derivation by applying a production rule. They might or might not get instantiated during the derivation and their number could be finite (finite derivations with finite derivation tree), or infinite (infinite derivation with infinite derivation tree). Set (finite or infinite) of all these variables has all properties of a partially ordered system (relation for this partially ordered structure is "being a subset" $\subset$). We formalize this notion through the following definitions.

DEFINITION 9.2.1. (AN USEFUL PARTIALLY ORDERED RELATION)

Let S and S' be two sets and N and N' two non-negative integers. We say that ordered pair (S,N) is *less than or equal to* ordered pair (S',N') and write $(S,N)\leq(S',N')$ iff $S\subset S'$ or $((S=S') \& (N\geq 0) \& (N'\geq 0) \& (N\leq N'))$∎

It is obvious that $\leq$ is a reflexive, anti-symmetrical and transitive relation and therefore a relation of partial order.

DEFINITION 9.2.2. (UNIVERSAL GUIDES)

Let G be an unification-based grammar. Universal guide structure for grammar G is a partially ordered set UG of ordered pairs (G,N). G is a set of some of the variables that were uninstantiated initially, when they appeared for the first time in a derivation in G and it is called *set component* of the pair (G,N). N is a non-negative integer and it is called *number component* of the pair. Relation for comparison of the elements from G is ≤ (from the definition 9.2.1.)∎

Let us note that if the algorithm in question has property of being finite, then the set of variables taking part in the derivation will always be finite. Therefore, since only finite sets are in play now, collection of subsets of all variables taking part in a derivation is a *proper guide*-structure. By proper guide structure we mean guide structure as it was introduced by Dymetman et al. (i.e. a partially ordered system in which every ordered chain is finite and all initial values for guides are finite). Since all these sets are finite, the condition that each strictly decreasing ordered chain is finite, is fulfilled, too, provided that the number component of the guide structure is initially instantiated in a proper fashion, as we will demonstrate later. Suggested guide variables are therefore ordered pairs whose set components are sets of currently uninstantiated variables in the derivation that have initially entered the derivation as uninstantiated, accompanied with the number components. As we will show only finite number of times is possible that by an application of a production rule none of the variables get instantiated. In such situations in order to prove that *guide consumption condition* is always respected, we exploit the number component of the universal guides.

This collection of sets of uninstantiated variables involved in a derivation is therefore a basis of an universal guide (UG) structure, that will sometimes, under certain conditions imposed on the underlaying grammar, possess the property of "only finite strictly decreasing chains" and therefore become a proper guide (PG), as Dymetman, et al. defined it.

The presence of universal guides in a grammar is almost always implicit, but for the sake of proving facts about them, it can be made explicit, as well. Next, we sketch a procedure for making their presence explicit in a grammar. The language defined by the new grammar will remain the same as the language defined by the original grammars. The new variables are just pieces of redundancy that can be conveniently exploited.

Before adding new material to the grammar, let us say that all arguments in predicates of the grammar must be renamed prior to this procedure, so no two predicates share a name for one of their arguments. One way to do this would be to index arguments of a predicate by the predicate's name, i.e. $X_1^p,\ldots,X_n^p$ for the predicate *p*. Also, we omit details of our list representation for sets and set relations and operations ($\cup$, $\subset$, $\supset$, $\subseteq$, $\supseteq$, -).

Next we explain the procedure:

(1) The following are additional arguments for each literal in the grammar:

| | |
|---|---|
| *Inst* - | set of all instantiated arguments in this literal for a particular rule in question. Each left-hand side literal of a rule will have variable *Inst* instantiated to a set of literal's arguments that are here instantiated before the rule gets expanded. |
| *UnInst* - | set of all uninstantiated arguments in this literal for a particular rule in question. Each left-hand side literal of a rule will have variable *UnInst* instantiated to a set of literal's arguments that are here uninstantiated before the rule gets expanded. |
| *UnIn* - | set of all variables that were initially (when they entered the derivation) uninstantiated and are uninstantiated before this rule gets expanded - In guide variable (set component). |
| *UnOut* - | set of all variables that were initially (when they entered the derivation) uninstantiated and are still uninstantiated after this rule gets expanded - Out guide variable (set component). |

| | |
|---|---|
| *InstGl* - | variables that participate in the derivation and are currently instantiated. |
| *Num* - | auxiliary non-negative integer used for the relation $\leq$ - In guide variable (numeric component). |
| *Num1* - | auxiliary non-negative integer used for the relation $\leq$ - Out guide variable (numeric component). |

(2) Start always with invoking the topmost predicate as follows:

(a) In place of *Inst*, have a set containing $x_{1,i},\ldots,x_{k,i}$, where $X_{1,i},\ldots,X_{k,i}$ are all arguments instantiated initially (the sets should be implemented in a way that they can accept more elements later during the derivation process).

(b) In place of *UnInst*, have a set containing $x_{1,j},\ldots,x_{l,j}$, where $X_{1,j},\ldots,X_{l,j}$ are all arguments uninstantiated initially.

(c) In place of *UnIn*, have a set containing $x_{1,j},\ldots,x_{l,j}$.

(d) In place of *UnOut*, have a set containing $x_{1,i},\ldots,x_{k,i},x_{1,j},\ldots,x_{l,j}$.

(e) In place of *InstGl*, have an empty set.

(f) In place of *Num*, have the constant *n*, equal to the number of rules in the grammar.

(3) Add the following two rules to the grammar:

decrease(A,B,N,N1) --> $A \subset B$, N1 becomes n. (n is number of rules in the grammar).

decrease(A,B,N,N1) --> not($A \subset B$), N1 becomes N-1.

This predicate will ensure that a *proper ordering* is achieved even for rules that do not instantiate a single variable.

(4) Each rule with a right-hand side as

$p(\alpha_1,\ldots,\alpha_r)$ --> $q_1(\beta_{1,1},\ldots,\beta_{1,n}),\ldots,q_n(\beta_{n,1},\ldots,\beta_{n,m})$, ($\alpha$'s and $\beta$'s represent instantiated as well as uninstantiated literals) is replaced by

$p(\alpha_1,\ldots,\alpha_r$,Inst$^p$,UnInst$^p$,UnIn,UnOut,InstGl,Num,Num1) -->

UnIn becomes UnIn$\cup$(UnInst$^p$-InstGl),

UnOut becomes UnOut-Inst$^p$,

InstGl becomes $\text{InstGl} \cup \text{Inst}^p$,

$q_1(\beta_{1,1},\ldots,\beta_{1,n},\text{Inst}^1,\text{UnInst}^1,\text{UnIn},\text{UnOut}^1,\text{InstGl},\text{Num},\text{Num}_1),\ldots,$

$q_n(\beta_{n,1},\ldots,\beta_{n,m},\text{Inst}^n,\text{UnInst}^n,\text{UnIn}^{n-1},\text{UnOut}^n,\text{InstGl},\text{Num}_{n-1},\text{Num}_n),$

UnOut becomes $(\text{UnOut} \cup \text{UnOut}^n)\text{-InstGl}$,

decrease(UnIn,UnOut,$\text{Num}_n$,Num1).

(5) Each rule with no right-hand side as

$p(\alpha_1,\ldots,\alpha_r)$, ($\alpha$'s represent instantiated as well as uninstantiated literals) is replaced by

$p(\alpha_1,\ldots,\alpha_r,\text{Inst}^p,\text{UnInst}^p,\text{UnIn},\text{UnOut},\text{InstGl},\text{Num},\text{Num1})$ -->

UnIn becomes $\text{UnIn} \cup (\text{UnInst}^p\text{-InstGl})$,

UnOut becomes $\text{UnOut-Inst}^p$,

InstGl becomes $\text{InstGl} \cup \text{Inst}^p$,

decrease(UnIn,UnOut,Num,Num1).

Here, *Inst* and *UnInst* are corresponding arguments of the left-hand side predicate of this rule and *UnIn, UnOut* and *InstGl global* variables. We mean *global* in the sense that they appear in every predicate of the original grammar. This step describes the process of variables getting instantiated during the unification of a predicate with a left-hand side of a production rule.

The following theorem makes a connection between *proper guides* (from [DIP90]) and *universal guides*, introduced here.

**Theorem 9.2.1.:** If there is a proper guide structure for a logic grammar (in its appropriate form from [DIP90]) satisfying GCC and NCC, then the universal guide structure, under the DFLR algorithm, is a proper guide structure, too∎

*Proof.*

By the main theorem from [DIP90], finite derivation trees are guaranteed under GCC and NCC, when DFLR algorithm is applied to the logic program. That in turn means that a finite set of variables that is set of all variables entering a derivation uninstantiated, exists and will be discovered by DFLR algorithm. Since the derivation in question is an arbitrary one, this means that the sets of variables will always be finite. During any derivation in the grammar, some production rules are applied that instantiate zero or more variables taking part in the derivation. Rules that instantiate variables, decrease the size of the set of currently uninstantiated variables, that is initially finite. Thus, by applying these rules only, the set of uninstantiated variables gets reduced to an empty set after a finite number of steps. However, it is also possible that during a finite derivation (by assumption, all derivation here are finite) rules that do not instantiate any variable will be applied, too. Again, since the derivation is finite, that can happen only a finite number of times. In fact, if $n$ is number of rules in the grammar, the number of times that rules not instantiating variables can be consecutively applied, is less than or equal to $n$. The reason for this is that application of more than $n$ non-instantiating rules would mean that at least one of them repeats and consequently, infinite recursion occurs, contradicting the fact that the derivation is finite. Thus, by instantiating *Num* component to $n$ initially, resetting it to $n$ every time a variable gets instantiated and decreasing it every time when no variable is instantiated by application of a non-instantiating rule, we ensured that every *strictly decreasing ordered chain* of guides will be finite. If only rules that instantiate a variable are applied, that is obvious, because of the nature of relation $\subset$. When a non-instantiating rule is applied, then the numeric component of guide is decreased by one. Since starting value is always $n$ and one can not apply more than $n$ such consecutive rules, the value cannot go under zero and therefore every such strictly decreasing, ordered chain is finite, too. Therefore, universal guide structure is a proper guide structure, as well$_\blacksquare$

The notion of universal guides is more general than the notion of guides as introduced by Dymetman, et al. in the following sense: It does not assume a particular algorithm under which a grammar will be processed. Thus, it is applicable to any algorithm and to apply it, would mean to specify conditions on grammars that would guarantee finiteness of the forementioned sets (universal guides). Of course, the nature of the conditions will depend on the nature of the algorithm.

Guides proposed by Dymetman, et al., if they could be introduced at all, guarantee finiteness of one specific (DFLR) algorithm if the grammar fulfills GCC and NCC.

Also universal guides treat parsing and generation in a symmetrical way, as it was done by *proper guides*, but unlike with the *proper guides*, universal guides need not to be instantiated and created differently for parsing and generation. Guides consumption is allowed at all levels, not only at the lexical level. Also by the previous theorem, *universal guides* will always be an usable structure, when *proper guides* work. Thus, they are at least as usable as proper guides.

An example of the application of the approach with the universal guides is the theorem 9.2.2. that assesses the finiteness of Essential Arguments Algorithm (EAA), described in depth in [S90a], [S90b] and [S91]. Before the theorem is stated, let us point out what some of the notions used in the theorem represent.

Minimal set of essential arguments for a logic grammar non-terminal is a set of some of its arguments whose instantiation is sufficient for its successful expansion and there is no proper subset of this set with the same property. A logic grammar symbol can have more than one minimal set of the essential arguments. For more details about this and related notions, [S90a], [S90b] and [S91] are suggested references.

Next, we state the theorem as an illustration of the UG approach.

**Theorem 9.2.2.:** Let S be starting symbol of a logic grammar G and let U be set of all initially uninstantiated variables in S. If there is a S production rule with its right-hand side having a permutation of logic symbols such that each symbol, when it gets its turn for expansion has at least one of its minimal sets of essential arguments instantiated and the same is true for any subsequent logic symbol when it gets its turn for the expansion, EAA will produce only finite derivation trees for this grammar∎

*Proof.*

By the definition of *minimal sets of essential arguments*, they guarantee finite expansion of the corresponding LG symbols. Finite expansion for each rhs LG symbol of an S production rule means also finite expansion of the entire production rule. Assume that the number of different production rules in G is *n*. Then, in a finite derivation there cannot exist a sequence of more than *n* rules that do not instantiate any variable. Otherwise, a production rule would be repeated in the same manner without instantiating any of the present variables which would in turn cause an infinite derivation. Therefore we can use the universal guides as they were defined by Definition 9.2.2.. Moreover, instantiating *Num* component to *n* initially, resetting it to *n* every time a variable gets instantiated and decreasing it every time when no variable is instantiated by application of a non-instantiating rule, we ensure that every *strictly decreasing ordered chain* of universal guides will be finite. The argument for this is identical to the one in the proof of the Theorem 9.2.1. Then, since application of EAA (to the new grammar it creates from the original one) is actually application of TDLR evaluation strategy and the universal guides are also proper guides, finite derivation trees are guaranteed∎

Thus, the notion of universal guides provides us with a choice, alternative and more general, when trying to assess efficiency of a grammar processing algorithm.

# 10. TWO GRAMMAR PROCESSING ALGORITHMS AND THEIR COMPARISON

## 10.1. Semantic-Head-Driven-Generation Algorithm (SHDGA)

## vs.

## Essential Arguments Algorithm (EAA)

Recently, two important new algorithms have been published ([SNMP89], [SNMP90], [S90a], [S90b] and [S91]) that address the problem of automated generation of natural language expressions from a structured representation of meaning. Both algorithms follow the same general principle: given a grammar and a structured representation of meaning, produce one or more corresponding surface strings and do so with a minimal possible effort. In this study we limit our analysis of the two algorithms to unification-based formalisms.

### 10.1.1. Introduction to SHDGA and EAA

In their first phase, the algorithms do certain amount of preprocessing on the underlaying grammar in order to be able to apply their respective evaluation strategies later.

The first algorithm, which we call here the Semantic-Head-Driven Generation Algorithm (SHDGA), uses information about semantic heads in grammar rules to obtain the best possible traversal of the generation tree, using a mixed top-down/bottom-up strategy. In its preprocessing phase SHDGA compiles grammar rules in one of two ways, depending on whether the rule in question is a "chain rule" or not. A chain rule is one in which the semantics of the left-hand side (abbreviated here as lhs) is identical to the semantics of some right-hand side (rhs) constituent, the rule's "semantic head". Here, the algorithm needs an explicit indication of what the semantics is, for a given grammar symbol. In two major papers on SHDGA, [SNMP89] and [SNMP90], this indication is given by introducing "/" symbol, that separates syntactic from semantic part of a grammar symbol (*SYMSYN/SYMSEM*). Thus, what follows "/" symbol is considered semantics of a given grammar symbol. In addition, the *"chained"* relation, the transitive closure of the syntax part of the semantic head relation is computed. That is, if *HEADSYN/HEADSEM* is the semantic head of *LHSSYN/LHSSEM*, then *HEADSYN* and *LHSSYN* are in the *"chained"* relation. Therefore, preprocessing phase of SHDGA consists of introducing a notation that explicitly indicates what semantics is for a given symbol, aligning the grammar rules into "chain rules", or "non-chain rules" group and computing the relation *"chained"*. We can talk of this preprocessing phase as of compilation phase of the algorithm. In the evaluation phase of the algorithm, SHDGA is given a grammar symbol (goal) with its semantics fully instantiated and is expected to discover corresponding surface string(s), if such exists. SHDGA selects a "non-chain" rule, whose left-hand side symbol (pivot) has semantic component unifiable with the semantics of the given goal and pivot can be connected to the goal by a sequence of "chain rules". That rule is expanded and the same procedure is applied to the grammar symbols on the right-hand side of it, if there are right-hand side symbols. Since the rule is "non-chain", no right-hand side symbol can have the same semantics as pivot. After the expansion of the "non-chain" rule is completed, its left-hand side symbol will get connected with the goal, by applying a sequence of "chained rules". Thus, next, a "chain rule" is selected whose left-hand side symbol has the same semantics as pivot's semantics and it is possible to connect it to the goal by a sequence of "chain rules". The same procedure is applied to the pivot's siblings on the right-hand side of this rule. The application of "chain rules" is continued in the same manner until at one moment, the left-hand side symbol of such a "chain rule" is actually the goal symbol. The application of this SHDGA's evaluation strategy can best be described by the way this algorithm traverses analysis trees, that we will explain shortly in this section.

The second algorithm, which we call the Essential Arguments Algorithm (EAA), rearranges grammar productions at compile time in such a way that the evaluation (even one like a simple top-down left-to-right evaluation) will follow an optimal (or nearly optimal) path. By optimal path we mean a path in which no unsuccessful expansions will ever occur, or in terms of traversing of an analysis tree, no backtracking ever happens. In its preprocessing phase, EAA computes the so called minimal sets of essential arguments (abbreviated as msea's) for each of grammar symbols. If variables from a minimal set of essential arguments of a grammar symbol are all instantiated, a finite deterministic expansion of the grammar symbol is possible. Since we assume a finite deterministic expansion, no wrong moves (and backtracking) is allowed. An expansion of that kind can end with a failure or a success, but it is an expansion with no guessing. Thus, when used for generation, EAA will assume that the semantics of the topmost grammar symbol is instantiated and will try to rearrange rhs grammar symbols in rules where the topmost symbol is on the left-hand side in such a manner that whenever a symbol is next for expansion, at least one of its msea's is instantiated. If such ordering(s) of grammar symbols exists, it is (they are) adopted and EAA actually creates a (or several) grammar(s) equivalent to the original one and executable in a top-down, left-to-right fashion. If such reordering of grammar symbols is not possible, EAA will apply so called inter-clausal reordering (abbreviated as ICR). It will *lift up* a grammar rule having a rhs symbol of the currently processed rule on its left-hand side, by replacing the symbol in the current rule with the right-hand side symbols of its rule. Then, for this new rule, it will try reordering of its symbols along the path of instantiated msea's and if that is now possible, this new rule is added to the grammar. If that is still not the case, it will try to further process this rule in the same (ICR) fashion, until the ordering along the msea's path is possible. EAA therefore creates a grammar that is equivalent to the original one, that possibly contains new rules and that possibly has different ordering of grammar symbols. The new grammar is ready for top-down, left-to-right execution. In the following sections of this paper EAA is described and explained through the manner in which it traverses analysis trees.

Both algorithms have resolved several outstanding problems in dealing with natural language grammars, including handling of left recursive rules, non-monotonic compositionality of representation and deadlock-prone rules. Here, we clarify and expand our comparison of these two algorithms given in [MS92]. We consider their capabilities with respect to generality, completeness, efficiency and reversibility criteria.

We treat cases when they find optimal msea-lead traversals, as well as cases when only non-optimal ones can be found, giving a complete overview of their properties in different situations. We also explain how the inter-clausal inversion procedure, added to the initial version of EAA, contributes to the generality of EAA, making it applicable always when a finite derivation tree for a language construct (surface string or semantics) is possible.

SHDGA traverses the derivation tree in the semantic-head-first fashion. Starting from the goal predicate node (called the "root"), containing a structured representation (semantics) from which to generate, it selects a production whose left-hand side semantics unifies with the semantics of the root. If the selected production passes the semantics unchanged from the left to some nonterminal on the right (the so-called "chain rule"), this later nonterminal becomes the new root and the algorithm is applied recursively. On the other hand, if no right-hand side literal has the same semantics as the root (the so called non-chain rule), the production is expanded and the algorithm is recursively applied to every literal on its right-hand side. SHDGA first selects a non-chain rule for expansion. That means it visits the node in the tree representing the lhs literal of the non-chain rule and then it is recursively applied to the nodes representing the rhs literals of the same rule. When the evaluation of a non-chain rule is completed and all nodes in the subtree rooted at the node representing its lhs literal are visited, SHDGA connects its left-hand side literal (called the "pivot") to the initial root using (in a backward manner, going bottom-up) a series of appropriate chain rules, one at a time. At this time, all remaining literals in the chain rules are expanded in a fixed order (say left-to-right). The subtrees rooted at the nodes corresponding to them are traversed using the same algorithm recursively.

Since SHDGA traverses the derivation tree in the fashion described above, this traversal can be considered neither top-down (TD), nor bottom-up (BU), nor left-to-right (LR) globally, with respect to the entire tree. However, since there is nothing inherent in SHDGA that suggests how the siblings of the semantic head literal are selected for expansion on the right-hand side of a chain rule, or how a non-chain rule is evaluated, it is usually implemented to be LR in those situations and we can say that it is LR locally. In fact the overall traversal strategy combines both the TD mode (non-chain rule application) and the BU mode (backward application of chain rules).

EAA takes a unification grammar (usually Prolog-coded) altering the order of right-hand side nonterminals in rules. It reorders literals in the original grammar in such a way that the *msea-lead traversal order* is achieved for a given evaluation strategy (eg. top-down left-to-right), if such an ordering exists. By msea-lead traversal order we mean order in which every literal has at least one of its mseas instantiated when it gets its turn for expansion. This restructuring is done at compile time, so in effect a new executable grammar is produced. If there are more than one msea instantiated at a moment when the next one for expansion is to be selected, then EAA branches and it may choose any ordering of literals with instantiated msea's. In such cases it actually creates more than one grammar equivalent to the original one and they differ from the original one only in ordering of literals. The resulting parser or generator is TD but not LR with respect to the original grammar. However, the new grammar is evaluated TD and LR (i.e., using a standard Prolog interpreter). As a part of the node reordering process EAA calculates the minimal sets of essential arguments for all literals in the grammar, which in turn will allow to project an optimal msea-lead evaluation order (in which there will be no backtracking). The optimal evaluation order is achieved by creating the so called msea-lead order, expanding only those literals which are "ready" at any given moment, i.e., those that have at least one of their msea's instantiated.[2] Sometimes however, reordering of literals in which each is "ready" when its turn for expansion comes, does not exist and then EAA will invoke its inter-clausal reordering procedure. Thus, in such a situation the reordering is done both locally within each rule and globally between different rules. Then, the ordering achieved by the algorithm will be optimal msea-lead with respect to a new (by EAA created) grammar that is equivalent to the original one.

[2]Obviously, a msea-lead order (traversal) is also optimal, because the determinism of msea's guarantees that there will be no wrong moves and backtracking. On the other hand, it is also clear that there may be optimal traversals which are not msea-lead (for instance those lead by correctly guessing the bindings of variables for grammar symbols which do not have their msea's instantiated at the moment of visit).

Inter-clausal reordering (ICR) generalizes and exploits technique described in [S91] and also known as the *lifting transformation*. ICR creates some new grammar rules that can be evaluated in a top-down, left-to-right fashion, adding some redundancy to the initial grammar that can be exploited for computational purposes. With respect to the original grammar, this technique is nothing else than partial evaluation technique as described by Shieber, Pereira, et al.. We think that calling it inter-clausal reordering is here more appropriate taking into consideration the context in which is used and that is called upon only when intra-clausal reordering is not possible. It is implemented as a compile-time means. When EAA discovers that there is no reordering of literals where there is always a *ready* literal at the moment for the next expansion (msea-lead ordering), then it picks up *the most instantiated* literal to perform the lift up transformation on it. This transformation consists of replacing the corresponding node in the analysis tree with the subtree rooted at it. If that does not yield a tree for which at the given level mseas ordering can be achieved, the procedure is continued. The procedure could be continued by doing the same for another (*second most instantiated*) literal at the same level, or for *the most instantiated* one at one level below (that was just lifted up).

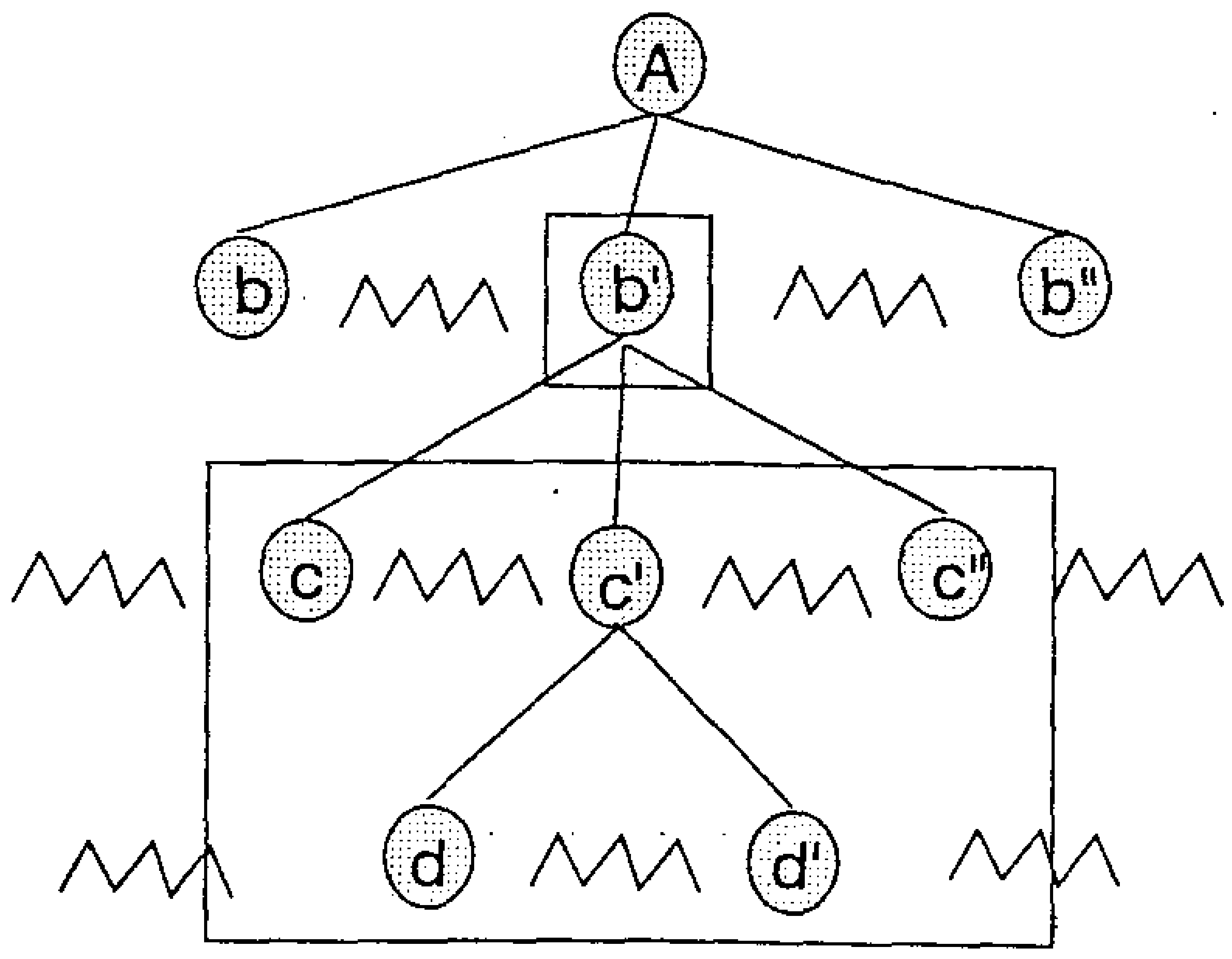


**Figure 10.1.1.1.: A Tree Before A *Lift Up Transformation.***

Eventually, this will result in creating a tree that can be used for the same derivation and for which a msea-lead reordering exists. The whole procedure, when implemented, creates new rules for a given grammar until with the new rules msea-lead traversal exists. The new rules are just another way of using the same initial grammar and do not change the language defined by the grammar.

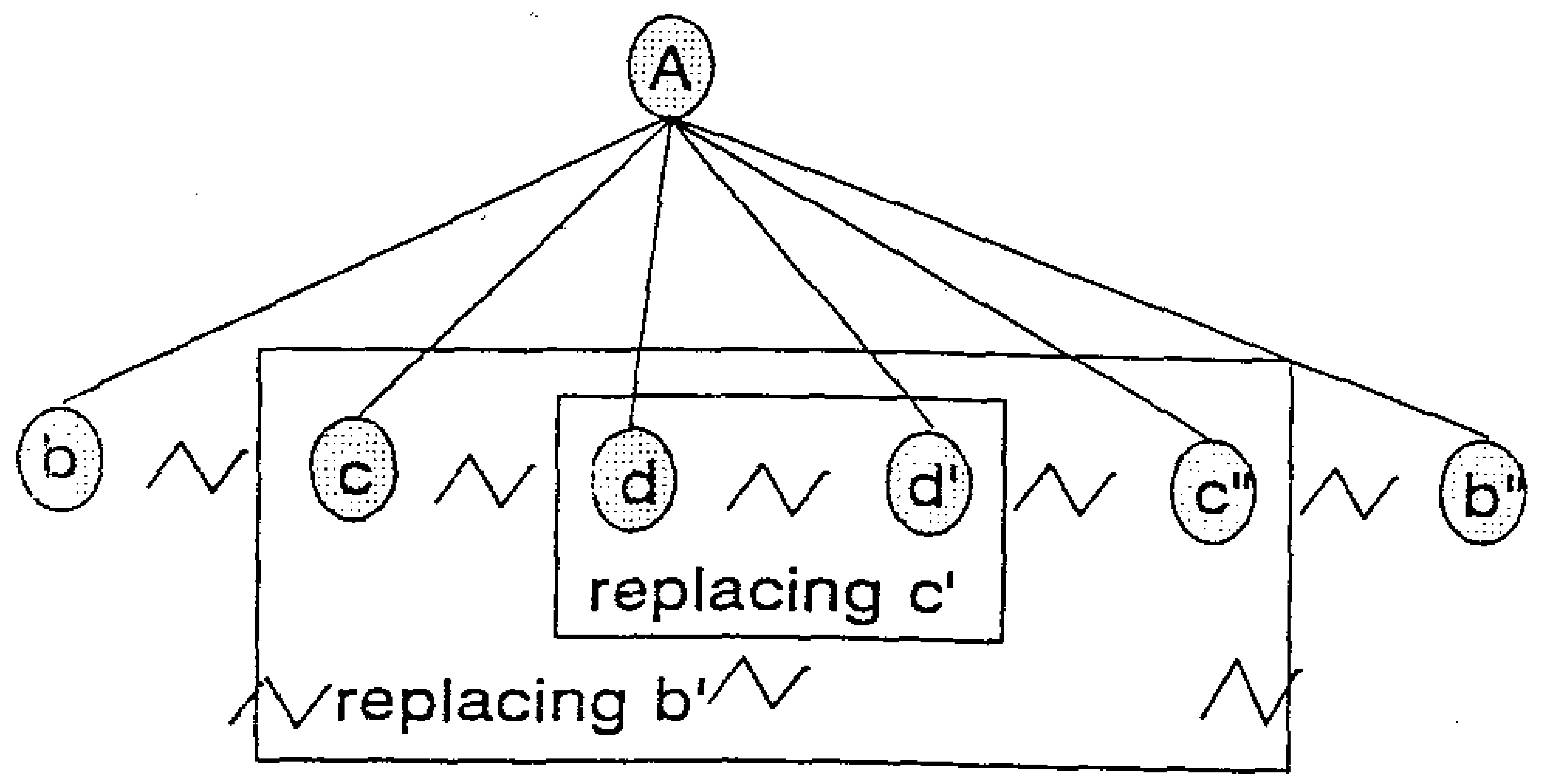


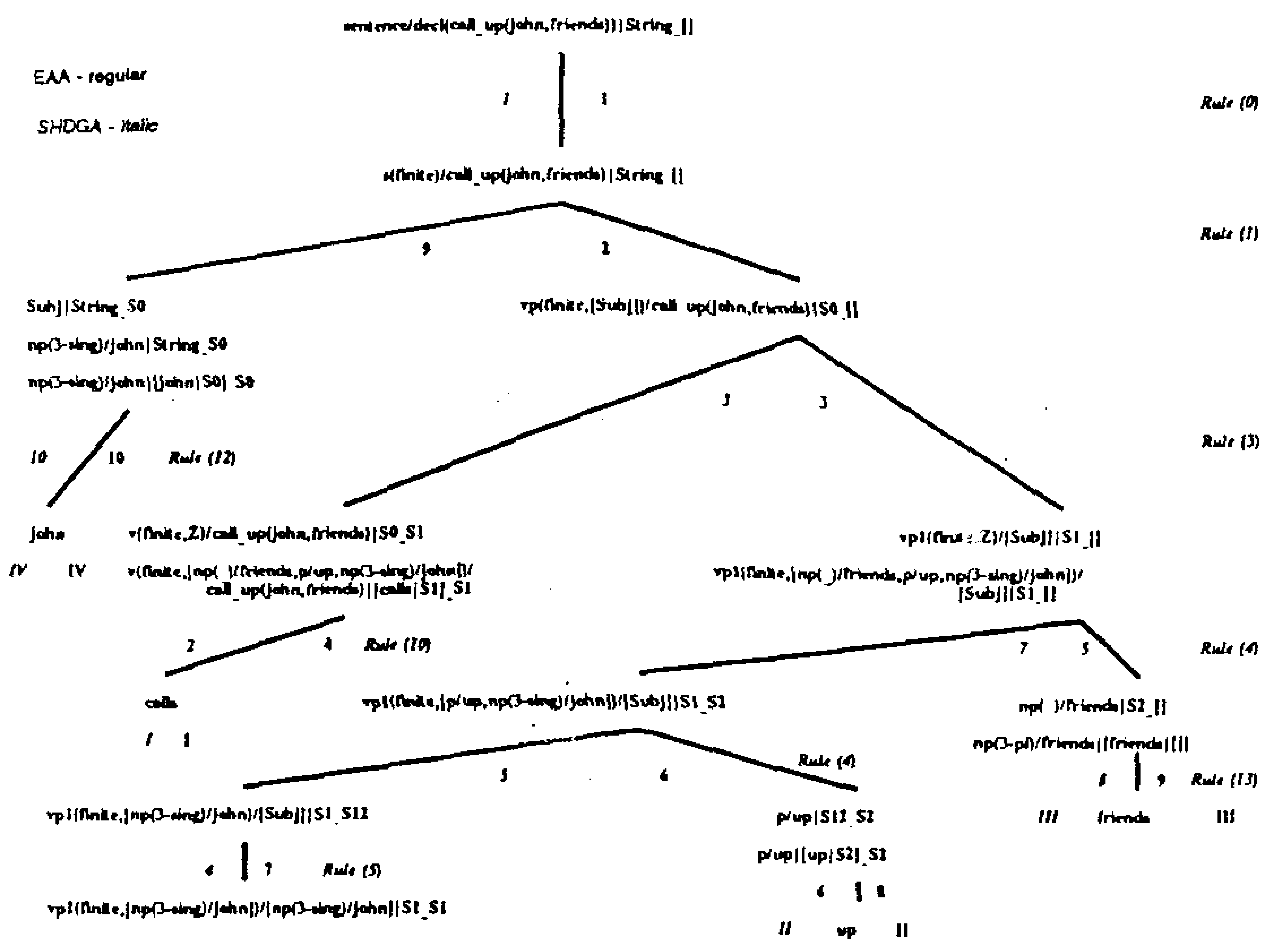
Figure 10.1.1.2.: Tree From Figure 10.1.1.1. After A *Lift Transformation.*

To make the argument more concrete and more visual, assume that the tree given in figure 10.1.1.1. represents an analysis tree for a derivation and that none of the literals *b,..,b',..,b"* is *ready* when the expansion turn for one of them comes. We assume that *b'* is the *most instantiated* one among them and therefore the one on which *lift up transformation* will be tried first. That would mean replacing *b'* with the subtree rooted at it, so an intermediate tree with *c,..,c',..c" lifted up* to the level of *b*'s is obtained. Now, a msea-lead reordering will be attempted among *b,..,c,..,c',..,c",..b"*. Suppose that is impossible too. Then, if *c'* happens to be *the most instantiated* literal among *b,..,c,..,c',..,c",..b"*, *c'* will be next for the *lift up transformation*. It will get replaced by the subtree rooted at it, therefore yielding the tree from figure 10.1.1.2.. This procedure will be continued until a msea-lead reordering is possible. That is certain eventually to happen, as we will show later in this study. If there is a msea-lead ordering after *c*'s and *d*s were lifted to the level of *b*'s, in the EAA msea-lead traversal of the original tree from figure 10.1.1.1. *d*s are to be traversed before *c'* and *c*'s are to be traversed before *b'*.

The following example illustrates the traversal strategies of both algorithms. The grammar is taken from [SNMP90] and "normalized" in order to simplify the exposition.

```
.......
sentence/decl(S) -->        s(finite)/S.                                              (0)
sentence/imp(S) -->         vp(nonfinite,[np(_)/you])/S.                              (1)
.......
s(Form)/S -->               Subj, vp(Form,[Subj])/S.                                  (2)
.......
vp(Form,Subcat)/S -->       v(Form,Z)/S, vp1(Form,Z)/Subcat.                          (3)
vp1(Form,[Compl|Z])/Ar -->  vp1(Form,Z)/Ar, Compl.                                    (4)
vp1(Form,Ar)/Ar.                                                                      (5)
.......
vp(Form,[Subj])/S -->       v(Form,[Subj])/VP, aux(Form,[Subj],VP)/S.                 (6)
.......
aux(Form,[Subj],S)/S.                                                                 (7)
aux(Form,[Subj],A)/Z -->    adv(A)/B, aux(Form[Subj],B)/Z.                            (8)
.......
v(finite,[np(_)/O,np(3-sing)/S])/love(S,O) -->           [loves].                     (9)
v(finite,[np(_)/O,p/up,np(3-sing)/S])/call_up(S,O) -->   [calls].                    (10)
v(finite,[np(3-sing)/S])/leave(S) -->                    [leaves].                   (11)
.......
np(3-sing)/john -->                                      [john].                     (12)
np(3-pl)/friends -->                                     [friends].                  (13)
.......
adv(VP)/often(VP) -->                                    [often].                    (14)
```

The derivation tree for both algorithms is presented below (figure 10.1.1.3.). The input semantics is given as *decl(call_up(john,friends))*. The output string becomes *john calls up friends*. The difference lists for each step are also provided. They are separated from the rest of the predicate by the symbol |. The different orders in which the two algorithms expand the branches of the derivation tree and generate the terminal nodes are marked, in italics for SHDGA and in roman case for EAA. The rules that were applied at each level are also given.



sentence/decl(call_up(john,friends))|String_[]
EAA - regular
SHDGA - italic
*1*
1
*Rule (0)*
s(finite)/call_up(john,friends)|String_[]
*9*
2
*Rule (1)*
Subj|String_S0
np(3-sing)/john|String_S0
np(3-sing)/john|[john|S0]_S0
*10*
10
*Rule (12)*
vp(finite,[Subj])/call_up(john,friends)|S0_[]
*3*
3
*Rule (3)*
john
*IV*
IV
v(finite,Z)/call_up(john,friends)|S0_S1
v(finite,[np(_)/friends,p/up,np(3-sing)/john])/call_up(john,friends)|[calls|S1]_S1
vp1(finite,Z)/[Subj]|S1_[]
vp1(finite,[np(_)/friends,p/up,np(3-sing)/john])/[Subj]|S1_[]
*2*
4
*Rule (10)*
*7*
5
*Rule (4)*
calls
*I*
1
vp1(finite,[p/up,np(3-sing)/john])/[Subj]|S1_S2
np(_)/friends|S2_[]
np(3-pl)/friends|[friends|[]]
*8*
9
*Rule (13)*
*III*
friends
III
*5*
6
*Rule (4)*
vp1(finite,[np(3-sing)/john])/[Subj]|S1_S12
*4*
7
*Rule (5)*
vp1(finite,[np(3-sing)/john])/[np(3-sing)/john]|S1_S1
p/up|S12_S2
p/up|[up|S2]_S2
*6*
8
*II*
up
II

Figure 10.1.1.3.: SHDGA's and EAA's Traversals.

EAA can also produce the same output string, but the order in which nodes *vp1(finite,[p/up,np(3-sing)/john])/[Subj]/S1_S2* and *np(_)/friends/S2_[]* (level 4) and also, *vp1(finite,[np(3-sing)/john])/[Subj]/S1_S12* and *p/up/S12_S2*, at the level below, are visited, will be reversed. This happens because both literals in both pairs are "ready" for the expansion at the moment when the selection is to be made. Note that the traversal made by SHDGA and the first traversal taken by EAA actually generate the terminal nodes in the same order. This property is formally defined by STAS relation, previously introduced.

The traversals by SHDGA and EAA as marked on the graph are *same-to-a-subtree (STAS)*. This means that the order in which the terminals were produced (the leaves were visited) is the same (in this case: *calls up friends john*). As noted previously, EAA can make other traversals to produce the same output string and the order in which the terminals are generated will be different in each case. (This should not be confused with the order of the terminals in the output string, which is always the same). The orders in which terminals are generated during alternative EAA traversals are: *up calls friends john, friends calls up john, friends up calls john.* In general, EAA can make a traversal corresponding to any permutation of "ready" literals in the right-hand side of a rule.

### 10.1.2. Completeness-wise Superiority of EAA over SHDGA

#### *10.1.2.1. Completeness-wise Superiority of EAA over SHDGA in MSEA-lead Optimal Cases*

Here, we analyze EAA and SHDGA with respect to the completeness criterion, considering the cases when they both or only one of them find msea-lead optimal traversals of analysis trees. As we pointed out earlier, there are also optimal traversals which are not msea-lead, but those would have to contain some guessing. Since we assume that each algorithm has equal chances to find optimal traversals by doing wild guessing, we will not consider cases when an optimal traversal is found by correct random guesses.

EAA simply refuses to expand a rule, if there is no msea instantiated at the moment of the expansion and as we pointed out, if there is no msea-lead path in the original grammar, it produces an equivalent grammar for which such a path does exist (all at compile time).[3]

SHDGA does not possess the same property and allows guessing. Thus, theoretically it is possible that SHDGA guesses an optimal traversal, but there is nothing inherent to SHDGA that facilitates correct guesses. With SHDGA, a correct guess means a correct guess of a rule among the applicable ones. SHDGA does not specify how to choose a rule ("chain", or "non-chain") among applicable ones and it is usually implemented to select the topmost among them. As pointed out, these cases with guessing are not of interest here. They assume pure randomness and we are interested in the situations when finding an optimal traversal is driven by some inherent properties of an algorithm and not by random correct guesses. Since non-msea-lead optimal means some guessing and we saw that guessing is done only by SHDGA and in a completely random fashion, we suffice with comparing the algorithms with respect to the msea-lead optimal solutions that they do or do not find.

We should notice that in the example given in section 10.1.1. and illustrated by the figure 10.1.1.3., SHDGA happened to make all the right moves, i.e., it always expanded a literal whose msea happened to be instantiated. As we will see in the following sections, this will not always be the case for SHDGA and will become a source of serious efficiency problems. On the other hand, whenever SHDGA indeed follows an optimal msea-lead traversal, EAA will have a traversal that is same-to-a-subtree with it. This can be summarized by the next theorem.

[3]We should point out that there are msea-lead traversals which are not EAA msea-lead traversals. In fact, any msea-lead traversal that first visits a child node, then after a while its parent node, and then after a while another child node, violates the rule of EAA traversals that either first visit all children nodes, and then the parent node, or first the parent node, and after that all children nodes.

**Theorem 10.1.2.1.1.**: Let T be an ODT and S set of all initially uninstantiated variables in the root logic grammar symbol. If there is a SHDGA traversal of T such that at each particular step only vertices whose at least one minimal set of essential arguments is disjunct with their UnI component are visited, then there is an EAA traversal same-to-a-subtree (STAS) with the SHDGA traversal∎

In other words, if the SHDGA, at each particular step during its implicit traversal of the analysis tree, visits only the vertices representing literals that have at least one of their sets of essential arguments instantiated at the moment of the visit, then the traversal taken by the SHDGA is the same-to-a-subtree (STAS) as one of the traversals taken by EAA.

*Proof.*

For this proof we will use induction by the number N of nodes in the tree.

Base Case: $N = 1$.

If the tree has only one node, then the claim of the theorem trivially holds.

Inductive Hypothesis (IH) : $N \leq K$ ($K \geq 1$).

Given set S of all initially uninstantiated variables in the root vertex, if the tree T has K or less nodes and SHDGA, at each particular step during its implicit traversal of the analysis tree, visits only the vertices representing literals that have at least one of their sets of essential arguments instantiated at the moment of the visit, then the traversal taken by the SHDGA is the same-to-a-subtree (STAS) as one of the traversals taken by EAA.

Case N = K+1.

There are two possible situations: either (i) root node r is visited first (a rule having the root logic grammar symbol on its left-hand side, with all and only variables from S being uninstantiated, is expanded first) or (ii) some other node $n_i$ somewhere in a subtree $T_i$ rooted at an immediate descendent f the root r is first visited node during the SHDGA traversal (again, for the corresponding logic grammar symbol uninstantiated are all variables from S, plus all other variables taking part in the corresponding rule having the $n_i$ logic grammar symbol on its left-hand side, that are not instantiated for this rule). Figures 10.1.2.1.1. and 10.1.2.1.2. visualize these situations.

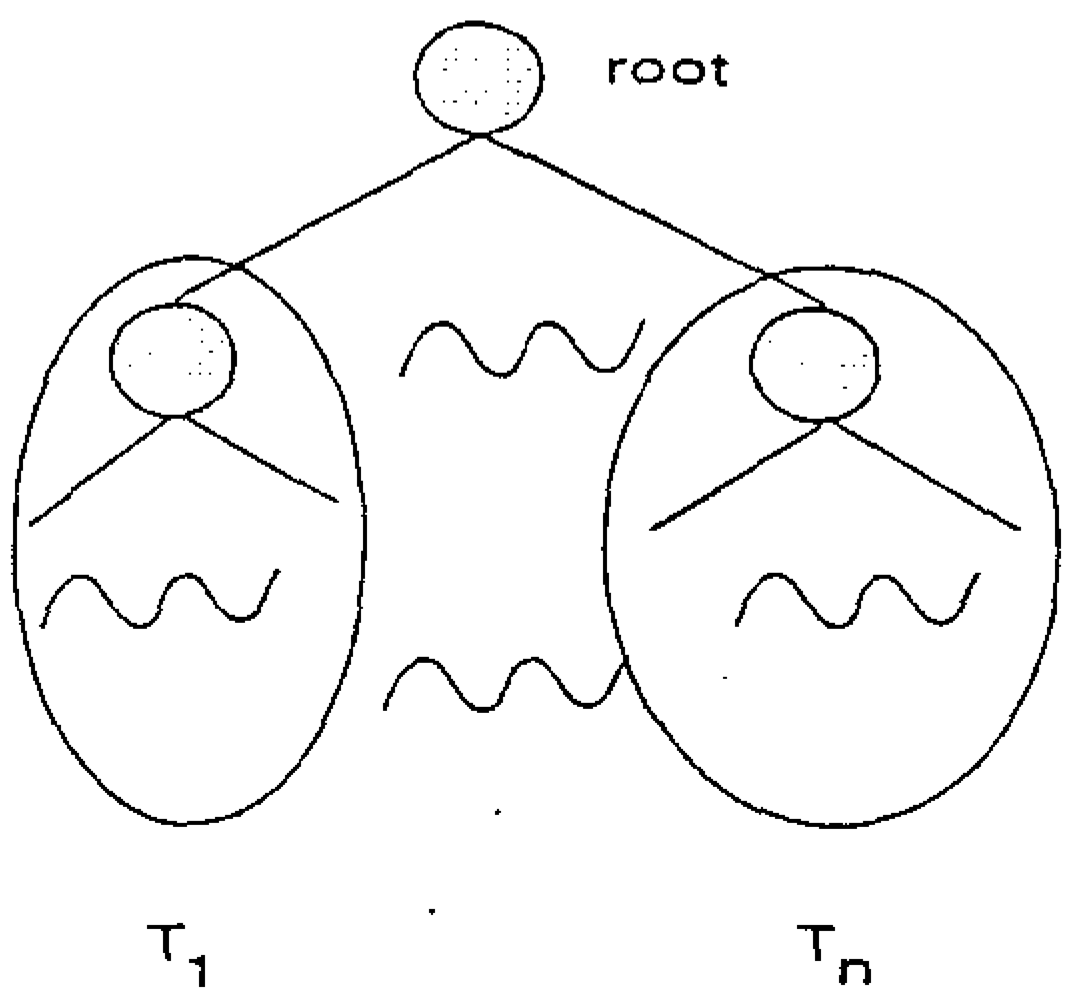


**Figure 10.1.2.1.1.: Case (i) from the Theorem 10.1.2.1.**

If (i) is the case, then say, $T_1,...,T_n$ are all subtrees rooted at an immediate descendent of r. Say SHDGA chooses the permutation $T_i^1,...,T_i^n$ of trees $T_1,...,T_n$ as the order in which they are traversed. Since each of $T_i^1,...,T_i^n$ are trees with less than K nodes, by (IH), there are corresponding SHDGA and EAA (${}_{SHDGA}T_i^1,...,{}_{SHDGA}T_i^n$ and ${}_{EAA}T_i^1,...,{}_{EAA}T_i^n$, respectively) traversals for them that are STAS (${}_{SHDGA}T_i^1$ with ${}_{EAA}T_i^1$,...,${}_{SHDGA}T_i^n$ with ${}_{EAA}T_i^n$). The sought STAS EAA traversal of T ${}_{EAA}T$, will then be traversal that will have the same instantiation components for its nodes as in ${}_{EAA}T_i^1$,...,${}_{EAA}T_i^n$. Also, for r, the in-instantiation status is the initial one and out-instantiation status is the final one. Let number of nodes in $T_i^1,\dots,T_i^n$, respectively be $t_i^1,\dots,t_i^n$. Let $n_i^j$ be a node from $T_i^j$ and let its ordinal numbers from ${}_{EAA}T_i^j$ be ${}_{EAA}o_i^j$. Then, $n_i^j$'s ordinal components in ${}_{EAA}T$ will be number $t_i^1+\dots+t_i^{j-1}+{}_{EAA}o_i^j+1$. It is obvious that such ${}_{SHDGA}T$ and ${}_{EAA}T$ are then STAS.

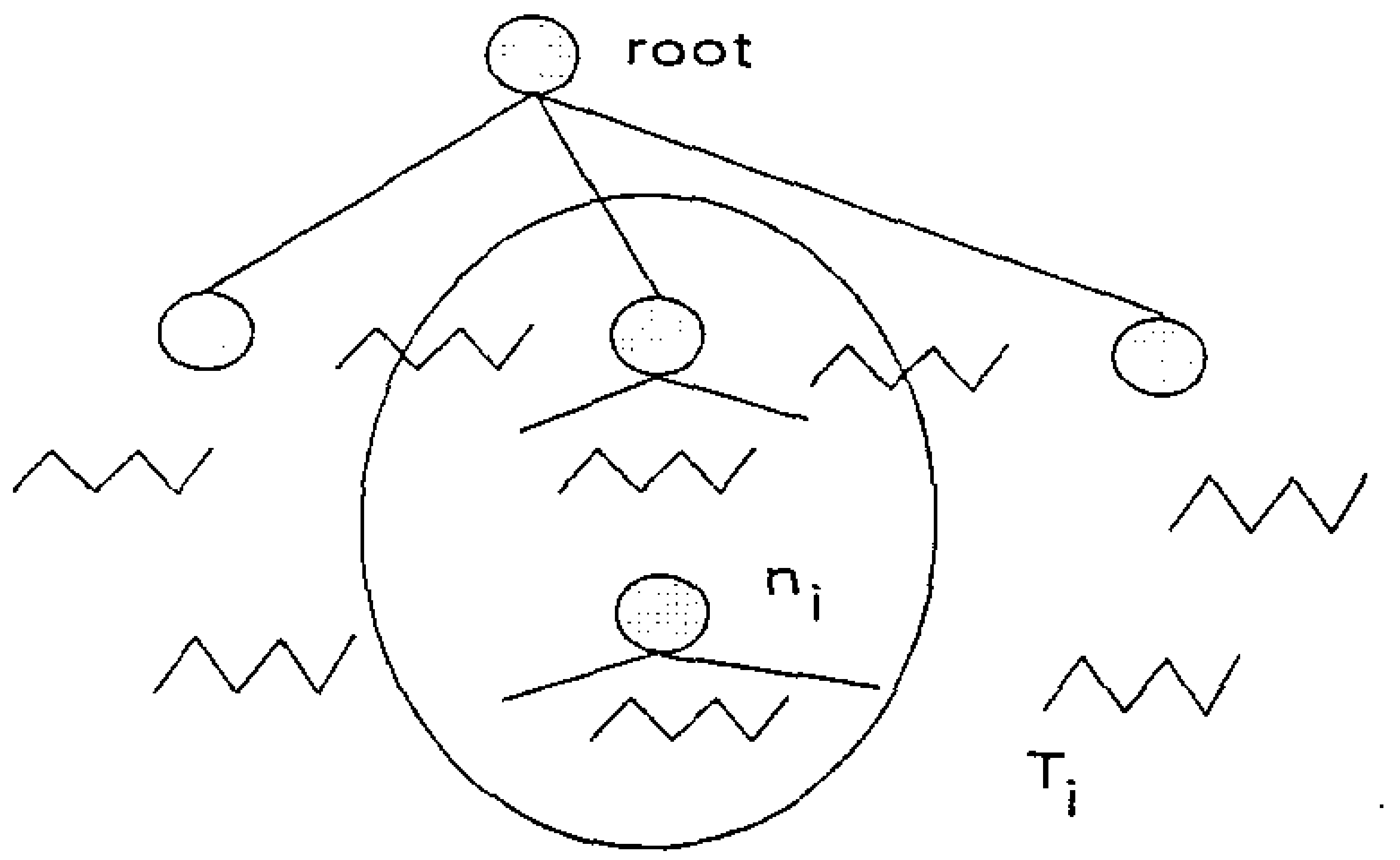


**Figure 10.1.2.1.2.: Case (ii) from the Theorem 10.1.2.1.**

However, if (ii) was the case, then say, SHDGA chooses some other node $n_i$ other than r to be visited first during the traversal. Let $T_i$ be the subtree of the tree T rooted at $n_i$. Since $T_i$ has less than or equal to K nodes, (IH) can be applied to it and therefore, there are two traversals ${}_{SHDGA}T_i$ and ${}_{EAA}T_i$ (SHDGA and EAA respectively) that are STAS. Given the initial in-instantiation status for their variables, they finish with same out-instantiation status $\sigma$. Let T' be the initial tree T after $T_i$ was removed from it, together with the edges coming into $n_i$ and let $\sigma$ be a given instantiation status for variables in T'. Now, we can apply (IH) to T', given $\sigma$, because number of nodes in T' is certainly less than or equal to K. Thus, for T', there are two STAS traversals ${}_{SHDGA}T'$ and ${}_{EAA}T'$, produced by SHDGA and EAA, respectively. Let ${}_{EAA}T$ be the following EAA traversal: For nodes from $T_i$, all set components remain as they were in ${}_{EAA}T_i$. For nodes from T', only ordinal numbers change comparing to what they were in ${}_{EAA}T'$. If the number of nodes in $T_i$ is $t_i$, then new ordinal numbers for the nodes from T', will be increased by $t_i$ each. Obviously, such EAA traversal is STAS with ${}_{SHDGA}T$. This completes the proof of the theorem∎

The following simple extract from a grammar, defining a *wh-question*, illustrates a case when both algorithms find msea-lead optimal traversals (in fact EAA finds two) and SHDGA's and EAA's traversals are STAS.

```
whques/WhSem -->          whsubj(Num)/WhSubj,
                          whpred(Num,Tense,[WhSubj,WhObj])/WhSem, whobj/WhObj.      (1)
..........
whsubj(_)/who -->                              [who].                              (2)
whsubj(_)/what -->                             [what].                             (3)
..........
whpred(sing,perf,[Subj,Obj])/wrote(Subj,Obj) -->  [wrote].                         (4)
..........
whobj/this -->                                 [this].                             (5)
..........
```

The input semantics for this example is *wrote(who,this)* and the output string *who wrote this*. The numbering for the edges taken by the SHDGA algorithm is given in italics and for the EAA in roman case. Both algorithms expand the middle subtree first, then the left and finally the right one. Each of the three subtrees has only one path, therefore the choices of their subtrees are unique and therefore both algorithms agree on that, too. However, the way they actually traverse these subtrees is different. For example, the middle subtree is traversed bottom-up by SHDGA and top-down by EAA. *whpred* is expanded first by SHDGA (because it shares the semantics with the root and there is an applicable non-chain rule) and also by EAA (because it is the only literal on the right-hand side of the rule (1) that has one of its msea's instantiated (its semantics)).

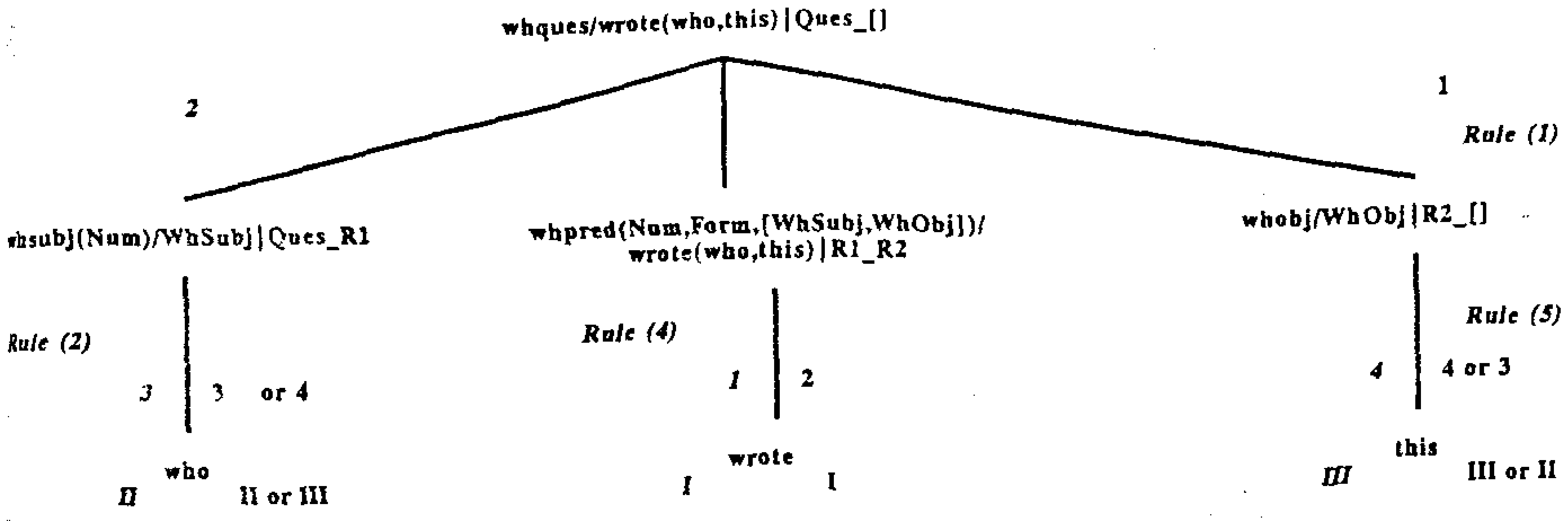

whques/wrote(who,this)|Ques_[]
2
1
Rule (1)
whsubj(Num)/WhSubj|Ques_R1
whpred(Num,Form,[WhSubj,WhObj])/ wrote(who,this)|R1_R2
whobj/WhObj|R2_[]
Rule (2)
3
3 or 4
who
II
II or III
Rule (4)
1
2
wrote
I
I
Rule (5)
4
4 or 3
this
III
III or II

Figure 10.1.2.1.3.: SHDGA's and EAA's Traversals of *WHQUES* Derivation Tree.

After the middle subtree is completely expanded, both sibling literals for the whpred have their semantics instantiated and thus they are both ready for expansion. We must note that SHDGA will always select the leftmost literal (in this case, *whsubj*), whether it is ready or not. EAA can select the same one, but it can also expand *whobj* first and then *whsubj*, since they are both *ready*. In the first solution by EAA, the terminals are generated in the order *wrote who this*, while in the second one the order is *wrote this who*. The first traversal of EAA and the only one of SHDGA are same-to-a-subtree.

The example from the section on efficiency comparison of the two algorithms presents an example where

EAA finds an optimal traversal and SHDGA does not and runs into serious inefficiency. Together with the previous theorem that justifies the following claim:

**Theorem 10.1.2.1.2.**: Let G be an unification-based grammar and ${}^{MSEA}O_{SHDGA}$ and ${}^{MSEA}O_{EAA}$ sets of all optimal msea-lead traversals of derivation trees for corresponding language constructs from G, found by SHDGA and EAA, respectively. Then, ${}^{MSEA}O_{SHDGA}$ is a proper subset of ${}^{MSEA}O_{EAA}$ (${}^{MSEA}O_{SHDGA} \subset {}^{MSEA}O_{EAA}$)∎

A proof of theorem 10.1.2.1.2. is given by the example from the section on efficiency comparison (section 10.1.3.) that presents an example where EAA finds an optimal msea-lead solution and SHDGA does not and by the claim of the theorem 10.1.2.1.1..

Our concern now naturally shifts to the non-optimal situations for both algorithms. We show next that whenever a finite derivation exists (and therefore a finite derivation tree), EAA is capable of discovering it. Not only that, although that solution might not be optimal with respect to the initial grammar, it will be optimal with respect to an equivalent grammar, created from the original one by the use of *lift up transformations*. This procedure for creation of a new grammar by adding new rules will naturally enlarge the size of the grammar. In section on reversibility comparison of two algorithms we point out that EAA is multi-directional algorithm that can be used for parsing as well as for generation, as opposed to SHDGA that can be used only for generation tasks. Thus, there are traversals of analysis trees corresponding to the process of parsing that SHDGA can not discover. Therefore, we show that not only there exists an optimal msea-lead traversal discovered by EAA and not discovered by SHDGA, then also there are traversals discovered by EAA (in an optimal or non-optimal fashion) that SHDGA will not find at all.

*10.1.2.2. Completeness-wise Superiority of EAA over SHDGA in Non-Optimal Cases*

In practice however, we very often deal with imperfect semantic representations that actually loose some information that is available at the surface string level. Thus, when we attempt to generate from such a semantic representation, guessing may be necessary to allow several solutions from which the right one(s) will be selected when the context in which they are used become known. Therefore, in EAA implementation, definition of msea's is relaxed and certain level of indeterminacy is allowed at the lexical level.[4] As a consequence, we have, as we show next, absolute completeness and a "reasonable" minimality in backtracking instead of pure determinacy and optimality. It is the price to pay for dealing with imperfect representations. However, this compromise also makes situations in which the EAA created grammar becomes extensively large (because of numerous cases in which ICR is used in order to introduce new rules) highly improbable and rare.

The following theorem establishes EAA's capability of discovering any finite derivation tree for a pair of corresponding language constructs (a surface string and a corresponding semantics), provided that such a tree exists.

**Theorem 10.1.2.2.1.**: For any finite FODT T and any given instantiation status of variables representing arguments in node predicates given by the set $IS_{IN}$ of all initially uninstantiated variables participating in the derivation, there exists an EAA traversal of the tree∎

*Proof*

We use transfinite induction with respect to the number of nodes in the tree. Let number of nodes in the

[4]Instead of insisting on "at most one" choice for binding of participating variables in the lexical predicates, this is relaxed to "at most n" in specific applications. The choice of n depends on the nature and the degree of ambiguity of the input language and the cost of backtracking introduced by this, because of the possibility of unsuccessful expansions.

tree be denoted by N.

Case N=1:

One node tree corresponds to derivation that is an use of a rule with no right-hand side. For such a literal anything is a set of essential arguments and therefore an EAA traversal trivially exists.

Case N≤K (K≥1):

Inductive Hypothesis (IH): For any finite FODT with K or less nodes and any given instantiation status of variables representing arguments in node predicates, there exists an EAA traversal of the tree.

Case N=K+1:

There are two possible situations: (a) there is a child of the root of the tree that is *ready* (it has one of its msea's instantiated, or (b) no child of the root is *ready*.

Case (a):

Let $N_i$ denote the *ready* node and $S_i$ the subtree rooted at $N_i$. Tree $S_i$ has $n_i$ nodes and that number is less than or equal to K and with $IS_{IN}$ as its initial instantiation status is subject to (IH). The application of (IH) gives us an EAA traversal EAA($S_i$,$IS_{IN}$) of $S_i$ , leaving at the end $IS_{IN'}$ as instantiation status of the variables taking part in this derivation. After extracting $S_i$ from the initial tree T, we get another tree T'. We can now apply (IH) to T' with its initial instantiation status $IS_{IN'}$, because T' has less than or equal to K nodes. Therefore there exists an EAA traversal EAA(T',$IS_{IN'}$) of T'. In the sought EAA traversal of the entire tree T, all components of the nodes remain the same as in EAA($S_i$,$IS_{IN}$) and EAA(T',$IS_{IN'}$), except that the ordinal numbers for the nodes from T' are increased by $n_i$, the number of nodes in $S_i$. Figure 10.1.2.2.1. should be helpful in visualizing this part of the proof.

Case (b):

No child of the root can be a leaf node, because then it is a *ready* node. Let $N_i$ be the most instantiated node among the root's children, $S_i$ the subtree rooted at $N_i$, the number of nodes in $S_i$ some n (of course $\leq K$) and $N_{i,1},\ldots,N_{i,m}$ all of $N_i$'s children.

Let T' be a tree obtained from the initial tree T where $N_i$ was removed and instead of it, nodes $N_{i,1},\ldots,N_{i,m}$ are lifted up to the $N_i$'s place, together with the respective subtrees $S_{i,1},\ldots,S_{i,m}$ rooted at those nodes. Also edges from the root to each of $N_{i,1},\ldots,N_{i,m}$ are added. The new tree T' has exactly K nodes and therefore we can apply the inductive hypothesis to it. By (IH) there exists an EAA traversal EAA(T',$IS_{IN}$) for T' and given instantiation status $IS_{IN}$. The EAA traversal for the initial tree T (EAA(T,$IS_{IN}$)) can be constructed from EAA(T',$IS_{IN}$) in the following manner: Let $V_j$ be the last visited node among the nodes from the subtrees $N_{i,1},\ldots,N_{i,m}$ and its ordinal number is ord($V_j$). All nodes whose ordinal numbers from EAA(T',$IS_{IN}$) are less than or equal to ord($V_j$) will have the same components in EAA(T,$IS_{IN}$), as they had in EAA(T',$IS_{IN}$). UnI and UnO components of $N_i$ will be the same as UnO component of $V_j$ and ordinal number for $N_i$ will be ordinal number for $V_j$ increased by one. All other nodes in T will have same components in EAA(T,$IS_{IN}$) as they had in EAA(T',$IS_{IN}$), except that their ordinal numbers will be increased by one∎

The previous theorem proves that whenever there is a finite derivation, EAA is capable of finding and traversing the corresponding analysis tree. Thus, there cannot exist an algorithm that discovers a derivation tree that EAA cannot discover and therefore the same is true for SHDGA.

Figure 10.1.2.2.2. and 10.1.2.2.3. should be helpful in visualizing how trees T and T', respectively look like.

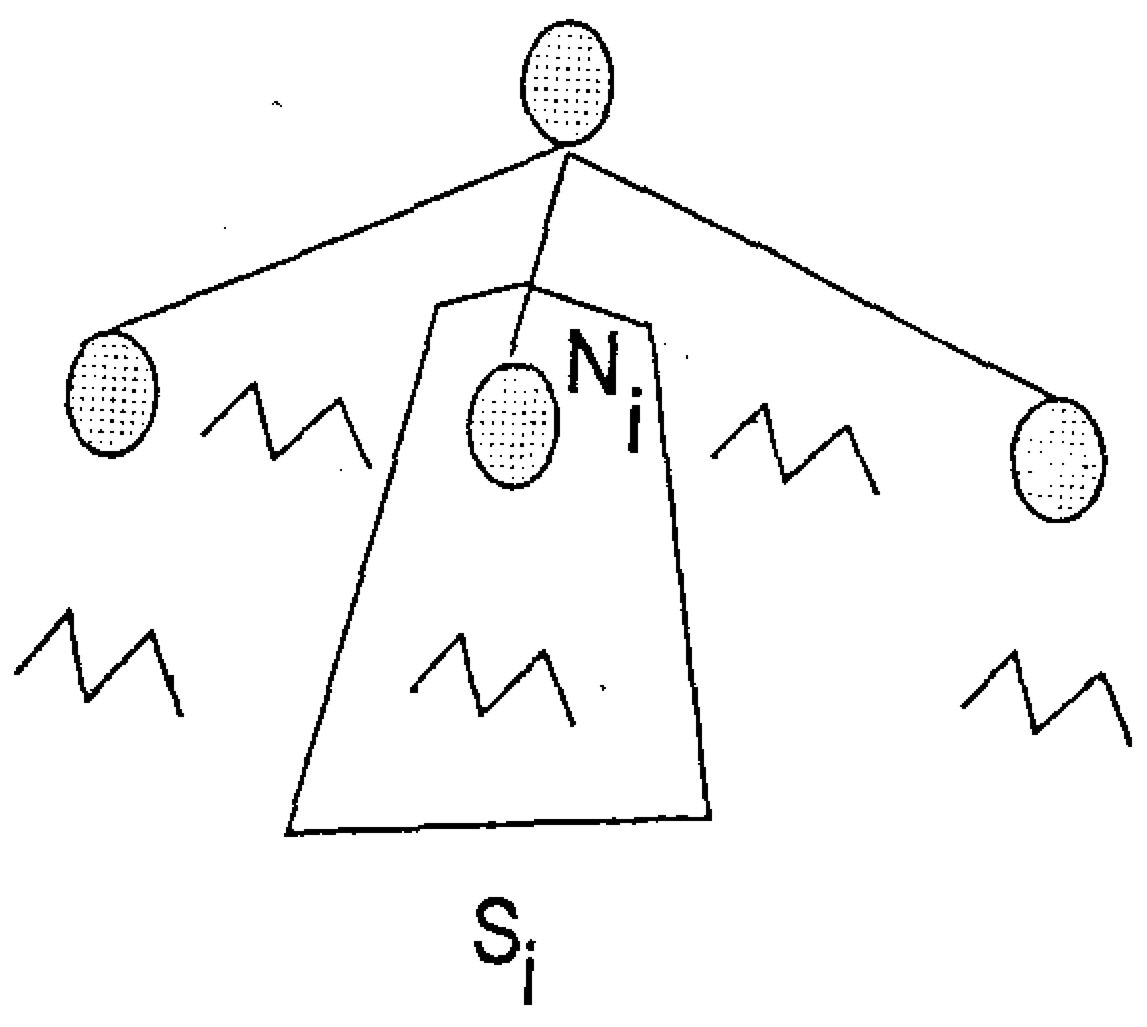


Figure 10.1.2.2.1.: Existence of an EAA Traversal (Case (a)).

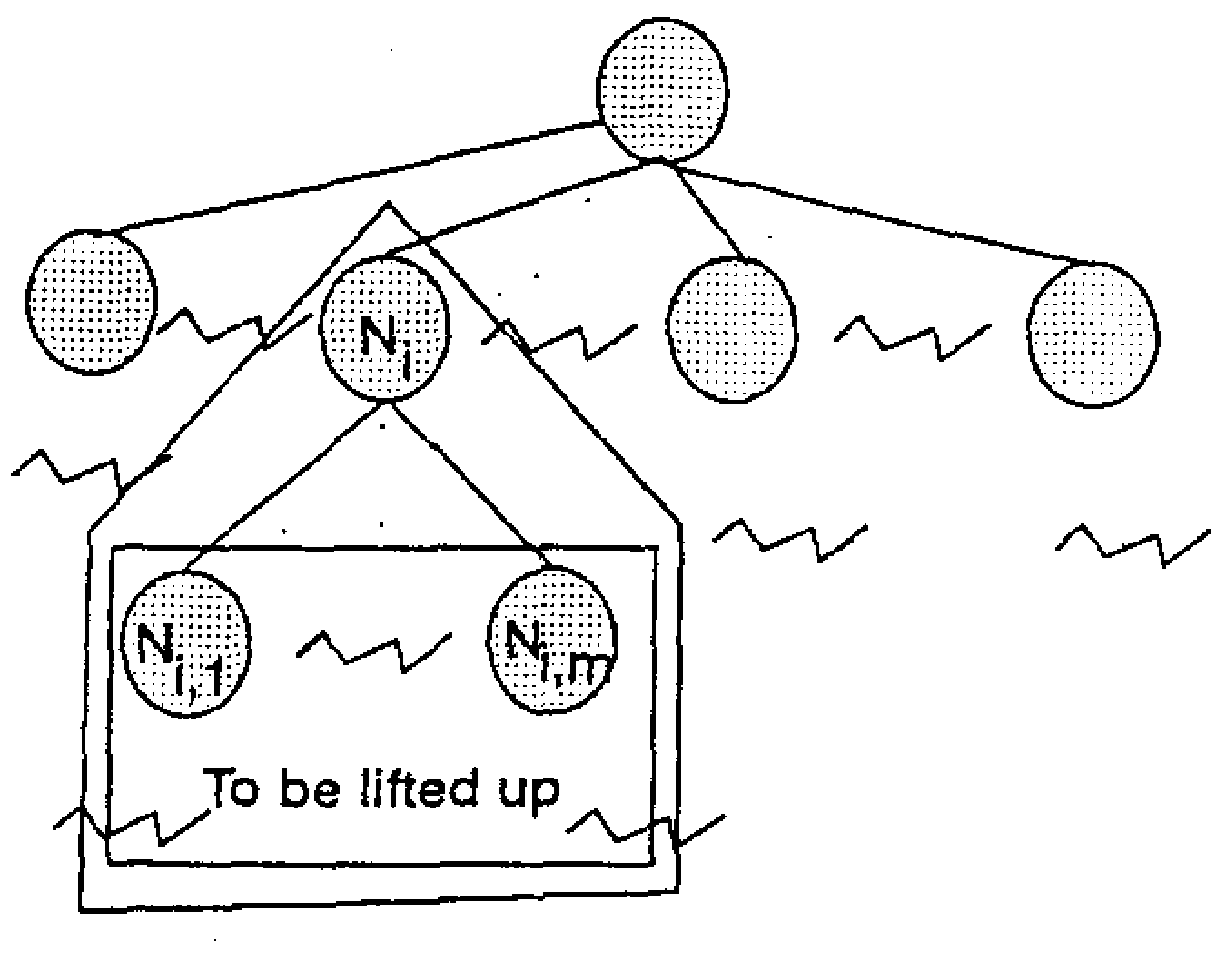


Figure 10.1.2.2.2.: Existence of an EAA Traversal - Tree T (Case (b)).

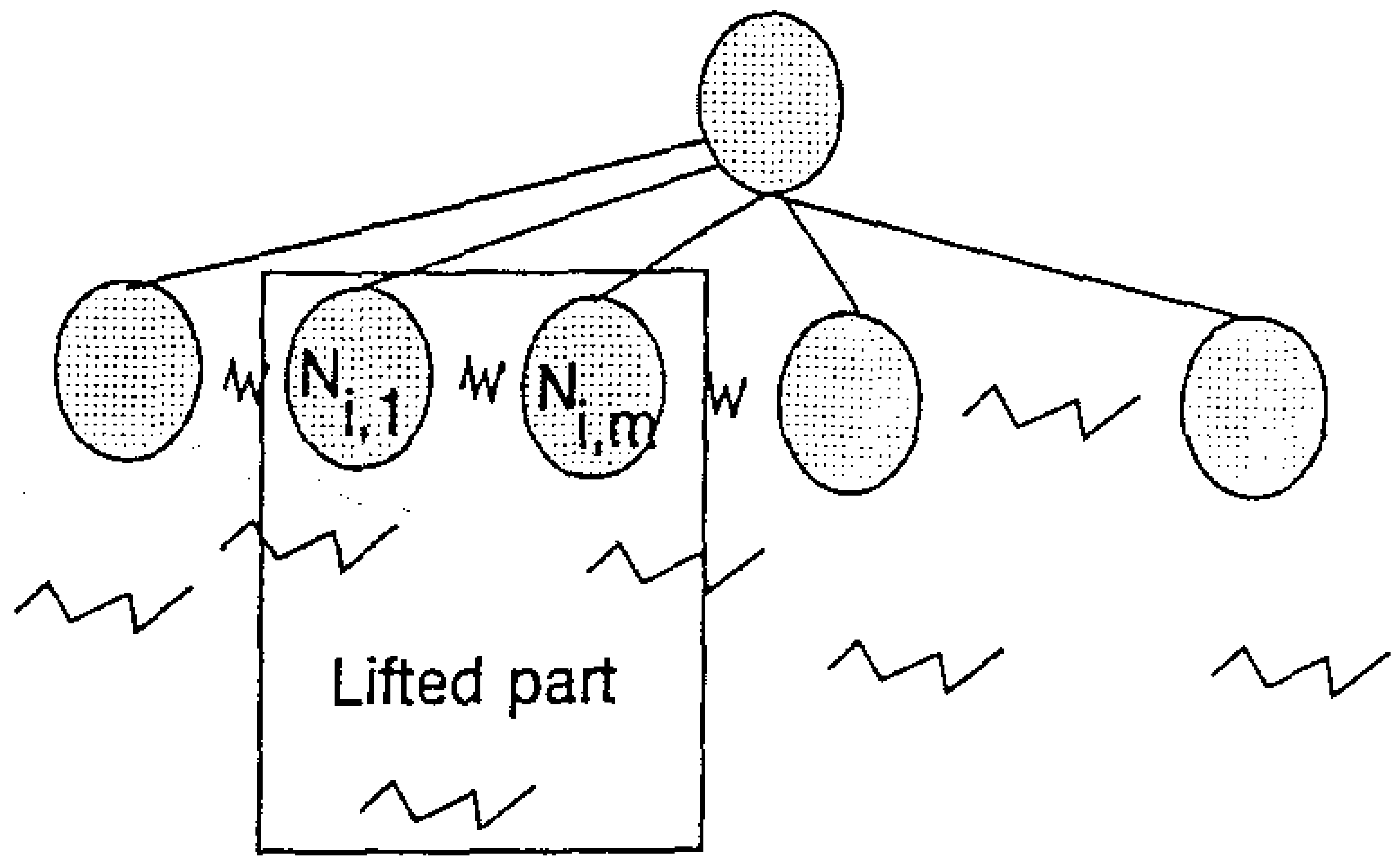


**Figure 10.1.2.2.3.: Existence of an EAA Traversal - Tree T' (Case (b)).**

**Corollary 10.1.2.2.1.**: Whenever SHDGA discovers a derivation tree, EAA will discover it, too

Putting together the corollary 10.1.2.2.1. and the facts elaborated in the section 10.1.4. on reversibility that EAA is capable of handling parsing tasks as well as generation and SHDGA can only deal with generation and its corresponding traversals, we can draw the following conclusion.

**Theorem 10.1.2.2.2.**: Let G be an unification-based grammar and $T_{SHDGA}$ and $T_{EAA}$ set of all derivation trees discovered by SHDGA and EAA, respectively. Then $T_{SHDGA}$ is a proper subset of $T_{EAA}$ ($T_{SHDGA} \subset T_{EAA}$)

The reason for EAA being more complete than SHDGA in both, optimal and non-optimal cases is in the fact that EAA takes into consideration global picture for a given grammar by employing the ICR technique. SHDGA on the other hand makes only local decisions, with respect to one production rule. Without ICR, EAA traversal of an analysis tree would be a top-down (TD) a-full-subtree-always (AFSA) traversal. Therefore, being an AFSA traversal, once EAA traversal (without ICR) commits to a subtree, it does not move to another subtree before all nodes in the current one are visited. SHDGA has the same property.

The actual EAA (with inter-clausal reordering as its integral part) is neither TD, nor AFSA traversal any more. It can move from one subtree to another, back and forth when that is needed. This makes EAA traversals more general than the ones taken by SHDGA. As it will be shown later in this paper, there are sometimes efficiency advantages of having ICR available, too.

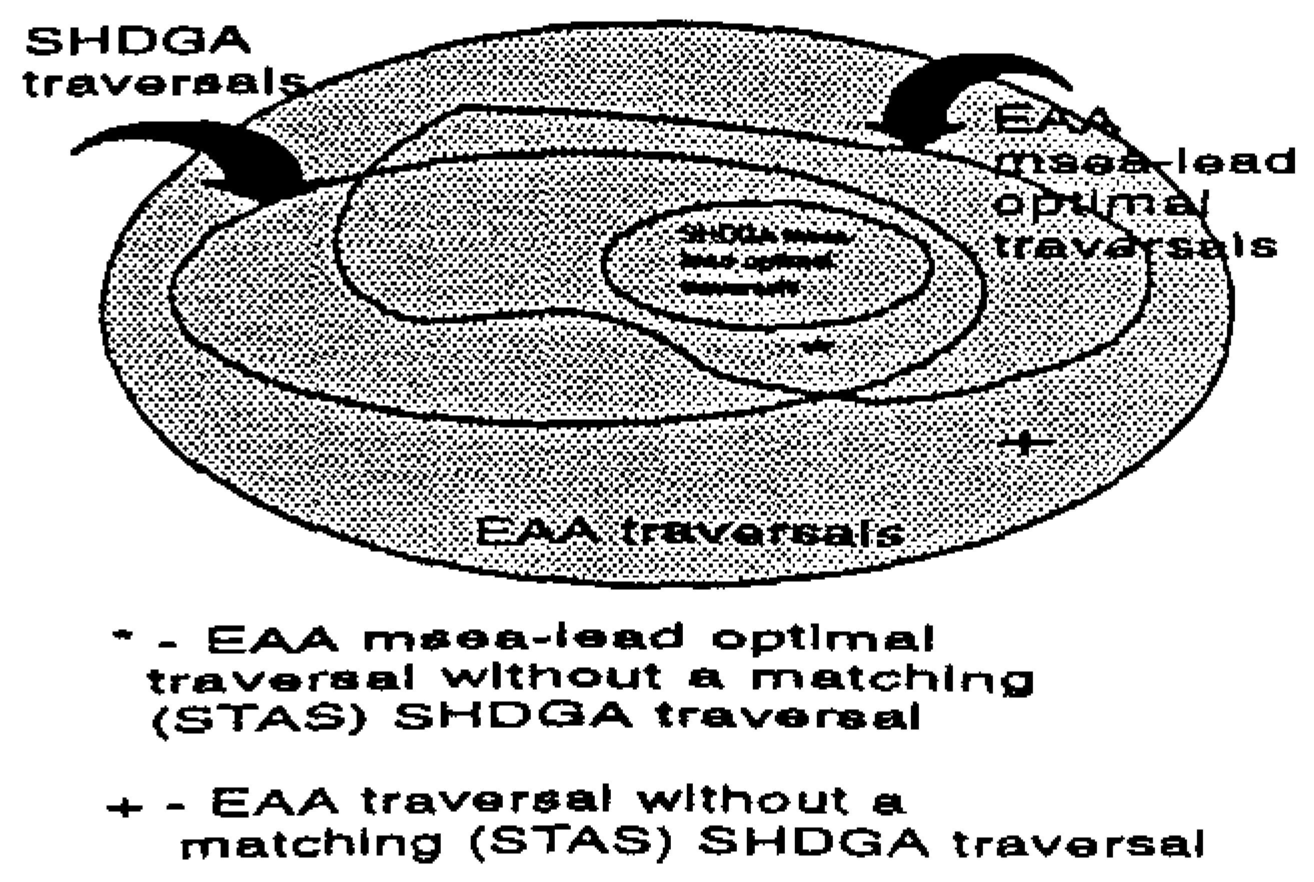

SHDGA traversals
EAA masea-lead optimal traversals
EAA traversals
+
*
* - EAA masea-lead optimal traversal without a matching (STAS) SHDGA traversal
+ - EAA traversal without a matching (STAS) SHDGA traversal

**Figure 10.1.2.2.4.: Completeness-wise Relation between SHDGA and EAA.**

The figure 10.1.2.2.4. summarizes the completeness-wise relation between EAA and SHDGA in optimal and non-optimal cases.

The situations in which an optimal solution cannot be found by EAA usually were caused by either left-recursive rules, or treatment of potentially unbound subcategorization lists, or deadlock-prone rules. Left recursion can be handled through a normalization process or ICR at compile-time. The problems with subcategorization lists and deadlock-prone rules are also handled by applying *lifting up transformations*, because the main reason for difficulties with them by EAA without ICR is that these are the situations when

TD evaluation is proven to be inferior. By including ICR, EAA got the capability of not being TD if necessary and more than that, not even being AFSA, which resulted in expanding the applicability of the algorithm, as well as improving its performance efficiency-wise in non-optimal situations.

### 10.1.3. Efficiency-wise Superiority of EAA over SHDGA

The following example is a simplified fragment of a parser-oriented grammar for *yes or no questions*. Using this fragment we will illustrate some deficiencies of SHDGA. It presents a situation in which EAA finds an optimal solution while SHDGA does not and it ends up in an extreme inefficiency.

```
..........
sentence/ques(askif(S)) --> yesnoq/askif(S).                                (1)
..........
yesnoq/askif(S) -->                                                         (2)
        aux_verb(Num,Pers,Form)/Aux,
        subj(Num,Pers)/Subj,
        main_verb(Form,[Subj,Obj])/Verb,
        obj(_,_)/Obj,
        adj([Verb])/S.
..........
aux_verb(sing,one,pres_perf)/have(pres_perf,sing-1) --> [have].             (3)
..........
aux_verb(sing,one,pres_cont)/be(pres_cont,sing-1) --> [am].                 (4)
..........
aux_verb(sing,one,pres)/do(pres,sing-1) --> [do].                           (5)
```

```
..........

aux_verb(sing,two,pres)/do(pres,sing-2) --> [do].                     (6)

..........

aux_verb(sing,three,pres)/do(pres,sing-3) --> [does].                 (7)

..........

aux_verb(pl,one,pres)/do(pres,pl-1) --> [do].                         (8)

..........

subj(Num,Pers)/Subj --> np(Num,Pers,su)/Subj.                         (9)

..........

obj(Num,Pers)/Obj --> np(Num,Pers,ob)/Obj.                            (10)

..........

np(Num,Pers,Case)/NP --> noun(Num,Pers,Case)/NP.                      (11)
np(Num,Pers,Case)/NP --> pnoun(Num,Pers,Case)/NP.                     (12)

..........

pnoun(sing,two,su)/you --> [you].                                     (13)
pnoun(sing,three,ob)/him --> [him].                                   (14)

..........

main_verb(pres,[Subj,Obj])/see(Subj,Obj) --> [see].                   (15)
main_verb(pres_perf,[Subj,Obj])/seen(Subj,Obj) --> [seen].            (15a)
main_verb(perf,[Subj,Obj])/saw(Subj,Obj) --> [saw].                   (15b)

..........

adj([Verb])/often(Verb) --> [often].                                  (16)

..........
```

The analysis tree for the input semantics *ques ( askif ( often ( see ( you,him ) ) ) )* (the output string being *do you see him often*) is given on figure 10.1.3.1..

Both algorithms start with the rule (1). SHDGA selects (1) because it has the left-hand side nonterminal with the same semantics as the root and it is a non-chain rule. EAA selects (1) because its left-hand side unifies with the initial query (-?- *sentence (OutString_[]) / ques(askif(often(see(you,him))))* ).

Next, rule (2) is selected by both algorithms. Again, by SHDGA, because it has the left-hand side nonterminal with the same semantics as the current root (*yesnoq/askif...*) and it is a non-chain rule; and by EAA, because the *yesnoq/askif...* is the only nonterminal on the right-hand side of the previously chosen rule and it has an instantiated msea (its semantics). The crucial difference takes place when the right-hand side of rule (2) is processed. EAA deterministically selects *adj* for expansion, because it is the only rhs literal with an instantiated msea's. As a result of expanding *adj*, the *main_verb* semantics becomes instantiated and therefore *main_verb* is the next literal selected for expansion. After processing of *main_verb* is completed, *Subject*, *Object* and *Tense* variables are instantiated, so that both *subj* and *obj* become ready. Also, the tense argument for *aux_verb* is instantiated (*Form* in rule (2)). After *subj* and *obj* are expanded (in any order), *Num* and *Pers* for *aux_verb* are bound and finally *aux_verb* is ready, too.

In contrast, the SHDGA does not specify how to choose among the rhs literals from a "non-chain" rule. Thus, an arbitrary selection is made and in its most frequent implementations SHDGA will proceed by selecting the leftmost literal (*aux_verb(Num,Pers,Form)/Aux*) of the rule (2). At this moment, none of its arguments is instantiated and any attempt to unify with an auxiliary verb in a lexicon will succeed. Suppose then that *have* is returned and unified with *aux_verb* with *pres_perf* as *Tense* and *sing_1* as *Number*. This restricts further choices of *subj* and *main_verb*. However, *obj* will still be completely randomly chosen and then *adj* will reject all previous choices. The decision for rejecting them will come when the literal *adj* is expanded, because its semantics is *often(see(you,him))* as inherited from *yesnoq*, but it does not match the previous choices for *aux_verb*, *subj*, *main_verb* and *obj*. Thus we are forced to backtrack repeatedly and it may be a while before the correct choices are made.

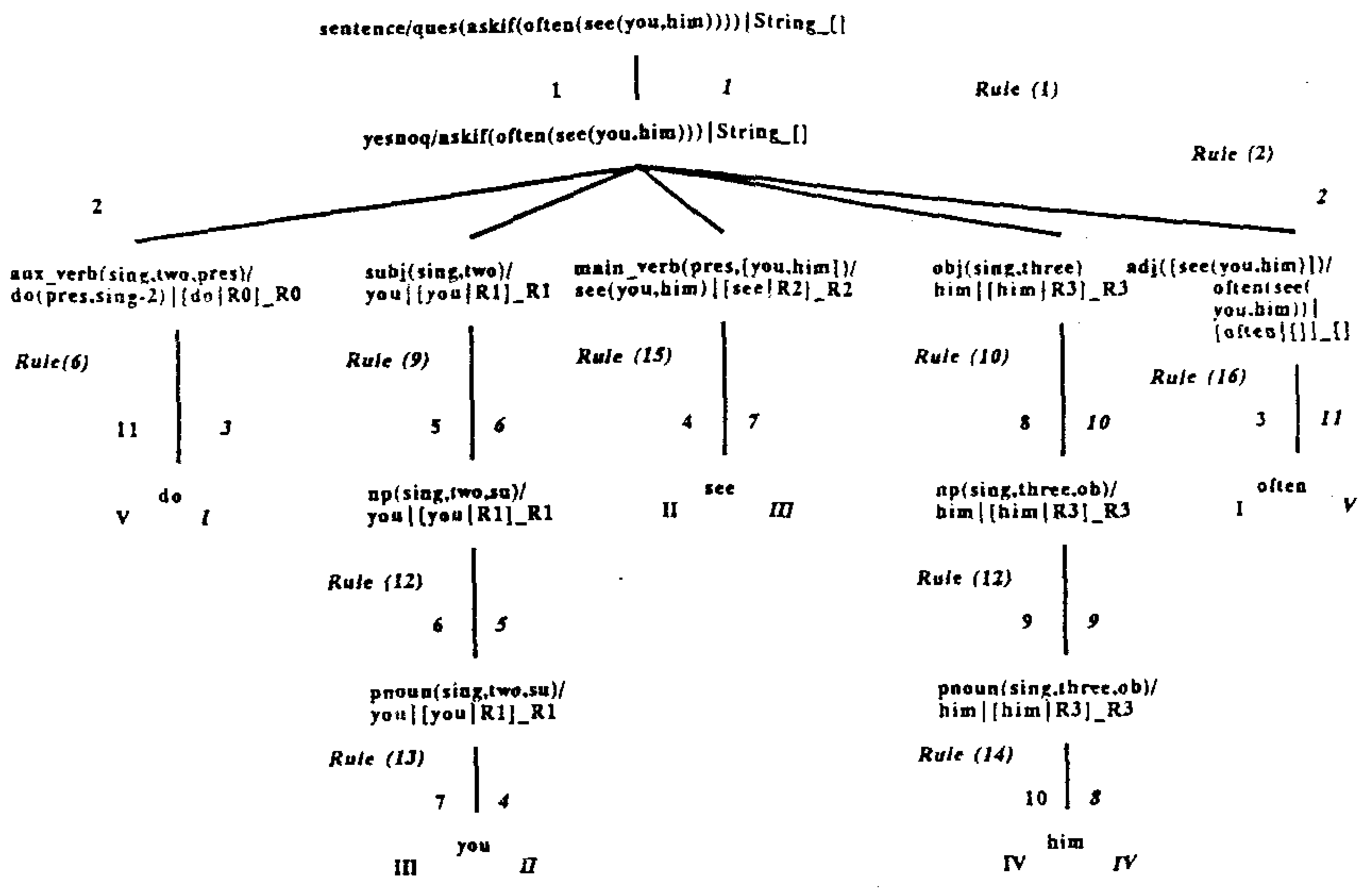


Figure 10.1.3.1.:SHDGA's and EAA's Traversals of *YESNOQ* Derivation Tree.

In fact the same problem will occur whenever SHDGA selects a rule for expansion such that its leftmost right-hand side literal (first to be processed) is not "ready". Since SHDGA does not check for "readiness" before expanding a predicate, other examples similar to the one discussed above can be found easily. We may also point out that the fragment used in the previous example is extracted from an actual computer grammar for English (Sager's Linguistic String Grammar) and therefore, it is not an artificial problem.

In fact, in general case if there are $n$ preterminals $v_1,...,v_n$, at which SHDGA have $p_1,...,p_n$ options for making non-deterministic choices among respectively, its average case analysis number (ACAN), and worst case analysis number (WCAN) will be $(p_1*...*p_n)/2^n$ and $p_1*...*p_n$, respectively. Best case analysis number (BCAN) would be $n$. With EAA however, all of them will be $n$, provided an optimal ordering of LG symbols exists.

The only way to avoid such problems with SHDGA would be to rewrite the underlying grammar, so that the choice of the most instantiated literal on the right-hand side of a rule is forced. This could be done by changing rule (2) in the example above into several rules which use meta nonterminals *Aux*, *Subj*, *Main_Verb* and *Obj* in place of literals *aux_verb*, *subj*, *main_verb* and *obj* respectively, as shown below:

```
..........
yesnoq/askif(S) ---> askif/S.
askif/S --->
          Aux, Subj, Main_Verb, Obj, adj([Verb],[Aux,Subj,Main_Verb,Obj])/S.
..........
```

Since *Aux*, *Subj*, *Main_Verb* and *Obj* are uninstantiated variables, we are forced to go directly to *adj* first. After adj is expanded the nonterminals to the left of it will become properly instantiated for expansion, so in effect their expansion has been delayed. However, this solution seems to put additional burden on the grammar writer, who need not be aware of the evaluation strategy to be used for a grammar.

We present another example illustrating a typical situation in which none of the algorithms finds an optimal traversal, but EAA by applying inter-clausal reordering (*lift up transformation*) shows again a better performance. Let us consider the following extract from a grammar taken from [S90b].

```
sent/P -->                np(Num,Pers)/P1, vp(Num,Pers,P1)/P.                 (1)
vp(Num,Pers,P1)/P -->     verb(Num,Pers)/P2, compl/P3, combine(P1,P2,P3,P).    (2)
```

Arguments *P1* and *P* carry the semantics' of *np* and *sent*, respectively. Suppose that *P1* is the only essential argument in *np* and that *Num* and *Pers* get fully instantiated whenever *P1* is instantiated previous to the expansion of a rule with *np* on its left-hand side. In *vp*, essential arguments are *P, Num, Pers* and when they are instantiated previous to an expansion, *P1*, will be instantiated after it. Obviously, this clause cannot be optimally inverted for generation, since in order to expand *vp* we need to know the bindings to *Num* and *Pers*, in addition to *P* (which is passed from *sent*). These we could get by expanding *np*, but we cannot do this either, since we need to know the bindings to *P1*, which is unavailable until *vp* is executed. This constitutes a deadlock. The information about the arguments can be summarized as follows:

| **Predicate** | **MSEA** | **Instantiated after the Expansion** |
|---|---|---|
| sent | {P} | - |
| np | {P1} | Num,Pers |
| vp | {P,Num,Pers} | P1 |
| verb | {P2,Num,Pers} | - |
| compl | {P3} | - |
| combine | {P} | P1,P2,P3 |

As we can see, the critical information required to expand *np* in the first clause, that is, the binding to *P1*, can be obtained only after a partial evaluation of *vp* and before the bindings for *Num* and *Pers* become indispensable. Therefore, even though {P,Num,Pers} create a msea for *vp*, the bindings to the last two are not used until the literal *verb(Num,Pers)/P2* is about to be evaluated. Indeed, we can combine the two clauses into one (*lift up transformation* on the tree from figure 10.1.3.2.) by expanding *vp* in (1) by appropriately instantiated right-hand side of (2). After that EAA can easily achieve the desired ordering of goals by creating a new rule in the following manner: *sent/P --> combine(P1,P2,P3,P), np(Num,Pers)/P1, verb(Num,Pers)/P2, compl/P3*. With respect to the initial derivation tree, traversals for EAA and SHDGA look like on figure 10.1.3.2.. The input semantics is assumed to be *chase(you,john)* and the output string *you chase john*. EAA will succeed in reordering, by lifting up *verb, compl* and *combine* into the level of *np* and then sorting them for visits as follows: *combine, np, verb, compl* (denoted in arabic numbers).

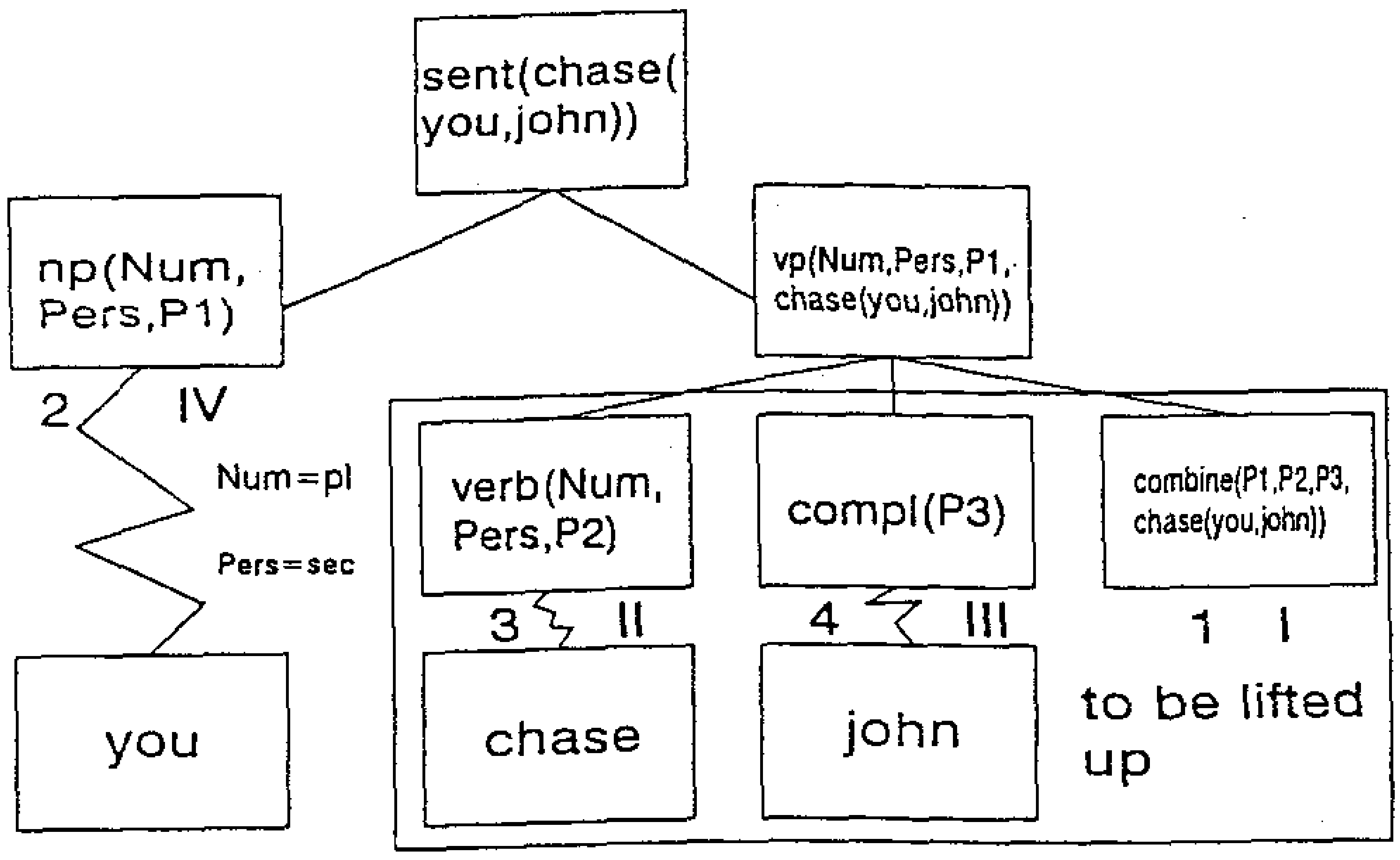


**Figure 10.1.3.2.:** ***Lift Up Transformation*** **and Efficiency Considerations.**

All choices are here fully deterministic. SHDGA traverses a tree in an *AFSA* fashion and therefore once committed to the subtree rooted at *vp*, it will not jump to another subtree before all nodes here are visited. That produces the following ordering of nodes: *combine, verb, compl, np* (it is denoted in roman numbers). That will cause the choice for *Num* and *Pers* to be made non-deterministically (probably first like *singular* and *first*, rejected when *np* gets its turn, then like *singular* and *second*, again rejected later on and so on until the correct guesses *plural* and *second* are finally made).

Thus, compile-time overhead of EAA is expressed here by the time needed for the *lift up transformation* (here, very minor because the first try succeeded). In fact this would result in creation of a new grammar rule and the time needed for that is the price to pay for obtaining an optimal traversal with respect to the new grammar. At run-time, there will be no backtracking and inefficiency by EAA's traversal. SHDGA will repeatedly backtrack at run-time until the correct guesses are made.

Summarizing all efficiency considerations we can conclude that when SHDGA finds an optimal msea-lead traversal, EAA will do the same. EAA will find optimal msea-lead traversals even when SHDGA cannot and when they both fail to find an optimal path with respect to the original grammar, the overhead is expected from both algorithms. However, in the case of EAA it is a compile-time one and a matter of the efficiency of the compiler, while for SHDGA it is a more costly, run-time overhead. Making non-deterministic choices and extensive backtracking are expected for SHDGA and *lift up* type of compile-time overhead for EAA. EAA finds optimal msea-lead traversals for the newly created grammar that is equivalent to the original one, but the time to create it affects the efficiency of its compiler.

Let us also mention that both algorithms handle left recursion satisfactorily. SHDGA processes recursive chain rules in a constrained bottom-up fashion and this also includes deadlock prone rules. EAA can either get rid of left recursive rules during the grammar normalization process, or it can handle them during its ICR phase.

### 10.1.4. Multi-directionality of EAA and Generation-oriented SHDGA

Another property of EAA regarded as superior over the SHDGA is its multi-directionality. EAA can be used for parsing as well as for generation and therefore it is a reversible algorithm. The algorithm will simply recognize that the top-level msea is now the string and will adjust to the new situation. Moreover, EAA can be run in any direction "paved" by the predicates' mseas as they become instantiated at the time a rule is taken up for expansion.

The following simple example illustrates the multi-directionality of EAA:

```
empty_list_first_or_second ([], A, yes_1).
empty_list_first_or_second ([A|B], C, no_1).
empty_list_first_or_second (A, [], yes_2).
empty_list_first_or_second (A, [B|C], no_2).
```

Obviously, the predicate *empty_list_first_or_second(X,Y,Z)* will have three different msea's: {X}, {Y} and {Z}. Instantiation of any of them would suffice for a finite expansion (even a successful one). EAA algorithm, if applied to this predicate would simply recognize which of the msea's is instantiated and proceed along a simple msea-lead path. Thus, EAA can be run here in three different directions, depending only on the instantiation status of the arguments of *empty_list_first_or_second(X,Y,Z)* predicate.

In contrast, SHDGA can only be guaranteed to work in one direction, given any particular grammar, although the same architecture can apparently be used for both generation ([SNMP90]) and parsing ([K90], [N89]). Thus, given an initial task where semantics is not instantiated, or the grammar is written in a manner where semantic head is not explicitly indicated, SHDGA is not applicable. Its deficiencies here are consequences of two facts. First is that, as opposed to EAA that in order to be executable needs only cryptic text of an unification-based grammar, SHDGA needs explicit information what the semantic head is (given by the symbol "/"). Second one is that its behavior when semantic head is not completely instantiated, is not even defined and therefore the algorithm would not be usable in the context of parsing. Even if, as some authors suggested, instead of semantic heads, we try to extend its applicability and use syntactic heads and proceed with specifying (hopefully with success) the algorithm's behavior for parsing in an analogous way, explicit information about heads would still be crucial. Symmetry of parsing and generation would be much less obvious than with EAA, where these are only two instances of one process whose only differences come from different initial instantiation statuses for the participating variables.

The point is that some grammars (as shown in the example above) need to be rewritten for parsing or generation, or else they must be constructed in such a way so as to avoid indeterminacy. While it is possible to rewrite grammars in a form appropriate for head-first computation, there are real grammars which will not evaluate efficiently with SHDGA, even though EAA can handle such grammars with no problems.

# 11. CONCLUSION

Natural language is an integral part of our lives serving as the primary vehicle by which people communicate and record information. It has the potential for expressing an enormous range of ideas, and for conveying complex thoughts succinctly. The aim of computational linguists is, in a sense, to capture this power. They have, over the years, adopted many different grammar formalisms for use by their parsers and generators, sometimes borrowing them from linguistics, sometimes developing them from scratch for the purpose in hand. Even today, there are probably two dozen or more distinct grammar formalisms in use in various NLP projects across the world. It is a recognized fact that there is a big gap between linguistic theories and practical, natural language applications. What is lacking most are first, constructive links between different formalisms, and then bridges between theories and practical applications, as well as evaluation systems for different practical language processing algorithms.

This study attempts to address two of the mentioned lacking issues and make a contribution to solving them.

This thesis builds a constructive connection between the traditional formalisms of Turing machines and type 0 phrase structure grammars, and the most current ones, unification based grammars (namely, DCG's). A procedure for directly rewriting any Turing machine program into an equivalent DCG, as well as another procedure for rewriting a type 0 phrase structure grammar into a DCG is described here. Taking into consideration the amount of results obtained within the traditional formalisms and the wide acceptance of unification based grammars as a formalism of today, this result provides a means for restating the results obtained within the old formalisms directly in the more contemporary ones. Since most Prolog implementations make DCG formalism directly available, we have a practical convenience of being able to run and subject to a "real life" test every result from the theory of Turing machines and phrase structure grammars. It is envisaged that these procedures will be automated in the future and that their computer implementations are an interesting continuation of the work done in this thesis. Two compilers (Turing machines to DCG's, and type 0 phrase structure grammars to DCG's) based on the results from Chapter 3 here would be a final outcome of such a task.

Another problem that we paid a special attention to in this study is building of an evaluation system for grammar processing algorithms (parsers and generators). In order to approach the problem formally, we had to start with a formalization of some basic concepts that were previously used by computational linguists mostly in an informal way. That lead us to building a formal foundation of the theory of logic grammars. This was intended to provide a basis for a formdir al introduction to the formalism of unification based grammars, similar to the one that exists for phrase structure grammars. Also, the definition of analysis tree traversals laid foundation for the later evaluation system that was developed in this work. Criteria of generality, completeness, soundness, reversibility and finiteness of grammar processing algorithms are formally defined, including different approaches that exist to these notions. We also defined a relation (STAS) on the set of all tree traversals that, as we demonstrated, can be very conveniently used as a tool for the evaluation of the completeness and reversibility criteria. It is an equivalence relation and it can be used for the classification of the algorithms, based on how the algorithms traverse their derivation trees. Simple metrics is provided for judging the efficiency of the algorithms. It is based on the number of edges tried during a derivation process and it covers the worst, average and best case analysis. Chapter 9 on the finiteness introduces the notion of universal guides based on the sets of unbound variables that appear within logic grammar symbols during a process of derivation. This notion is proved to be more general in several aspects than the notion of proper guides introduced in some previous work on the finiteness. It is applicable to any of the grammar processing algorithms unlike the proper guides and it captures the symmetry of the parsing and generation process much better than proper guides. This result, I believe, open a whole new area for exploring finiteness of grammar processing algorithms. It is my belief that many new results will be soon reported using universal guides as a tool. Finally, two grammar based generation algorithms are compared in Chapter 10 (semantic-head-driven generation algorithm and essential arguments algorithm). The basis for their comparison is the way they traverse analysis trees, and notions and techniques from the previous chapters were employed. Essential arguments algorithm proved to be superior in almost any aspect in any of the cases that are considered.

I strongly believe that the contributions from the thesis will even more encourage and enhance the research on the issues that are topics here, and that, in a way, they suggest a way forward for the future work in the area.

# 12. ACKNOWLEDGEMENTS

I would like to thank to Mathematical Faculty, Faculties of Sciences and Mathematics, from Belgrade University for supporting me during this research.

Part of the thesis was also based upon work supported by the Defense Advanced Research Project Agency under Contract N00014-90-J-1851 from the Office of Naval Research, the National Science Foundation under Grant IRI-89-02304 and the Canadian Institute for Robotics and Intelligent Systems (IRIS).

I also thank my advisor Dr. Tomek Strzalkowski for his inspiring guidance.

At the end, I would like to thank to the numerous friends and colleagues of mine who were able to put up with me during the course of the thesis work. They encouraged and supported me with their patience and tolerance.

## 13. REFERENCES

[AN76] ANDREKA, H. and NEMETI I., 1976.

"The Generalized Completeness of Horn Predicate Logic as a Programming Language".

D.A.I. Research Report No. 21, - 1976.

[A87] APPELT, D., 1987.

"Bidirectional Grammars and the Design of Natural Language Generation Systems".

In *TINLAP-3*, New Mexico State University.

Las Cruces, N.M., pp. 206-212.

[BBR87] BARTON G.E., BERWICK R.C. and RISTAD E.S., 1987.

*Computational Complexity and Natural Language*.

A Bradford Book, The M.I.T. Press.

[B78] BATES, M., 1978.

"The Theory and Practice of Augmented Transition Networks".

In *Natural Language Communication with Computers*, Edited by L. Bolc.

Lecture Notes in Computer Science, 63.

Springer-Verlag, New York, NY.

[B91] BLOCK, H.U., 1991.

"Two Optimizations for Semantic-Head-Driven Generators".

In *Proceedings of the Third European Workshop on Natural Language Generation*

Judenstein/Innsbruck, Austria.

[C78] COLMERAUER, A., 1978.

"Metamorphosis Grammars".

In *Natural Language Communication with Computers*, Edited by L. Bolc.

Lecture Notes in Computer Science, 63.

Springer-Verlag, New York, NY, pp. 133-189.

[DA89] DAHL, V., ABRAMSON, H., 1989.

*Logic Grammars.*

Springer Verlag New York Inc.

[DA84] DAHL, V. and ABRAMSON, H., 1984.

"On Gapping Grammars".

*Proceedings of the Second International Conference on Logic Programming*

Uppsala, Sweden, pp. 77-88.

[DW83] DAVIS, M.D. and WEYUKER, E.J., 1983.

*Computability, Complexity and Languages.*

Academic Press, Inc.

[D89] DEBRAY, S. K., 1989.

"Static Inference Modes and Data Dependencies in Logic Programs".

ACM Transactions on Programming Languages and Systems 11(3), pp. 418-450.

[D83] DIJKSTRA, E. W., 1983.

"Program Inversion".

Springer-Verlag New York Inc., pp. 351-354.

[D90a] DYMETMAN, M., 1990.

"A Generalized Greibach Normal Form for DCG's".

CCRIT, Laval, Quebec: Ministere des Communications Canada

[D90b] DYMETMAN, M., 1990.

"Left-Recursion Elimination, Guiding and Bidirectionality in Lexical Grammars".

To Appear.

[DI88] DYMETMAN, M. and ISABELLE, P., 1988.

"Reversible Logic Grammars for Machine Translation".

*Proceedings of the 2nd International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages*

Carnegie-Mellon University, Pittsburgh, PA.

[DIP90] DYMETMAN, M., ISABELLE, P. and PERRAULT, F., 1991.

"A Symmetrical Approach to Parsing and Generation".

*Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*

Helsinki, Finland, Vol. 3., pp. 90-96.

[GM89] GAZDAR, G. and MELLISH, C., 1989.

*Natural Language Processing in Prolog.*

Addison-Wesley, Reading, MA.

[G86a] GRISHMAN, R., 1986.

*Computational Linguistics.*

Studies in Natural Language Processing, Cambridge University Press.

[G86b] GRISHMAN, R., 1986.

"Proteus Parser Reference Manual".

Proteus Project Memorandum #4,

Courant Institute of Mathematical Sciences, New York University, N.Y.

[HU79] HOPCROFT, J.E. and ULLMAN, J.D., 1979.

*Introduction to Automata Theory, Languages and Computation.*

Addison-Wesley Publishing Company, Reading, Massachusetts.

[K90] KAY, M., 1990.

"Head-Driven Parsing".

In M. Tomita (ed.), *Current Issues in Parsing Technology*

Kluwer Academic Publishers, Dordrecht, the Netherlands.

[K84] KAY, M., 1984.

"Functional Unification Grammar: A Formalism for Machine Translation".

*Proceedings of the 10th International Conference on Computational Linguistics (COLING-84)*

Stanford University, Stanford, CA., pp. 75-78.

[KSB85] KLUZNIAK, F., SZPAKOWICZ, S. and BIEN, J.S., 1985.

*Prolog for Programmers.*

Academic Press, Inc.

[L91] LANDMAN, F., 1991.

*Structures for Semantics.*

Studies in Linguistics and Philosophy, Kluwer Academic Publishers.

[M90] MARTINOVIĆ, M., 1990.

*Definite Clause Grammars and Natural Language Processing in Prolog.*

Master Thesis in Computer Science,

May 1990, Pace University, New York, New York.

[MS92] MARTINOVIĆ, M. and STRZALKOWSKI, T., 1992.

"Comparing Two Grammar-Based Generation Algorithms: A Case Study".

*Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics,*

July 1992, Newark, Delaware.

[M56] MCCARTHY, J., 1956.

"The Inversion of Functions Defined by Turing Machines".

In *Automata Studies*, Ed. by C.E.Shannon, J. McCarthy

Princeton University Press, Princeton, N.J.

[MWWS87] MCCORD, M., WALKER, A., WILSON, W.G. and SOWA, J.F., 1987.

*Natural Language Processing in Prolog.*

Library of Congress Cataloging-in-Publication Data.

[N85] NAISH, L., 1985.

"Automating Control for Logic Programs".

*Journal of Logic Programming* 3, pp. 167-183.

[N89] VAN NOORD, G., 1989.

"An Overview of Head-Driven Bottom-Up Generation".

*Proceedings of the Second European Workshop on Natural Language Generation*, Edinburgh, Scotland.

[PMW90] PARTEE, B.H., TER MEULEN, A. and WALL, R.E., 1990.

*Mathematical Methods in Linguistics.*

Kluwer Academic Publishers, Dordrecht/Boston/London.

[PS90] PENG, P. and STRZALKOWSKI, T., 1990.

"An Implementation of A Reversible Grammar".

*Proceedings of the 8th Conference of the Canadian Society for the Computational Studies of Intelligence (CSCSI-90)*

University of Ottawa, Ottawa, Ontario, pp. 121-127.

[PS87] PEREIRA, F.C.N. and SHIEBER S.M., 1987.

*Prolog and Natural-Language Analysis.*

Center for the Study of Language and Information, Stanford, Ca.

[PW80] PEREIRA, F.C.N. and WARREN D.H.D., 1980.

"Definite Clause Grammars for Language Analysis".

*Artificial Intelligence* 13, pp. 231-278.

[R82] ROBINSON, J., 1982.

"DIAGRAM: A Grammar for Dialogues".

*Communications of the ACM* 25 (1), pp. 27-47.

[S81] SAGER, N., 1981.

*Natural Language Information Processing.*

Addison-Wesley, Reading, MA.

[SSW91] edited by SELLS, P., SHIEBER, S.M. and WASOW, T., 1991.

*Foundational Issues in Natural Language Processing.*

A Bradford Book, The M.I.T. Press.

[SNMP89] SHIEBER, S.M., VAN NOORD, G., MOORE, R.C. and PEREIRA, F.C.N., 1989.

"A Semantic-Head-Driven Generation Algorithm for Unification-Based Formalisms".

*Proceedings of the 27th Meeting of the ACL*

Vancouver, B.C., pp. 7-17.

[SNMP90] SHIEBER, S.M., VAN NOORD, G., MOORE, R.C. and PEREIRA, F.C.N., 1990.

"Semantic-Head-Driven Generation".

*Computational Linguistics*, Volume 16, Number 1.

[SM84] SHOHAM, Y. and MCDERMOTT, D.V., 1984.

"Directed Relations and Inversion of Prolog Programs".

*Proceedings of the International Conference of*

*Fifth Generation Computer Systems,*

Institute for New Generation Computer Technology, Tokyo, Japan, pp. 307-316.

[SC82] SIMMONS, R.F. and CHESTER, D., 1982.

"Relating Sentences and Semantic Networks with Procedural Logic".

*Communications of the ACM* 25 (8), pp. 527-547.

[SC79] SIMMONS, R.F. and CORREIRA, A., 1979.

"Rule Forms for Verse, Sentences and Story Trees".

In *Associative Networks*, Ed. by N.V. Findler, Academic Press, New York, N.Y., pp. 363-392.

[SS86] STERLING, L. and SHAPIRO, E., 1986.

*The Art of Prolog - Advanced Programming Techniques.*

M.I.T. Press Series in Logic Programming.

[S90a] STRZALKOWSKI, T., 1990.

"How to Invert A Natural Language Parser into An Efficient

Generator: An Algorithm for Logic Grammars".

*Proceedings of the 13th International Conference on Computational Linguistics (COLING-90)*

Helsinki, Finland, Vol. 2., pp. 90-96.

[S90b] STRZALKOWSKI, T., 1990.

"Reversible Logic Grammars for Natural Language Parsing and Generation".

*Computational Intelligence Journal*, Vol. 6., pp. 145-171.

[S91] STRZALKOWSKI, T., 1991.

"A General Computational Method for Grammar Inversion".

*Proceedings of a Workshop Sponsored by the Special Interest Groups on*

*Generation and Parsing of the ACL*

Berkeley, CA., pp. 91-99.

[T87] TOMITA, M., 1987.

"An Efficient Augmented-Context-Free Parsing Algorithm".

*Computational Linguistics*, Volume 13, Numbers 1-2.

[W80] WARREN, D.H.D., 1980.

"Logic Programming and Compiler Writing".

*Software - Practice and Experience*, Volume 10.

[W88] WEDEKIND, J., 1988.

"Generation as Structure Driven Derivation".

*Proceedings of the 12th International Conference on Computational Linguistics (COLING-88)*

Budapest, Hungary, pp. 732-737.