Computational Pitfalls in Tractable Grammatical Formalisms

Marten H. Trautwein

Computational Pitfalls in Tractable Grammatical Formalisms

ILLC Dissertation Series 1995-15

institute *for* logic, language *and* computation

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-5256090
fax: +31-20-5255101
e-mail: illc@fwi.uva.nl

Computational Pitfalls in Tractable Grammatical Formalisms

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof.dr P.W.M. de Meijer
ten overstaan van een door het college van dekanen ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op donderdag 21 september 1995 te 10.00 uur

door

Marten Henrik Trautwein

geboren te Amsterdam.

Promotor: dr P. van Emde Boas
Co-promotores: dr T.M.V. Janssen
dr L. Torenvliet
Faculteit der Wiskunde en Informatica
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam

The investigations were supported by the Foundation for Language, Speech and Logic (TSL), which is subsidized by the Netherlands Organization for Scientific Research (NWO).

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Trautwein, Marten Henrik

Computational pitfalls in tractable grammatical formalisms
/ Marten Henrik Trautwein. - Amsterdam :Institute for
Logic, Language and Computation. - Ill. - (ILLC
dissertation series ; 1995-15)
Proefschrift Universiteit van Amsterdam. - Met index, lit.
opg.
ISBN 90–74795–34–x
NUGI 855
Trefw.: complexiteitstheorie / computerlinguïstiek.

Cover: Tarot ∼ The Fool.
Printed and bound by CopyPrint 2000, Enschede.

Contents

Acknowledgments 9

1 Overview 11

1.1 Motivation for this Research . . . 11

1.1.1 Motivation for our Approach . . . 12

1.1.2 Assessing Complexity Results . . . 13

1.2 An Outline of this Thesis . . . 14

2 Restricted Attribute-Value Grammars 17

2.1 Introduction . . . 17

2.2 Definitions and Notation . . . 18

2.2.1 Attribute-Value Grammars . . . 18

2.2.2 Restricted Attribute-Value Grammars . . . 20

2.3 Complexity of the Recognition Problem . . . 21

2.3.1 The Recognition Problem is NP-hard . . . 22

2.3.2 The Recognition Problem is NP-complete . . . 26

2.4 Weak Generative Capacity . . . 26

2.5 The Honest Parsability Constraint and Consequences . . . 28

2.6 Veer out the HPC . . . 29

2.7 Simulating a Context Free Grammar in GNF . . . 29

2.8 Constructing an Honestly Parsable Attribute-Value Grammar . . . 32

2.8.1 Arithmetic by AVGs . . . 32

2.8.2 Creating a counter of logarithmic size . . . 35

2.8.3 From AVG to HP-AVG . . . 37

3 Complexity of Categorial Unification Grammar 39

3.1 Introduction . . . 39

3.2 Preliminaries . . . 40

3.2.1 Complexity Theory . . . 40

3.2.2 The Standard Feature Theory . . . 42

3.3 Categorial Unification Grammar 51
3.3.1 The Grammatical Formalisms 51
3.3.2 Fixed Recognition is *NP*-hard 57
3.3.3 Universal Recognition is in *NP* 61
3.3.4 Weak Generative Capacity 63

4 Functional Unification Grammar **65**
4.1 Introduction in FUG 65
4.1.1 Comparison with the Standard Feature Theory 65
4.1.2 Comparison with CUG 68
4.2 The Simulation 69
4.3 Correctness of the Simulation 74
4.3.1 Definitions 74
4.3.2 Proof of Correctness 75
4.4 Formal Properties 78
4.4.1 Complexity of the Recognition Problems 78
4.4.2 Weak Generative Capacity 82

5 Head-driven Phrase Structure Grammar **85**
5.1 Introduction in HPSG 85
5.1.1 Comparison with the Standard Feature Theory 86
5.1.2 Comparison with CUG 89
5.2 The Simulation 91
5.3 Correctness of the Simulation 99
5.3.1 Definitions 99
5.3.2 Proof of Correctness 100
5.4 Formal Properties 103
5.4.1 Complexity of the Recognition Problems 103
5.4.2 Weak Generative Capacity 107

6 Lexical Functional Grammar **109**
6.1 Introduction in LFG 109
6.1.1 Comparison with the Standard Feature Theory 110
6.1.2 Comparison with CUG 112
6.2 The Simulation 115
6.3 Correctness of the Simulation 125
6.3.1 Notation and Definitions 125
6.3.2 Proof of Correctness 126
6.4 Formal Properties 129
6.4.1 Complexity of the Recognition Problems 129
6.4.2 Weak Generative Capacity 133

7 Complexity of Functional Grammar **137**
7.1 Introduction . . . 137
7.1.1 Functional Grammar . . . 137
7.1.2 Complexity Theory . . . 138
7.1.3 Outline of this chapter . . . 140
7.2 Fund Formation Process . . . 141
7.2.1 An Informal Introduction . . . 142
7.2.2 The Formalization . . . 145
7.2.3 The Fund Formation Problem is Undecidable . . . 149
7.3 Clause Structure Formation Process . . . 153
7.3.1 An Informal Introduction . . . 153
7.3.2 The Formalization . . . 155
7.3.3 The Complexity of the Clause Structure Formation Problem . 160
7.4 Form Specification Process . . . 163
7.4.1 An Informal Introduction . . . 163
7.4.2 The Formalization . . . 164
7.4.3 The Form Specification Problem is *NP*-complete . . . 168
7.5 Order Specification Process . . . 173
7.5.1 An Informal Introduction . . . 173
7.5.2 The Formalization . . . 174
7.5.3 The Order Specification Problem is *NP*-complete . . . 176
7.6 Conclusions and Further Research . . . 179

8 Conclusions **183**

Bibliography **185**

Index **190**

Samenvatting **195**

Acknowledgments

This thesis is the end-product of the NWO-project "Complexity Theory and Grammatical Theories," which was applied for by Theo Janssen and Leen Torenvliet. I would neither have started nor completed this thesis without the support in its widest sense of many people. First I want to thank my secular parents — Connie and Freek — and my scientific parents — Leen, Theo, and Peter — for the free education they gave me. I am grateful for the opportunities they provided and for accepting my choices. I hope that they are not disappointed that I did not follow in their footsteps.

I am indebted to the graduation committee — in particular Johan van Benthem, Machtelt Bolkestein, Michael Moortgat, and Bill Rounds — for their valuable criticism in reviewing this thesis. I also thank my "paranimfen" — Arie Mijnlieff and Paul Vriend — and my brother — Kees// — for spending their valuable time on proof-reading my work and their eagerness to help with anything. I appreciated their help and support during our mutual studies, and I missed their contributions in the past four years.

With joy I recall my visits at the Research Institute for Language and Speech in Utrecht and the Department of Computational Linguistics of the University of Amsterdam. I am grateful to everybody who made these stays a pleasant experience. Special thoughts go to Kees and Koen who proved extremely valuable in solving song fragments, and Tjoe Liong and Dik who illuminated the theory of Functional Grammar.

I wish to thank my colleagues at the third floor for the appreciation I received as coffee manager. Especially, I thank Martijn, Erik, Erik, Yuri, Ernest, Sophie, Harry, and all the others who joined me for lunch.

Most of all I am indebted to my closest friends — Marianne, Frank, Frans, Petra, Vincent, and Irma — who sometimes worried over my progress. I am also grateful to the friends who waited patiently for me when I was busy, but were ready for me when a songquiz had to be solved

Before I finish off let me go back to the early eighties, when Els and Wytse organized my brother's and mine initial meeting with a computer. In my view they are the creators of my interest in computers, which resulted in this thesis. I am very grateful they initiated us in the wonders of computers in general and their Apple II in particular. However, if it was not for Arne and Meike, we would never have found the power switch that conjured up all the miracles.

Amsterdam Marten Trautwein
July, 1995.

Chapter 1

Overview

In this dissertation we apply complexity theory to various grammatical formalisms. We presuppose that the reader has basic knowledge of the formal aspects of linguistics and the formal language theory. Familiarity with the rudiments of complexity theory is required to fully understand the material that that we will present. The main work in the interdisciplinary field of complexity theory and natural language is the book of Barton, Berwick and Ristad (1987). We encourage the reader who is unfamiliar with complexity theory to read Barton, Berwick and Ristad's (1987) Chapter 1. That first chapter contains an informal introduction into this interdisciplinary field and the rudiments of complexity theory. Readers familiar with complexity theory may be interested in reading Barton, Berwick and Ristad's (1987) more formal Chapter 2. That chapter discusses the contribution of complexity theory to natural language; advanced topics that pass the review are parallel processing and compilation.

1.1 Motivation for this Research

The merit of applying complexity theory to grammatical formalisms is that the complexity analyses provide different kinds of information. The complexity analyses do not only show *whether* a problem is difficult, but also *why* it is difficult and possibly even *how* to restrict the problem in order to make it less difficult. The effect of the complexity analyses are twofold. On the one hand, the analyses provide statements concerning the development of the theory of the grammatical formalisms. On the other hand, the analyses express general statements concerning implementations of these grammatical formalisms. In this dissertation the main emphasis lies on the development of theories.

The statements that result from the complexity analysis of a grammatical formalism are based on the following observations. The way in which humans process natural language suggests that the structure of a natural language enables *efficient* processing. That is, natural language utterances contain enough information to enable efficient processing. This efficient processing implies that determining whether a sequence of words forms a sentence is tractable. Hence accurate formalisms for natural language determine in an efficient way whether a sequence of words forms a sentence, i.e., the *recognition problems* of these formalisms are *tractable*. Thus almost by definition, the formalisms studied in this thesis are supposed to be *tractable grammatical formalisms*.

In this thesis we consider the complexity of various recognition problems. The complexity results that follow have to be interpreted with care and in an intelligent way. One should not brand a formalism as counterintuitive or unnatural solely because its recognition problem is intractable (cf., Rounds 1991). Instead, one should exploit the fact that the complexity result tells why the recognition problem is intractable. Thus complexity analyses provide insight into the structures that a grammatical formalism assigns to natural language utterances. An intractability result indicates that the structures that the grammatical formalism uses contain insufficient information. Moreover, the complexity analysis shows where the grammatical formalism lacks information about the structure of natural language.

In the manner described above, the analyses contribute to the development of the theory of grammatical formalisms. We do not doubt that these theories will be able to accommodate for the deficiencies that the complexity analyses reveal. So in a certain sense, we can regard these deficiencies as *computational pitfalls*.

With regard to the general statements concerning implementations we confine ourselves to the following. The recognition problems of the various grammatical formalisms that we consider are all intractable. This means that it is highly unlikely that efficient algorithms exist for these recognition problems. Hence algorithms for these recognition problems that are used in actual practice have to exploit one of the standard methods to handle intractable problems. These methods tackle intractable problems with varying success.

1.1.1 Motivation for our Approach

The subsequent chapters of this dissertation discuss six grammatical formalisms: attribute-value grammars, Categorial Unification Grammar, Functional Unification Grammar, Head-driven Phrase Structure Grammar, Lexical Functional Grammar, and Functional Grammar. These six grammatical formalisms share their affinity to logic. The first five are pure unification-based formalisms. They are based on the logical system of feature theory. The remaining one, Functional Grammar, is largely influenced by predicate logic. The logical systems on which these six grammatical formalisms are based are well studied and some complexity results are known. Unfortunately, these complexity results are not valid propositions about the complexity of the recognition problems of the grammatical formalisms as a whole.

The next section puts the complexity results that have been achieved in formalisms of logical systems in a different perspective. We will show that no insight in the complexity of a grammatical formalism is gained by looking only at the complexity of some isolated part of the grammatical formalism. Hence complexity analyses will only contribute to computational linguistics if the analyzed formalizations are connected closely with actual grammatical formalisms. Obviously, we want the complexity results in this thesis to be useful statements about the difficulty of grammatical formalisms in computational linguistics. Therefore, we analyze the complexity of each of the six grammatical formalisms as a whole, instead of restricting our attention to the underlying logical systems.

1.1.2 Assessing Complexity Results

The unification-based formalisms illustrate well that the complexity results obtained for their recognition problems and the complexity results obtained for their underlying logical systems may differ. Some recent formalizations of feature theory are the deterministic finite automata of Kasper and Rounds (1990); Smolka's (1992) first-order predicate logic; Blackburn and Spaan's (1993) modal logic; and the feature algebra of Baader, Bürckert, Nebel, Nutt and Smolka (1993).

These formalisms describe the use of feature theory in computational linguistics. Although they are a source of interesting technical research, the complexity results that have been achieved offer little help to actual computational linguists.

The unification-based formalisms used in computational linguistics do not consist of bare feature theories. These formalisms combine grammar components with feature theories. We claim that the complexity results from the formalizations of feature theory do not hold for a unification-based grammar that combines the feature theory with a grammar component.

On upper bounds. The complexity of feature theories does not provide an upper bound for the complexity of unification-based formalisms that incorporate these theories. A novice in complexity theory might expect that a problem is not harder than the problem's hardest component. However, combining problems may yield a problem that is harder than each of the problems considered separately. For instance, Johnson (1988) presents an attribute-value grammar that combines a context-free grammar with a simple feature theory. The satisfiability problem of this feature theory and the universal recognition problem of context-free grammars are both decidable. Nevertheless, Johnson shows that the universal recognition problem of the attribute-value grammar is undecidable in general.

From Johnson's (1988) work, we see that combining problems may change the complexity from decidable to undecidable. However, even when we confine ourselves to decidable problems, the complexity of the recognition problem of a unification-based grammar that uses some feature theory may be higher than the complexity of the satisfiability problem of that feature theory. In Chapter 2 we discuss restricted attribute-value grammars. These restricted attribute-value grammars combine a regular grammar with a feature theory, whose satisfiability problem is tractable. Nevertheless, we show that the recognition problem of these restricted attribute-value grammars is intractable.

On lower bounds. The complexity of feature theories does not always provide a lower bound for the complexity of unification-based grammatical formalisms that incorporate these theories. In general, it seems that the complexity of the combination of two problems is at least as difficult as the complexity of these two problems in isolation. However, if a problem A contains information about solutions for a problem B, and vice versa, then the combination of A and B may have lower complexity than A and B in isolation. For instance, let problem A be the complement of problem B. Then the combinations "A or B" and "A and B" have trivial solutions: "always answer yes" and "always answer no."

To be somewhat more specific, consider the two examples below. They show that in some special cases the complexity of a unification-based grammar may be lower than the complexity of its feature theory. Hence care has to be taken against drawing hasty conclusions about the lower bound complexity of the unification-based grammar from the complexity of the feature theory.

1.1.1. EXAMPLE.

- Assume a class of grammars that generate finite languages. The combination of a feature theory with a grammar from this class yields a unification-based grammar that generates a finite language. Obviously, the recognition problem of this unification-based grammar does not depend on the satisfiability problem of the feature theory. Hence the lower bound complexity of this class of unification-based grammars is not provided by the complexity of the feature theory.
- The feature theory does not provide a lower bound if the unification-based grammar uses only a fragment of the feature theory. This happens, for instance, when the formalism of the unification-based grammar restricts the unification operation of the feature theory.

  One may object that the restriction of the unification operation should be incorporated in the formalization of the feature theory. Thus reducing the complexity of the satisfiability problem of the feature theory. However, there is no predefined way to construct unification-based grammars from a feature theory. So there may be many blurred restrictions on the unification. These blurred restrictions are the reason that the formalization of the feature theory may be too expressive and that the unification-based grammar uses only a fragment of the feature theory.

1.2 An Outline of this Thesis

The subsequent chapters of this dissertation were originally written as a collection of separate papers. These different papers were written for people with different backgrounds. Consequently, the supposed readers of this thesis will form a hybrid group. In order to make this work accessible to each individual reader, the chapters are set up as self-contained parts. Nevertheless, there are some dependencies between the material covered in the different chapters. In these cases some overlap occurs, mostly in the form of definitions.

We conclude this chapter with a brief summary of the subsequent chapters and some useful directions for reading.

Chapter 2: Johnson (1988) shows that the recognition problem for attribute-value grammars (AVGs) is undecidable. Therefore, the general form of AVGs is of no practical use. In this chapter we study a very restricted form of AVG, for which the recognition problem is decidable, the R-AVG. First we show that the recognition problem of R-AVGs is *NP*-complete. Then we show that the R-AVG formalism captures more than the context-free languages. Finally we introduce a variation on the so-called off-line parsability constraint, the honest parsability constraint, which lets different types of R-AVG coincide precisely with well-known time complexity classes.

Chapter 3: This chapter starts with the preliminaries of complexity theory and feature theory. Then we introduce Categorial Unification Grammar and analyze its complexity. We show that the recognition problem of Categorial Unification Grammar is *NP*-complete. A reduction from 3-SATISFIABILITY proves the *NP*-hard lower bound of the recognition problem

Chapters 4, 5, and 6: In these chapters we first introduce the formalisms in question: Functional Unification Grammar, Head-driven Phrase Structure Grammar, and Lexical Functional Grammar. Then we prove that primitive fragments of these formalisms can simulate Categorial Unification Grammar. These simulations prove the *NP*-hardness of the recognition problems of Functional Unification Grammar, Head-driven Phrase Structure Grammar, and Lexical Functional Grammar. Furthermore, the recognition problems of these grammatical theories are shown to be *NP*-complete. Lower and upper bounds on the weak generative capacity of these grammatical theories follow from the simulation and the complexity results.

Chapter 7: This chapter starts with the preliminaries of complexity theory. Then we introduce Dik's (1989) major revision of the theory of Functional Grammar in a step-wise manner. We introduce Functional Grammar by means of many examples that clarify the relevant parts of Functional Grammar. We distinguish four processes within the generation process of Functional Grammar: the fund formation process, the clause structure formation process, the form specification process, and the order specification process. These four processes are considered one after the other. We formulate each process as a decision problem, and determine its complexity.

Chapter 8: In this final chapter we present our overall conclusions and suggest directions for further research.

Directions for reading. Chapter 2 assumes that the reader is familiar with formal language theory, complexity theory and feature theory (e.g., Hopcroft and Ullman 1979, Smolka 1992). Furthermore, familiarity with Johnson's (1988) work on

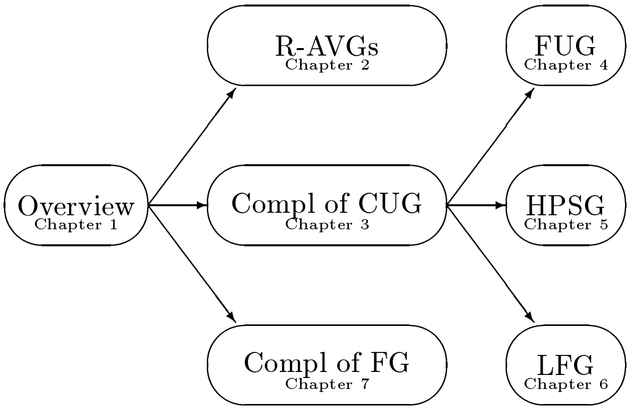


Figure 1.1: Direction for reading of the first seven chapters.

attribute-value grammars is preferred. The ideas on feature theory and complexity theory may be refreshed by the short introduction given in the next chapter: Section 3.2. The cursory reader may skip Chapter 2 on initial reading.

In Chapter 3, Section 3.2, we present the preliminaries on complexity theory and feature theory which are also required for the Chapters 4, 5, and 6. In Section 3.3 Categorial Unification Grammar is introduced and the NP-completeness of its recognition problem is proven. Knowledge of this section is required for the Chapters 4, 5, and 6. So Chapter 3 is essential for the understanding of the Chapters 4, 5, and 6. For the remaining part, these latter three chapters are self-contained, and may be considered in any order. Of course, the order in which the chapters are presented is the preferred order.

Chapter 7 is fully self-contained. The reader with an interest in Functional Grammar may leap to this chapter immediately. The chapter starts with some preliminaries on complexity theory. Except for the notions "honest function" and "oracle," these preliminaries overlap with Section 3.2.

Figure 1.1 depicts the directions for reading of the first seven chapters of this dissertation.

Chapter 2

Restricted Attribute-Value Grammars

2.1 Introduction

Although a universal feature theory does not exist, there is a general understanding of its objects. The objects of feature theories are abstract linguistic objects, e.g., an object "sentence," an object "masculine third person singular," an object "verb," an object "noun phrase." These abstract objects have properties like "tense," "number," "predicate," "subject." The values of these properties are either atomic, like "present" and "singular," or abstract objects, like "verb" and "noun-phrase." The abstract objects are fully described by their properties and their values. Multiple descriptions for the properties and values of the abstract linguistic objects are presented in the literature. Examples are:

1. Feature-graphs, which are labeled rooted directed acyclic graphs $G = (V, A)$, where F is a collection of labels, a sink in the graph represents an atomic value and the labeling function is an injective function $f : V \times A \mapsto F$.
2. Attribute-value matrices, which are matrices in which the entries consist of an attribute and a value or a reentrance symbol. The values are either atomic or attribute-value matrices.

From a computational point of view, all descriptions that are used in practical problems are equivalent; though there exist some theories with a considerably higher expressive power (Blackburn and Spaan 1993). For this chapter we adopt the feature-graph description, which we will define somewhat more formally in the next section. An attribute-value language (AVL) (Smolka 1992) consists of sets of logical formulas that describe classes of feature-graphs, by expressing constraints on the type of paths that can exist within the graphs. To wit: In a sentence like "a man walks" the edges labeled with "person" that leave the nodes labeled "a man" and "walks" should both end in a node labeled "singular." Such a constraint is called a path-equation in the attribute-value language.

A rewrite grammar (Chomsky 1956) can be enriched with an AVL to construct an attribute-value grammar (AVG), which consists of pairs of rewrite rules and logical formulas. The rewrite rule is applicable to a(nonterminal in a production only if the logical formula that expresses the relation between left- and right-hand side of the rule evaluates to true. The *recognition problem* for attribute-value grammars can be stated as: Given a grammar G and a string w, does there exist a derivation in G that respects the constraints given by its AVL, and that ends in w. As the derivation steps correspond to feature-graphs, this question can also be formulated as a question about the existence of a consistent sequence of feature-graphs that results in a feature-graph describing w. For the rewrite grammar, any formalism in the Chomsky hierarchy (from regular to type 0) can be chosen. From a computational point of view it is of course most desirable to restrict oneself to a formalism that on the one hand gives enough expressibility to describe a large fragment of the (natural) language, and on the other hand is restrictive enough to preserve feasibility. Perrault (1984) discusses the linguistic significance of such restrictions.

Johnson (1988) proved that attribute-value grammars that are as restrictive as being equipped with a rewrite grammar that is regular can already give rise to an undecidable recognition problem. Obviously, to be of any practical use, the rewrite grammar or the attribute-value language must be more restrictive. Johnson proposed to add the off-line parsability constraint, which is respected if the rewrite grammar has no chain rules or ϵ-rules.

We further investigate the properties of these restricted AVGs (R-AVGs). In the next section, we give some more formal definitions and notations. In Section 2.3 we show that the recognition problem of R-AVGs is NP-complete. In Section 2.4 we show that the class of languages generated by an R-AVG (R-AVGL) includes the class of context-free languages (CFL). It follows that any easily parsable class of languages (like CFL) is a proper subset of R-AVGL, unless $P = NP$. Likewise, R-AVGL is a proper subset of the class of context-sensitive languages, (CSL) unless $NP = PSPACE$ (the complexity class called polynomial space). In Section 2.5 we propose a further refinement on the off-line parsability constraint, which allows R-AVGs that respect this constraint to capture *precisely* complexity classes like NP or $NEXP$ (the complexity class called nondeterministic linear exponential time). That is, for any language L that has a parser that works in nondeterministic polynomial time, there exists an R-AVG, say G, such that $L = L(G)$. Though our refinement, the honest parsability constraint, is probably not a property that can be decided for arbitrary R-AVGs, we show that R-AVGs can be equipped with restricting mechanisms that enforce this property. The techniques that prove Theorem 2.4.1 and Theorem 2.5.3 result from Johnson's work. Therefore, the proofs of these theorems are deferred to the final sections of this chapter: Sections 2.7 and 2.8.

2.2 Definitions and Notation

2.2.1 Attribute-Value Grammars

The definitions in this section are in the spirit of Johnson (1988) and Smolka (1992). Consider three sets of pair-wise disjoint symbols.

A, the finite set of constants, denoted $(a, b, c, \ldots)$
V, the countable set of variables, denoted $(x, y, z, \ldots)$
L, the finite set of attributes, also called features, denoted $(f, g, h, \ldots)$

2.2.1. Definition. An *f-edge* from x to s is a triple (x, f, s) such that x is a variable, f is an attribute, and s is a constant or a variable. A *path* p is a, possibly empty, sequence of f-edges $(x_1, f_1, x_2), (x_2, f_2, x_3), \ldots, (x_n, f_n, s)$ in which the x_i are variables and s is either a variable or a constant. Often a path is denoted by the sequence of its edges' attributes, e.g., $p = f_1 \ldots f_n$. Let p be a path, sp denotes the *path that starts from* s, where s is a constant only if p is the empty path. If the path is non-empty, $p = f_1 \ldots f_n$ $(n \geq 1)$, then s is a variable. For paths sp and tq we write $sp \doteq tq$ iff p and q start in s and t respectively and end in the same variable or constant. The expression $sp \doteq tq$ is called a *path-equation.* A *feature-graph* is either a pair $(a, \emptyset)$, or a pair (x, E) where x is the root and E a finite set of f-edges such that

1. if (y, f, s) and (y, f, t) are in E, then $s = t$;
2. if (y, f, s) is in E, then there is a path from x to y in E;
3. if (y, f, s) is in E, then there is a no path from s to y in E.

2.2.2. Definition. An *attribute-value language* $\mathcal{A}(A, V, L)$ consists of sets of logical formulas that describe feature-graphs, by expressing constraints on the type of paths that can exist within the graphs.

- The *terms* of an attribute-value language $\mathcal{A}(A, V, L)$ are the constants and the variables $s, t \in A \cup V$.
- The *formulas* of an attribute-value language $\mathcal{A}(A, V, L)$ are path-equations and Boolean combinations of path-equations. Thus all formulas are either $sp \doteq tq$, where sp and tq are paths, or $\phi \wedge \psi$, $\phi \vee \psi$, or $\neg\phi$, where ϕ and ψ are formulas.

Assume a finite set of lexical forms, Lex, and a finite set of categories Cat. The set Lex will play the role of the set of terminals and the set Cat will play the role of the set of nonterminals in the productions.

2.2.3. Definition. A *constituent structure tree* (CST) is a labeled tree in which the internal nodes are labeled with elements of Cat and the leaves are labeled with elements of Lex. The string that is formed by the, left-to-right, concatenation of the labels of the leaves is called the *yield* of the constituent structure tree.

2.2.4. Definition. Let T be a constituent structure tree and F be a set of formulas in an attribute-value language $\mathcal{A}(A, V, L)$. An *annotated constituent structure tree* is a triple $<T, F, h>$, where h is a function that maps internal nodes in T onto variables in F.

2.2.5. Definition. A *lexicon* is a finite subset of Lex $\times$ Cat $\times$ $\mathcal{A}(A, \{x_0\}, L)$. A set of *syntactic rules* is a finite subset of $\bigcup_{i\geq 1}$ Cat $\times$ Cat^i $\times$ $\mathcal{A}(A, \{x_0, \ldots, x_i\}, L)$. An *attribute-value grammar* is a triple <Lexicon, Rules, Start>, where Lexicon is a lexicon, Rules is a set of syntactic rules and Start, the start symbol, is an element of Cat.

2.2.6. Definition.

1. **(Balcázar, Díaz and Gabarró 1988, p .150)** A class $\mathcal{C}$ of sets is *recursively presentable* iff there is an effective enumeration $M_1, M_2, \ldots$ of deterministic Turing machines which halt on all their inputs, and such that $\mathcal{C} = \{L(M_i) \mid i = 1, 2, \ldots\}$.
2. We say that a class of grammars $\mathcal{G}$ is *recursively presentable* iff the class of sets $\{L(G) \mid G \in \mathcal{G}\}$ is recursively presentable.

2.2.2 Restricted Attribute-Value Grammars

The only formulas that are allowed in the *attribute-value language* of restricted attribute-value grammars (R-AVGs) are path-equations and conjunctions of path-equations, i.e., disjunctions and negations are out. We will denote the attribute-value language of an R-AVG by $\mathcal{A}'(A, V, L)$ to make the distinction clear. The CST of an R-AVG is produced by a chain rule and ϵ-rule free regular grammar. The CST of an R-AVG can be either a left-branching or a right-branching tree, since the grammar contains at most one nonterminal in each rule.

2.2.7. Definition. The set of *syntactic rules* of a restricted attribute-value grammar is a subset of $\bigcup_{i\geq 1, k\leq 1}$ Cat $\times$ Lex^i $\times$ Cat^k $\times$ $\mathcal{A}'(A, \{x_0, x_k\}, L)$. A *restricted attribute-value grammar* is a pair <Rules, Start>, where Rules is a set of syntactic rules and Start, the start symbol, is an element of Cat.

2.2.8. Definition. An R-AVG <Rules, Start> *generates* an annotated constituent structure tree $<T, F, h>$ iff

1. the root node of T is Start, and
2. every internal node of T is licensed by a syntactic rule, and
3. the set F is consistent, i.e., describes a feature-graph.

Let $\phi[x/y]$ stand for the formula ϕ in which variable y is substituted for variable x. An internal node v of an annotated constituent structure tree is *licensed* by a syntactic rule $(c_0, l_1, \ldots, l_i, \phi)$ iff

1. the node v is labeled with category c_0, $h(v) = n_0$, and
2. all daughters of v are leaves, which are labeled with $l_1 \ldots l_i$, and
3. $\phi[x_0/n_0]$ is in the set F.

$S \to \sharp\ F$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$S \to \sharp\ T$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{NEW}$	$S \to \$$ $x_0\ \text{ASSIGN V} \doteq +$
$F \to 0\ F$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$F \to 1\ F$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$F \to p\ F$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$
$F \to \overline{p}\ F$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$F \to p\ T$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{NEW}$	$F \to \overline{p}\ T$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{NEW}$
$T \to 0\ T$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{NEW}\ 0 \doteq x_1\ \text{NEW}$	$T \to 1\ T$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{NEW}\ 1 \doteq x_1\ \text{NEW}$	
$T \to p\ A$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{NEW V} \doteq +$	$T \to \overline{p}\ A$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN} \wedge$ $x_0\ \text{NEW V} \doteq -$	
$A \to B$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$A \to S$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	
$B \to 0\ B$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$B \to 1\ B$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	
$B \to p\ A$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	$B \to \overline{p}\ A$ $x_0\ \text{ASSIGN} \doteq x_1\ \text{ASSIGN}$	

Table 2.1: The syntactic rules of R-AVG G.

An internal node v of an annotated constituent structure tree is *licensed* by a syntactic rule $(c_0, l_1, \ldots, l_i, c_1, \phi)$ iff

1. the node v is labeled with category c_0, $h(v) = n_0$, and
2. one of v's daughters is an internal node, v_1, which is labeled with category c_1, and $h(v_1) = n_1$, and
3. the daughters of v that are leaves are labeled with $l_1 \ldots l_i$, and
4. $\phi[x_0/n_0, x_1/n_1]$ is in the set F.

2.3 Complexity of the Recognition Problem

In this section we will discuss the complexity of the recognition problem of R-AVGs. First, we present an R-AVG G, whose recognition problem is *NP*-hard. Second, we show that the recognition problem for arbitrary R-AVGs is in *NP*.

2.3.1 The Recognition Problem is *NP*-hard

We will analyze the R-AVG G with start symbol S and whose syntactic rules are given in Table 2.1. A polynomial time many-one reduction f from the *NP*-complete problem SATISFIABILITY proves that the recognition problem of G is *NP*-hard. The *NP*-complete problem SATISFIABILITY (SAT) is defined as follows.

2.3.1. DEFINITION. SATISFIABILITY
INSTANCE: A formula φ, from propositional logic, in conjunctive normal form.
QUESTION: Is there an assignment of truth-values to the propositional variables of φ, such that φ is true?

Now we will first present the reduction f. Then we will show that this reduction is computable in polynomial time. Finally we prove that the reduction maps all and only all satisfying formulas onto strings that G generates. Thus we have proven that the recognition problem of the restricted attribute-value grammar G is *NP*-hard. We strengthen this result in Section 2.3.2 by showing that the recognition problem of arbitrary restricted attribute-value grammars is *NP*-complete.

The reduction from SAT to the recognition problem of G maps propositional logical formulas onto strings. We assume, without loss of generality, that the indices of the propositional logical variables are in binary representation. This reduction, f, is defined by the following four equations:

$$
\begin{array}{rcll}
f(\gamma_1 \wedge \ldots \wedge \gamma_m) & = & \sharp f(\gamma_1) \ldots \sharp f(\gamma_m)\$ & (\gamma_i \text{ a clause}) \\
f(l_1 \vee \ldots \vee l_m) & = & f(l_1) \ldots f(l_m) & (l_i \text{ a literal}) \\
f(p_i) & = & i\, p & (p_i \text{ a positive literal, } i \text{ in} \\
 & & & \text{binary representation}) \\
f(\overline{p_i}) & = & i\, \overline{p} & (\overline{p_i} \text{ a negative literal, } i \text{ in} \\
 & & & \text{binary representation})
\end{array}
$$

2.3.2. LEMMA. *The reduction f is computable in linear time, with respect to the size of the formula.*

Proof. By induction on the construction of SAT formulas. □

Lemma 2.3.2 proves that the reduction f is computable in polynomial time. Lemma 2.3.4 proves that the reduction maps all and only all satisfying formulas onto strings that G generates. These two lemmas together prove the *NP*-hardness (Theorem 2.3.5). The following lemma is used in the proof of Lemma 2.3.4.

2.3.3. LEMMA. *Let f be the reduction above, φ be a formula and $f(\varphi) = \sharp u_1 \ldots u_k\$$. If the R-AVG G from Table 2.1 generates an annotated constituent structure tree $<T, F, h>$, where the root node of T, v_0, is labeled S, the yield of T is $\sharp u_1 \ldots u_k\$$, and $h(v_0) = n_0$, then there is in the feature-graph described by F from node n_0, a path $\langle \text{ASSIGN}\, b_1 \ldots b_l \rangle$ which leads to the value $+$ and some $u_i = b_1 \ldots b_l\, p$, or which leads to the value $-$ and some $u_i = b_1 \ldots b_l\, \overline{p}$.*

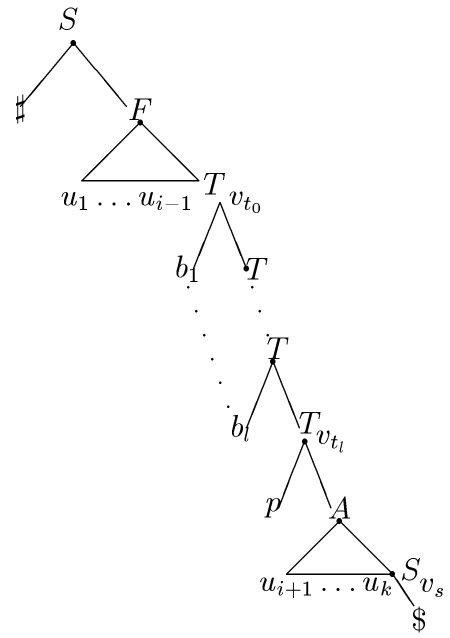


Figure 2.1: A constituent structure tree with yield $\sharp u_1 \ldots u_k \$$ $(u_i = b_1 \ldots b_l\, p)$.

Proof. The reduction f maps formula φ onto the string $\sharp u_1 \ldots u_k \$$, where $u_i = f(l_i)$ and l_i is a literal.

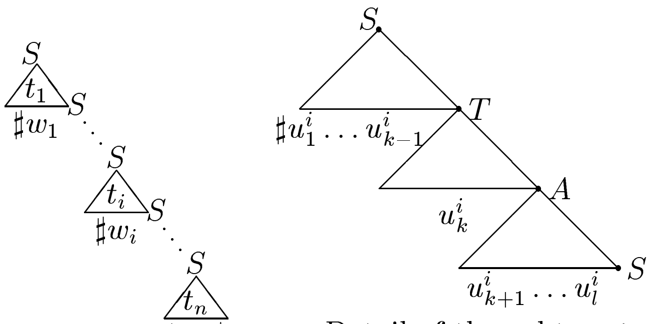
Let us assume that $u_i = b_1 \ldots b_l\, p$ $(i > 1)$, the cases where $u_i = b_1 \ldots b_l\, \overline{p}$ or $i = 1$ are similar. Let R-AVG G generate the annotated CST $<T, F, h>$, with $h(v_0) = n_0$. The yield of T is $\sharp u_1 \ldots u_k \$$ iff T has the form given in Figure 2.1.

Now let us consider the nodes in the constituent structure tree given in Figure 2.1, and the rules given in Table 2.1.

- The lowest node with label S, say v_s, is licensed by the rule with formula x_0 ASSIGN V $\doteq +$.
- The lowest node with label T, say v_{t_l}, is licensed by the rule with formula x_0 ASSIGN $\doteq x_1$ ASSIGN $\wedge$ x_0 NEW V $\doteq +$.
- Let us call the upper node with label T v_{t_0}. The mother of node v_{t_0} is licensed by a rule with formula x_0 ASSIGN $\doteq x_1$ ASSIGN $\wedge$ x_0 ASSIGN $\doteq x_1$ NEW
- The other nodes with label T are licensed by the rules with the formula x_0 ASSIGN $\doteq x_1$ ASSIGN $\wedge$ x_0 NEW $0 \doteq x_1$ NEW, or x_0 ASSIGN $\doteq x_1$ ASSIGN $\wedge$ x_0 NEW $1 \doteq x_1$ NEW.
- All remaining nodes are licensed by rules with formula x_0 ASSIGN $\doteq x_1$ ASSIGN.

Now let us consider the set of formulas F, and assume that the function h maps a node v_j in the constituent structure tree onto a node n_j in the feature-graph.

- Then the set of formulas F contains a sequence of formulas n_i ASSIGN $\doteq$ n_{i+1} ASSIGN that starts with $n_i = n_0$ and ends with $n_{i+1} = n_{t_0}$.



Detail of the subtree t_i.

Figure 2.2: A constituent structure tree with yield $\sharp w_1 \dots \sharp w_n \$$.

- Let the mother of node v_{t_0} be called v'_{t_0}. Then the set of formulas F also contains n'_{t_0} ASSIGN $\doteq n_{t_0}$ NEW.
- Furthermore, the set of formulas F contains a formula n_{t_0} NEW $1 \doteq n_{t_1}$ NEW, if $b_1 = 1$, and n_{t_0} NEW $0 \doteq n_{t_1}$ NEW, if $b_1 = 0$.
- Similarly, the set of formulas F contains the formulas $n_{t_{i-1}}$ NEW $b_i \doteq n_{t_i}$ NEW $(1 < i \leq l)$.
- Finally the set of formulas F contains a formula n_{t_l} NEW V $\doteq +$.

Hence from node n_0 there is a path $\langle \text{ASSIGN}\, b_1 \dots b_l\, \text{V} \rangle$ that leads to the value $+$.

The cases where $u_i = b_1 \dots b_l\, \overline{p}$, or $i = 1$ are the same. □

2.3.4. LEMMA. *Let f be the reduction above, φ be a formula in conjunctive normal form and $f(\varphi) = \sharp w_1 \dots \sharp w_n \$$, $w_i = u_1^i \dots u_k^i$. The R-AVG G from Table 2.1 generates an annotated constituent structure tree $<T, F, h>$, where the root node of T, v_0, is labeled S, the yield of T is $\sharp w_1 \dots \sharp w_n \$$ and $h(v_0) = n_0$ iff the formula φ is satisfiable.*

Proof. *(Sketch of the proof.)*

Only if: Let the R-AVG G generate the annotated constituent structure tree. The proof of Lemma 2.3.3 shows that for each substring $\sharp\, w_i = u_1^i \dots u_k^i$ there is, from the node n_0, a path $\langle \text{ASSIGN}\, b_1 \dots b_l\, \text{V} \rangle$, which leads to the value $+$ and some u_j^i is $b_1 \dots b_l\, p$, or which leads to the value – and some u_j^i is $b_1 \dots b_l\, \overline{p}$.

Clearly, the path $\langle \text{ASSIGN}\, b_1 \dots b_l\, \text{V} \rangle$ from node n_0 cannot lead to the values $+$ and – at the same time. So the assignment g for formula φ that assigns value true to variable $p_{b_1 \dots b_l}$ of formula φ if there is a path $\langle \text{ASSIGN}\, b_1 \dots b_l\, \text{V} \rangle$ from n_0 that leads to value $+$, and assigns false to this variable, if this path leads to value – is a consistent satisfying assignment for φ.

If: Let φ be a satisfiable formula. Then there is a consistent assignment g which assigns true to the formula φ.

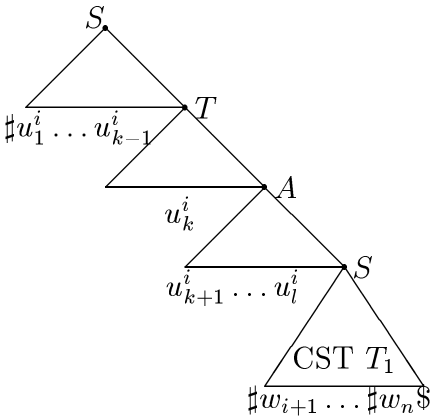


Figure 2.3: The constituent structure tree T' for yield $\sharp w_i \ldots \sharp w_n\$$.

We know from the proof of Lemma 2.3.3 that if G generates the annotated constituent structure tree $<T, F, h>$, the CST T has the form of the leftmost CST in Figure 2.2. Let the subtrees in the CST T have the form of the rightmost CST in Figure 2.2 only if assignment g assigns true to the k-th literal in the i-th clause of φ. We will now show, by induction on the number of subtrees, that the R-AVG G generates the annotated constituent structure tree $<T, F, h>$ with yield $\sharp w_1 \ldots \sharp w_n\$$.

Basis: Let CST T have yield $\sharp w_n\$$. Then G generates annotated constituent structure tree $<T, F, h>$, where path $\langle \text{ASSIGN}\, b_1 \ldots b_l\ \text{V}\rangle$ leads to value + if the k-th literal in the n-th clause is $p_{b_1 \ldots b_l}$, and path $\langle \text{ASSIGN}\, b_1 \ldots b_l\ \text{V}\rangle$ leads to value – if the k-th literal in the n-th clause is $\overline{p_{b_1 \ldots b_l}}$.

Induction: The hypotheses states that R-AVG G generates $<T_1, F, h>$, where the root node of T_1, v_0, is labeled S, the yield of T_1 is $\sharp w_{i+1} \ldots \sharp w_n\$$ and $h(v_0) = n_0$. We now claim that G generates the annotated constituent structure tree $<T', F', h>$, where T' is given in Figure 2.3, the root node of T', v'_0, is labeled S, the yield of T' is $\sharp w_i \ldots \sharp w_n\$$ and $h(v'_0) = n'_0$.

On the one hand, we know from the proof of Lemma 2.3.3 that from the node $h(v'_0) = n'_0$ there is a path $\langle \text{ASSIGN}\, b_1 \ldots b_l\ \text{V}\rangle$ that leads to value + if the k-th literal in the i-th clause is $p_{b_1 \ldots b_l}$, and that leads to value – if the k-th literal in the i-th clause is $\overline{p_{b_1 \ldots b_l}}$. On the other hand, we know that the two edges with label ASSIGN from the nodes n_0 and n'_0 lead to the same node.

So we have to check that the path $\langle \text{ASSIGN}\, b_1 \ldots b_l\ \text{V}\rangle$ from node n'_0 does not clash with any path that start with attribute ASSIGN, from node n_0. Assume that there is such a clash. This clash occurs iff the paths $\langle \text{ASSIGN}\, b_1 \ldots b_l\ \text{V}\rangle$ from the node n_0 and n'_0 lead to different values. However, this means that the assignment g is inconsistent. Thus the assumption is false and the claim holds. □

2.3.5. THEOREM. *Let w be a string and let G be the grammar from Table 2.1, then the recognition problem for w and G is NP-hard.*

Proof. By the Lemmas 2.3.2 and 2.3.4. □

2.3.2 The Recognition Problem is *NP*-complete

Now we will show that the complexity result of the recognition problem for restricted attribute-value grammars can be strengthened. Below we will prove an additional *NP* upper bound for an arbitrary string and a restricted arbitrary grammar, which results in an *NP*-complete recognition problem.

2.3.6. THEOREM. *Let w be any string and G be any restricted attribute-value grammar. Then the recognition problem for w and G is NP-complete.*

Proof. (Sketch of the proof.)
An *NP*-hard lower bound is proven by Theorem 2.3.5. An *NP* upper bound is proven when we can guess a solution, and check that solution in polynomial time. The *NP* upper bound is proven as follows.

Given a string w and a grammar G, we can guess a sequence of rules that encode the derivation for w. This sequence is linear with respect to the size of the string w. The guessed rules describe a constituent structure tree and a set of formulas. First, we must check that the constituent structure tree described by the rules has yield w. Second, we have to check that the set of formulas describes some feature-graph.

The first check is trivial. The second check is performed by a minor modification of Smolka's (1992, Section 5) constraint solving algorithm. The algorithm of Smolka takes quadratic time, hence the checks are performed in polynomial time. □

2.4 Weak Generative Capacity

In the previous section we showed that the recognition problem for R-AVGs is *NP*-complete. This seems to indicate that although the mechanism for generating CSTs in R-AVGs is extremely simple, the generative capacity of R-AVGs is different from the generative capacity of, e.g., context-free languages (CFLs), which have a polynomial time parsing algorithm (Earley 1970). Yet, a priori, there may exist CFLs that do not have an R-AVG.

2.4.1. THEOREM. *Let L be a context-free language. There exists an R-AVG G such that $L = L(G)$.*

Proof. If L is a context-free language, then there exists a context-free grammar G' in Greibach normal form such that $L = L(G')$. From this grammar G', we can construct a pushdown automaton M that accepts exactly the words in $L(G') = L$. Such a pushdown automaton M is actually a finite state automaton M' with a stack S. The finite state automaton M' may be simulated by a chain rule and ϵ-rule free regular grammar. Furthermore, we can construct an attribute-value language $\mathcal{A}'(A, V, L)$ that simulates the stack S. Thus it should be clear that there exists an R-AVG G that produces word w iff $w \in L(G')$. Details of this construction are deferred to Section 2.7. □

From this we can draw the conclusion that the class of context-free languages is indeed a proper subset of the class of R-AVG languages, unless $P = NP$.

2.4.2. THEOREM. *Let $\mathcal{C}$ be a recursively presentable class of grammars such that:*

1. *$G \in \mathcal{C}$ can be decided in time polynomial in the size of G, $|G|$*
2. *$G \overset{*}{\Rightarrow} w$ can be decided in time polynomial in the size of G and w, $|G| + |w|$.*

If every R-AVG G has a grammar in $\mathcal{C}$ then $P = NP$. In fact, for every language L in NP there is an explicit deterministic polynomial time algorithm.

Proof. Let L be a language in NP and $w \in \{0,1\}^*$. In Section 2.3 we provided an R-AVG G and a reduction that maps any formula F onto a string w_F such that $G \overset{*}{\Rightarrow} w_F$ iff $F \in SAT$. It was also shown that any R-AVG has a nondeterministic polynomial time, hence deterministic exponential time, recognition algorithm. Suppose every R-AVG G has a grammar in $\mathcal{C}$. Then there exists a $G' \in \mathcal{C}$ with $L(G') = L(G)$. We can decide in polynomial time whether $w_F \in L(G)$ for any w_F. So $P = NP$.

If every R-AVG G has a grammar in $\mathcal{C}$, then the algorithm for deciding "$w \in L$?" consists of: use Cook's reduction to produce a formula F that is satisfiable iff $w \in L$; use the reduction f from Section 2.3 to produce w_F and R-AVG G; enumerate grammars in $\mathcal{C}$ for the first grammar G' that has a description of length less than $\log\log|w|$ for which $L(G) \cap \{0,1\}^{\leq \log\log|w|} = L(G') \cap \{0,1\}^{\leq \log\log|w|}$ accept iff $w \in L(G')$. This gives a polynomial time algorithm that erroneously accepts or rejects w for only a finite number of strings w. The theorem now follows from the fact that both P and NP are closed under finite variation. □

2.4.3. COROLLARY. *If R-AVGs generate only context-free languages then $P = NP$.*

In fact it can be shown directly that R-AVGs also produce languages that are not context-free.

2.4.4. THEOREM. *The context-sensitive language $\{a^n b^n c^n\}$ is generated by an R-AVG.*

Proof. (Sketch of the proof.)
Typically, the R-AVG that generates the language $\{a^n b^n c^n\}$ first generates an amount of a's then an amount of b's and finally an amount of c's. Let us assume that the grammar generates i a's. During the derivation, the feature-graph can be used to store the amount of a's that is produced. Once the grammar starts to produce b's , the feature-graph will force the grammar to generate exactly i b's and next to generate exactly i c's as well. □

2.5 The Honest Parsability Constraint and Consequences

According to Theorem 2.4.2, it is unlikely that the languages generated by R-AVGs can be limited to those languages with a polynomial time recognition algorithm. In Section 2.3 we showed that all R-AVGs have nondeterministic polynomial time algorithms. Is it perhaps the case that any language that has a nondeterministic polynomial time recognition algorithm can be generated by an R-AVG. Does there exist a tight relation between time bounded machines and R-AVGs as, e.g., between linear bounded automata and context-sensitive languages? The answer is that the off-line parsability constraint that forces the R-AVG to have no chain- or ϵ-rules is just too restrictive to allow such a connection. The following trick to alleviate this problem has been observed earlier in complexity theory. The off-line parsability constraint (OLP) (Johnson 1988) relates the amount of "work" done by the grammar to produce a string linearly to the number of terminal symbols produced. It is therefore a sort of honesty constraint that is also demanded of functions that are used in, e.g., cryptography. There the deal is, for each polynomial amount of work done to compute the function at least one bit of output must be produced. In such a way, for polynomial time computable functions one can guarantee that the inverse of the function is computable in nondeterministic polynomial time.

As a more liberal constraint on R-AVGs admitting chain- and ϵ-rules we propose an analogous variation on the OLP

2.5.1. Definition. A grammar G satisfies the *honest parsability constraint* (HPC) iff there exists a polynomial p such that for each w in $L(G)$ there exists a derivation with at most $p(|w|)$ steps.

From Smolka's constraint solving algorithm and Section 2.3 it trivially follows that any *attribute-value grammar that satisfies the HPC* (HP-AVG) has an *NP* recognition algorithm. The problem with the HPC is of course that it is not a syntactic property of grammars. The question whether a given AVG satisfies the HPC (or the OLP for that matter) may well be undecidable. Nonetheless, we can produce a set of rules that, when added to an attribute-value grammar *enforces* the HPC. The newly produced language is then a subset of the old produced language with an *NP* recognition algorithm. Because of the fact that our addition may simulate any polynomial restriction, we regain the full class of AVGs that satisfy the HPC. In fact

2.5.2. Theorem. *The class,* P-AVGL, *of languages produced by the HP-AVGs is recursively presentable.*

We will give a detailed construction of such a set of rules in Section 2.8. The existence of such a set of rules and the work of Johnson now gives the following theorem.

2.5.3. Theorem. *For any language L that has an NP recognition algorithm, there exists a restricted attribute-value grammar G that respects the HPC and such that $L = L(G)$.*

Proof. (*Sketch of the proof.*)
Let M be the Turing machine that decides $w \in L$. Use a variation of Johnson's construction of a Turing machine to create an R-AVG that can produce any string w that is recognized by M. Add the set of rules that guarantee that only strings that can be produced with a polynomial number of rules can be produced by the grammar. The details are given in Section 2.8. □

2.6 Veer out the HPC

Instead of creating a counter of logarithmic size as we do in Section 2.8, it is quite straightforward to construct a counter of linear size (or exponential size if there is enough time). In fact, for well-behaved functions, the construction of a counter gives a method to enforce any desired time bound constraint on the recognition problem for attribute-value grammars. For instance, for nondeterministic exponential time we could define the *linear dishonest parsability constraint* (LDP) (allowing a linear exponential number of steps) which would give

2.6.1. Theorem. *The class of languages generated by R-AVGs obeying the LDP condition is exactly NEXP.*

2.7 Simulating a Context Free Grammar in GNF

This section contains the details of the construction presented in the proof of Theorem 2.4.1. We will show that every context-free language is generated by some restricted attribute-value grammar.

A context-free grammar (CFG) is a quadruple $\langle N, \Sigma, P, S\rangle$, where N is a set of nonterminals, Σ is a set of terminals, P is a set of productions, and $S \in N$ is the start nonterminal. A CFG is in *Greibach normal form* (GNF) iff the productions are of one of the following forms, where $a \in \Sigma$, $A \in N$, $A_1 \ldots A_n \in N \setminus \{S\}$ and ϵ the empty string (cf., Hopcroft and Ullman 1979, Sudkamp 1988):

$$\begin{array}{lcl} A & \rightarrow & a\, A_1 \ldots A_n \\ A & \rightarrow & a \\ S & \rightarrow & \epsilon \end{array}$$

Given a GNF $G = \langle N, \Sigma, P, S\rangle$, we can construct a restricted attribute-value grammar (R-AVG) G' that simulates grammar G. R-AVG G' consists of the same set of nonterminals and terminals as GNF G. The productions of R-AVG G' are described by Table 2.2. The only two attributes of R-AVG G' are TOP and REST. R-AVG G' contains $|N| + 1$ atomic values, one atomic value for each nonterminal

Productions of GNF G		Productions of R-AVG G'	
$S \to aA_1 \ldots A_n$	$\rightsquigarrow$	$S \to aA_1$	
		$\text{PUSH}(A_2 \ldots A_n) \wedge \text{EMPTY-STACK}$	
$A \to aA_1 \ldots A_n$	$\rightsquigarrow$	$A \to aA_1$	
		$\text{PUSH}(A_2 \ldots A_n)$	$(A \neq S)$
$S \to a$	$\rightsquigarrow$	$S \to aB$	$\forall B \in N \setminus \{S\}$
		$\text{POP}(B) \wedge \text{EMPTY-STACK}$	
$S \to a$	$\rightsquigarrow$	$S \to a$	
		EMPTY-STACK	
$A \to a$	$\rightsquigarrow$	$A \to aB$	$\forall B \in N \setminus \{S\}$
		$\text{POP}(B)$	$(A \neq S)$
$A \to a$	$\rightsquigarrow$	$A \to a$	
		EMPTY-STACK	
$S \to \epsilon$	neglected		

Table 2.2: Simulating productions of GNF G by R-AVG G'

and the special atomic value \$. The R-AVG G' uses the feature-graph to encode a pushdown stack, similar to the encoding of a list. The stack will be used to store the nonterminals that still have to be rewritten.

The three syntactic abbreviations below are used to clarify the simulation. We represent a stack by a Greek letter, or a string of symbols; the top of the stack is the leftmost symbol of the string. Let x_0 encode a stack γ, then the formulas in the abbreviation $\text{PUSH}(A_0 \ldots A_n)$ express that x_1 encodes a stack $A_0 \ldots A_n\gamma$. Likewise, the formulas in the abbreviation $\text{POP}(A)$ express that x_0 encodes a stack $A\gamma$, and x_1 encodes the stack γ. The abbreviation EMPTY-STACK expresses that x_0 encodes an empty stack.

$$
\begin{array}{rll}
\text{PUSH}(A_0 \ldots A_n) & \text{stands for} & x_1\,\text{TOP} \doteq A_0 \wedge \\
 & & x_1\,\text{REST TOP} \doteq A_1 \wedge \\
 & & \vdots \\
 & & x_1\,\text{REST}^n\text{TOP} \doteq A_n \wedge \\
 & & x_1\,\text{REST}^{n+1} \doteq x_0 \\
\text{POP}(A) & \text{stands for} & x_0\,\text{TOP} \doteq A \wedge \\
 & & x_0\,\text{REST} \doteq x_1 \\
\text{EMPTY-STACK} & \text{stands for} & x_0 \doteq \$
\end{array}
$$

We have to prove that GNF G and its simulation by R-AVG G' generate (almost) the same language. Obviously, R-AVG G' cannot generate the empty string. However, for all non-empty strings the following theorem holds.

2.7.1. THEOREM. *Start symbol S of GNF G derives string α ($\alpha \in \Sigma^+$) iff start symbol S of R-AVG G' derives string α with the empty stack.*

Proof. There are two cases to consider. First, S derives string α in one step. Second,

S derives string α in more than one step. Lemma 2.7.2 below is needed in the proof of the second case.

In one step. Let start symbol S derive string α in one step. GNF G contains a production $S \to \alpha$ iff R-AVG G' contains a production $S \to \alpha$ with the equation EMPTY-STACK. So S derives α in a derivation of GNF G iff S derives α with an empty stack in the derivation of R-AVG G'.

In more than one step. Start symbol S of GNF G derives string $\alpha = \beta\beta'$ in more than one step iff there is a leftmost derivation $S \overset{*}{\Rightarrow} \beta A \Rightarrow \beta\beta'$. GNF G contains production $A \to \beta'$ iff R-AVG G' contains production $A \to \beta'$ with the equation EMPTY-STACK. By Lemma 2.7.2 $S \overset{*}{\Rightarrow} \beta A$ for GNF G iff $S \overset{*}{\Rightarrow} \beta A$ with the empty stack for R-AVG G'. Hence S derives α for GNF G iff S derives α with empty stack for R-AVG G'. $\square$

2.7.2. LEMMA. *Start symbol S derives $\alpha A\gamma$ ($\alpha \in \Sigma^+, A\gamma \in (N\backslash\{S\})^+$) in a leftmost derivation of GNF G iff nonterminal S derives αA with stack $\gamma\$$ ($\$$ is the bottom-of-stack symbol) in the derivation of R-AVG G'.*

Proof. The lemma is proven by induction on the length of the derivation.

Basis. If S derives $\alpha A\gamma$ in one step, then GNF G contains production $S \to \alpha A\gamma$ and R-AVG G' contains production $S \to \alpha A$ with stack $\gamma\$$. If S derives αA with stack $\gamma\$$ in one step, then R-AVG G' contains production $S \to \alpha A$ with stack $\gamma\$$ and GNF G contains production $S \to \alpha A\gamma$.

Induction. The induction hypotheses states that $S \overset{n}{\Rightarrow} \alpha A\gamma$ for GNF G iff $S \overset{n}{\Rightarrow} \alpha A$ with stack $\gamma\$$ for R-AVG G'. Next we distinguish three cases: GNF G contains production $A \to aA_1A_2\ldots A_n$; production $A \to a$ and $\gamma = B'\gamma'$; or production $A \to a$ and $\gamma = \epsilon$.

1. GNF G contains a production $A \to aA_1A_2\ldots A_n$. Hence there is a leftmost derivation $S \overset{n+1}{\Rightarrow} \alpha aA_1A_2\ldots A_n\gamma$. GNF G contains the production $A \to aA_1A_2\ldots A_n$ iff R-AVG G' contains a production $A \to aA_1$ with equation PUSH($A_2\ldots A_n$). Since the induction hypotheses states that there is a derivation $S \overset{n}{\Rightarrow} \alpha A$ with stack $\gamma\$$, there is a derivation $S \overset{n+1}{\Rightarrow} \alpha aA_1$ with stack $A_2\ldots A_n\gamma\$$.
2. GNF G contains a production $A \to a$ and $\gamma = B'\gamma'$. Hence there is a leftmost derivation $S \overset{n+1}{\Rightarrow} \alpha aB'\gamma'$. GNF G contains the production $A \to a$ iff R-AVG G' contains productions $A \to aB$ with equation POP(B), for all $B \in N \setminus \{S\}$. Hence by the induction hypotheses, there is a derivation $S \overset{n+1}{\Rightarrow} \alpha aB'$ with stack $\gamma'\$$.
3. GNF G contains a production $A \to a$ and $\gamma = \epsilon$. Then there is a leftmost derivation $S \overset{n+1}{\Rightarrow} \alpha a$. GNF G contains the production $A \to a$ iff R-AVG

G' contains production $A \to a$ with equation EMPTY-STACK. Hence by the induction hypotheses, there is a derivation $S \overset{n+1}{\Rightarrow} \alpha a$ with stack \$.

$\square$

Because every context-free language is generated by some GNF G, every context-free language is generated by some R-AVG G'.

2.8 Constructing an Honestly Parsable Attribute-Value Grammar

This section contains the details of the proof of Theorem 2.5.3. We show how to add a binary counter to an attribute-value grammar (AVG). This counter enforces the *honest parsability constraint* (HPC) upon the AVG. To keep this section legible we sometimes use the attribute-value matrices (AVMs) as descriptions. In Section 2.8.2, we show how to create a counter for the AVG. This counter will polynomially bound the number of rules that are applied in a derivation. Hence the counter guarantees that strings are produced by a polynomial amount of rules. In Section 2.8.3 we extend the syntactic rules and the lexicon of the AVG such that the counter that bounds the size of a derivation is added to the AVG.

2.8.1 Arithmetic by AVGs

We start with a little bit of arithmetic.

Natural numbers. The AVMs below encode natural numbers in binary notation. The sequences of attributes 0 and 1 in these AVMs encode natural numbers, from least- to most-significant bit. The attribute V has value 1 (or 0) iff it has a sister attribute 1 (or 0).

1. The AVMs $\begin{bmatrix} \textsc{v} & 0 \\ 0 & + \end{bmatrix}$ and $\begin{bmatrix} \textsc{v} & 1 \\ 1 & + \end{bmatrix}$ encode the natural numbers zero and one.
2. The AVMs $\begin{bmatrix} \textsc{v} & 0 \\ 0 & [F] \end{bmatrix}$ and $\begin{bmatrix} \textsc{v} & 1 \\ 1 & [F] \end{bmatrix}$ encode natural numbers iff the AVM $[F]$ encodes a natural number.

Syntactic rules that tests two numbers for equality. Assume a nonterminal A with some AVM $\begin{bmatrix} \textsc{n} & [F] \\ \textsc{m} & [H] \end{bmatrix}$, where $[F]$ and $[H]$ encode natural number x and y, respectively. We present one syntactic rule that derives from this nonterminal A a nonterminal B with AVM $\begin{bmatrix} \textsc{n} & [F] \\ \textsc{m} & [H] \end{bmatrix}$ if $x = y$.

Clearly, this simple test takes one step. A more sophisticated test, which also tests for inequality, would compare $[F]$ and $[H]$ bit-by-bit. Such a test would take an amount of derivation steps that is linear with respect to the size of the smallest AVM.

$$\boxed{\begin{array}{l} A \rightarrow\ B \\ \quad x_0\text{N} \doteq x_0\text{M} \\ \quad \wedge\, x_0 \doteq x_1 \end{array}}$$

Table 2.3: The rule to test two numbers for equality.

Syntactic rules that multiply by two. Assume a nonterminal A with some AVM $\left[\ \text{N}\ \ [F]\ \right]$, where $[F]$ encodes natural number x. We present one syntactic rule that derives from this nonterminal A a nonterminal B with the AVM $\left[\ \text{N}\ \ [H]\ \right]$, where $[H]$ encodes natural number $2x$.

The number N in $[H]$ equals two times N in $[F]$ iff the least-significant bit of N in $[H]$ is 0, and the remaining bits form the same sequence as the number N in $[F]$. Multiplication by two takes one derivation step.

$$\boxed{\begin{array}{l} A \rightarrow\ B \\ \quad x_1\text{N V} \doteq 0 \\ \quad \wedge\, x_0\text{N} \doteq x_1\text{N } 0 \end{array}}$$

Table 2.4: The rule to multiply by two.

Syntactic rules that increments by one. Assume a nonterminal A with some AVM $\left[\ \text{N}\ \ [F]\ \right]$, where $[F]$ encodes natural number x. We present five syntactic rules that derive from this nonterminal A a nonterminal C with AVM $\left[\ \text{N}\ \ [H]\ \right]$, where $[H]$ encodes natural number $x + 1$.

The increment of N requires two additional pointers in the AVM of A: attribute P points to the next bit that has to be incremented; attribute Q points to the most-significant bit of the (intermediate) result. These additional pointers are hidden from the AVMs of the nonterminals A and C.

The five rules from Table 2.5 increment N by one. Nonterminal A rewrites, in one or more steps, to nonterminal C, potentially through a number of nonterminals B.

The first and fourth rule of Table 2.5 state that adding one to a zero bit sets this bit to one and ends the increment. The second and third rule state that adding one to a one bit sets this bit to zero and the increment continues. The fifth rule states that adding one to the most-significant bit sets this bit to zero and yields a new most-significant one bit. We claim that $A \stackrel{*}{\Rightarrow} C$ takes an amount of derivation steps that is linear with respect to the size of AVM $[F]$.

Rules, similar to the ones above, can be given that decrement the attribute N by one. We only have to take a little extra care that the number 0 cannot be decremented.

Syntactic rules that sum two numbers. In this section we use the previous test and increment rules (indicated by =). Assume a nonterminal A with some

$A' \rightarrow C'$ $x_0\mathrm{N}\ \mathrm{V} \doteq 0$ $\wedge\, x_0\mathrm{N}\ 0 \doteq x_1\mathrm{N}\ 1$ $\wedge\, x_1\mathrm{N}\ \mathrm{V} \doteq 1$	$A' \rightarrow B$ $x_0\mathrm{N}\ \mathrm{V} \doteq 1$ $\wedge\, x_0\mathrm{N}\ 1 \doteq x_1\mathrm{P}$ $\wedge\, x_1\mathrm{N}\ 0 \doteq x_1\mathrm{Q}$ $\wedge\, x_1\mathrm{N}\ \mathrm{V} \doteq 0$	$B \rightarrow B$ $x_0\mathrm{P}\ \mathrm{V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 1 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{Q}\ \mathrm{V} \doteq 0$ $\wedge\, x_0\mathrm{Q}\ 0 \doteq x_1\mathrm{Q}$
$B \rightarrow C'$ $x_0\mathrm{P}\ \mathrm{V} \doteq 0$ $\wedge\, x_0\mathrm{Q}\ \mathrm{V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 0 \doteq x_0\mathrm{Q}\ 1$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$	$B \rightarrow C'$ $x_0\mathrm{P}\ \mathrm{V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 1 \doteq +$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{Q}\ \mathrm{V} \doteq 0$ $\wedge\, x_0\mathrm{Q}\ 0\ \mathrm{V} \doteq 1$ $\wedge\, x_0\mathrm{Q}\ 0\ 1 \doteq +$	

Table 2.5: Five rules to increment N by one.

AVM $\begin{bmatrix} \mathrm{N} & [F] \\ \mathrm{M} & [H] \end{bmatrix}$, where $[F]$ and $[H]$ encode natural number x and y, respectively. We present syntactic rules (Tables 2.6–2.9) that derive from this nonterminal A a nonterminal C with AVM $\begin{bmatrix} \mathrm{N} & [F'] \\ \mathrm{M} & [H] \end{bmatrix}$, where $[F']$ encodes the natural number $x+y$.

$A \rightarrow A'$ $x_0\mathrm{M} \doteq x_1\mathrm{M}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{P}$ $\wedge\, x_1\mathrm{M} \doteq x_1\mathrm{Q}$ $\wedge\, x_1\mathrm{R} \doteq x_1\mathrm{N}$	$C' \rightarrow C$ $x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$

Table 2.6: Two rules to hide the auxiliary pointers.

The increment of N by M is similar to the increment by one. Here, three additional pointers are required: the attributes P and Q point to the bits in N and M respectively that have to be summed next; attribute R points to the most-significant bit of the (intermediate) result. In the addition two states are distinguished. In one state, the carry bit is zero, indicated by nonterminal A'. In the other state, the carry bit is one, indicated by nonterminal B. We claim that $A \overset{*}{\Rightarrow} C$ takes an amount of derivation steps that is linear with respect to the size of the largest AVM.

Syntactic rules that sum a sequence of numbers. In this section we use the previous summation rules (indicated by $=$). Assume a nonterminal A with some AVM $\begin{bmatrix} \mathrm{L} & [F'] \end{bmatrix}$, where $[F']$ encodes a list of numbers. To wit

$$[F'] = \begin{bmatrix} \mathrm{F} & [G_1] \\ \mathrm{R} & \begin{bmatrix} \mathrm{F} & [G_2] \\ \mathrm{R} & \ldots \begin{bmatrix} \mathrm{F} & [G_n] \\ \mathrm{R} & + \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$A' \rightarrow A'$ $x_0\mathrm{P\ V} \doteq 0$ $\wedge\, x_0\mathrm{Q\ V} \doteq 0$ $\wedge\, x_0\mathrm{R\ V} \doteq 0$ $\wedge\, x_0\mathrm{P}\ 0 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 0 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 0 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$	$A' \rightarrow A'$ $x_0\mathrm{P\ V} \doteq 1$ $\wedge\, x_0\mathrm{Q\ V} \doteq 0$ $\wedge\, x_0\mathrm{R\ V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 1 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 0 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 1 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$	$A' \rightarrow A'$ $x_0\mathrm{P\ V} \doteq 0$ $\wedge\, x_0\mathrm{Q\ V} \doteq 1$ $\wedge\, x_0\mathrm{R\ V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 0 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 1 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 1 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$
$B \rightarrow B$ $x_0\mathrm{P\ V} \doteq 1$ $\wedge\, x_0\mathrm{Q\ V} \doteq 0$ $\wedge\, x_0\mathrm{R\ V} \doteq 0$ $\wedge\, x_0\mathrm{P}\ 1 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 0 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 0 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$	$B \rightarrow B$ $x_0\mathrm{P\ V} \doteq 0$ $\wedge\, x_0\mathrm{Q\ V} \doteq 1$ $\wedge\, x_0\mathrm{R\ V} \doteq 0$ $\wedge\, x_0\mathrm{P}\ 0 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 1 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 0 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$	$B \rightarrow B$ $x_0\mathrm{P\ V} \doteq 1$ $\wedge\, x_0\mathrm{Q\ V} \doteq 1$ $\wedge\, x_0\mathrm{R\ V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 1 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 1 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 1 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$

Table 2.7: Rules when the carry bit is not changed.

$A' \rightarrow B$ $x_0\mathrm{P\ V} \doteq 1$ $\wedge\, x_0\mathrm{Q\ V} \doteq 1$ $\wedge\, x_0\mathrm{R\ V} \doteq 0$ $\wedge\, x_0\mathrm{P}\ 1 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 1 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 0 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$	$B \rightarrow A'$ $x_0\mathrm{P\ V} \doteq 0$ $\wedge\, x_0\mathrm{Q\ V} \doteq 0$ $\wedge\, x_0\mathrm{R\ V} \doteq 1$ $\wedge\, x_0\mathrm{P}\ 0 \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{Q}\ 0 \doteq x_1\mathrm{Q}$ $\wedge\, x_0\mathrm{R}\ 1 \doteq x_1\mathrm{R}$ $\wedge\, x_0\mathrm{N} \doteq x_1\mathrm{N}$ $\wedge\, x_0\mathrm{M} \doteq x_1\mathrm{M}$

Table 2.8: Rules when the carry bit is changed.

where $[G_i]$ encodes natural number x_i. We present syntactic rules (Table 2.10) that derive from this nonterminal A a nonterminal B with AVM $\begin{bmatrix} \text{SUML} & [F] \\ \text{L} & [F'] \end{bmatrix}$, where $[F]$ encodes the natural number $y = \Sigma_i x_i$.

The summation requires an additional pointer in the AVM $[F']$: attribute P points to the next element in the list that has to be summed. We claim that $A \overset{*}{\Rightarrow} B$ takes an amount of derivation steps that is linear with respect to the size of $[F']$.

2.8.2 Creating a counter of logarithmic size

In this section we will create AVMs of the following form, where the AVMs $[F_i]$ represent natural numbers:

$$\begin{bmatrix} \text{COUNTER} & \begin{bmatrix} \text{SIZE} & [F_1] \\ \text{N} & [F_2] \\ \text{M} & [F_3] \\ \text{POLY} & [F_4] \end{bmatrix} \end{bmatrix}.$$

$A' \rightarrow C'$ $x_0\mathrm{P} \doteq +$ $\wedge\, x_0\mathrm{Q} = i$ $\wedge\, x_0\mathrm{R} = j$ $\wedge\, i = j$ $\wedge\, x_0 \doteq x_1$	$A' \rightarrow C'$ $x_0\mathrm{P} = i$ $\wedge\, x_0\mathrm{Q} \doteq +$ $\wedge\, x_0\mathrm{R} = j$ $\wedge\, i = j$ $\wedge\, x_0 \doteq x_1$	$A' \rightarrow C'$ $x_0\mathrm{P} \doteq +$ $\wedge\, x_0\mathrm{Q} \doteq +$ $\wedge\, x_0\mathrm{R} \doteq +$ $\wedge\, x_0 \doteq x_1$
$B \rightarrow C'$ $x_0\mathrm{P} \doteq +$ $\wedge\, x_0\mathrm{Q} = z$ $\wedge\, x_0\mathrm{R} = z + 1$ $\wedge\, x_0 \doteq x_1$	$B \rightarrow C'$ $x_0\mathrm{P} = z$ $\wedge\, x_0\mathrm{Q} \doteq +$ $\wedge\, x_0\mathrm{R} = z + 1$ $\wedge\, x_0 \doteq x_1$	$B \rightarrow C'$ $x_0\mathrm{P} \doteq +$ $\wedge\, x_0\mathrm{Q} \doteq +$ $\wedge\, x_0\mathrm{R}\ \mathrm{V} \doteq 1$ $\wedge\, x_0\mathrm{R}\ 1 \doteq +$ $\wedge\, x_0 \doteq x_1$

Table 2.9: Rules that stop the summation.

$A \rightarrow A'$ $x_1\mathrm{N}\ \mathrm{V} \doteq 0$ $\wedge\, x_1\mathrm{N}\ 0 \doteq +$ $\wedge\, x_0\mathrm{L} \doteq x_1\mathrm{L}$ $\wedge\, x_0\mathrm{L} \doteq x_1\mathrm{P}$	$A' \rightarrow A'$ $x_0\mathrm{SUML} = y$ $\wedge\, x_0\mathrm{P}\ \mathrm{F} = z$ $\wedge\, x_1\mathrm{SUML} = y + z$ $\wedge\, x_0\mathrm{P}\ \mathrm{R} \doteq x_1\mathrm{P}$ $\wedge\, x_0\mathrm{L} \doteq x_1\mathrm{L}$	$A' \rightarrow B$ $x_0\mathrm{P} \doteq +$ $\wedge\, x_0\mathrm{SUML} \doteq x_1\mathrm{SUML}$ $\wedge\, x_0\mathrm{L} \doteq x_1\mathrm{L}$

Table 2.10: Three rules that sum a list of numbers.

Attribute COUNTER is used to distinguish the AVMs that encodes the counter from those in the original attribute-value grammar. We will neglect the attribute COUNTER in the remainder of this section, because it is not essential here. The attributes SIZE, N, M and POLY encode natural numbers. The attribute SIZE records the size of the string that will be generated. The attribute POLY records the maximum number of derivation steps that is allowed for a string of size SIZE. The attributes N and M are auxiliary numbers.

The construction of the counter starts with an initiation-step. The further construction of the counter consists of cycles of two phases. Each cycle starts in nonterminal A.

Initiation step and first phase. The initiation-step sets the numbers SIZE and N to 0, and the numbers M and POLY to 1. In the first phase of each cycle, the numbers SIZE and N are incremented by 1.

The second phase of the cycle. In this phase the numbers N and M are compared. If N is twice M, then (*i*) number POLY is extended by k bits, (*ii*) number M is doubled, and (*iii*) number N is set to 0. If N is less than twice M, nothing happens.

The left rule of the second phase doubles the number M in the second and the third equation. The test "Is N equal to 2M?" therefore reduces to one (the first)

$S \rightarrow A$	$A \rightarrow B$
$x_1\textsc{size v} \doteq 0$	$x_0\textsc{size} = x$
$\wedge\, x_1\textsc{size } 0 \doteq +$	$\wedge\, x_1\textsc{size} = x + 1$
$\wedge\, x_1\textsc{n v} \doteq 0$	$\wedge\, x_0\textsc{n} = y$
$\wedge\, x_1\textsc{n } 0 \doteq +$	$\wedge\, x_1\textsc{n} = y + 1$
$\wedge\, x_1\textsc{m v} \doteq 1$	$\wedge\, x_0\textsc{m} \doteq x_1\textsc{m}$
$\wedge\, x_1\textsc{m } 1 \doteq +$	$\wedge\, x_0\textsc{poly} \doteq x_1\textsc{poly}$
$\wedge\, x_1\textsc{poly v} \doteq 1$	
$\wedge\, x_1\textsc{poly } 1 \doteq +$	

Table 2.11: Initiation-step and first phase.

equation. The fourth equation extend the number POLY with k bits. The fifth and sixth equations set the number N to 0.

The right rule is always applicable. If the right rule is used where the left rule was applicable, then the number N will never be equal to 2M in the rest of the derivation. Thus POLY will not be extended any more.

$B \rightarrow A$	$B \rightarrow A$
$x_0\textsc{n} = x_1\textsc{m}$	$x_0 \doteq x_1$
$\wedge x_0\textsc{m} = x$	
$\wedge x_1\textsc{m} = 2x$	
$\wedge x_1\textsc{poly } 0^i \textsc{ v} \doteq 0 \ \ (0 \leq i < k)$	
$\wedge x_0\textsc{poly} \doteq x_1\textsc{poly } 0^k$	
$\wedge x_1\textsc{n v} \doteq 0$	
$\wedge x_1\textsc{n } 0 \doteq +$	

Table 2.12: The second phase.

We claim that the left rule appears at most $\log(n)$ times and the right rule at most n times in a derivation for input of size n. Obviously, the number POLY is at most $2^{k \log i} = i^k$ when the number SIZE is i.

2.8.3 From AVG to HP-AVG

In this section we show how to transform an AVG into an AVG that satisfies the HPC (HP-AVG). Since all computation steps of the HP-AVG only require a linear amount of derivation steps, total derivations of HP-AVGs have polynomial length.

We can divide the attributes of the HP-AVG into two groups. The attributes that encode the counters, and the attributes of the original AVG. The former will be embedded under the attribute COUNTER, the latter under the attribute GRAMMAR. In the sequel, we mean by $\phi|\textsc{grammar}$ the formula ϕ embedded under the attribute GRAMMAR, i.e., the formula obtained from ϕ by substituting the variables x_i by $x_i\textsc{grammar}$.

The HP-AVG is obtained from the AVG in three steps: change the start symbol, the lexicon and the syntactic rules. First, the HP-AVG contains the rules of the previous section, which construct the counter. The nonterminal S from Table 2.11 is the start symbol of the HP-AVG. For the nonterminal A the start symbol of the AVG is taken. Nonterminal B from Table 2.12 is a new nonterminal, not occurring in the AVG.

Second, the HP-AVG contains an extension of the lexicon of the AVG. The entries of the lexicon are extended in the following way. The size of the lexical form is set to one, and the amount of derivation steps is zero. Thus if (w, X, ϕ) is the lexicon of the AVG, then (w, X, ψ) is the lexicon of the HP-AVG, where

$$\begin{aligned} \psi &= \phi|\text{GRAMMAR} \\ &\wedge\ x_0\text{COUNTER SIZE V} \doteq 1 \\ &\wedge\ x_0\text{COUNTER SIZE } 1 \doteq + \\ &\wedge\ x_0\text{COUNTER POLY V} \doteq 0 \\ &\wedge\ x_0\text{COUNTER POLY } 0 \doteq + \end{aligned}$$

Third, the HP-AVG contains extensions of the syntactic rules of the AVG. The syntactic rules are extended in the following way. The numbers POLY and SIZE of the daughter nonterminals are collected in the lists PLIST and SLIST. Both lists are summed. The number SIZE of the mother nonterminal is equal to the sum of SIZE's, and the number POLY of the mother nonterminal is one more than the sum of POLY's. Thus if $(X_0, X_1, \ldots, X_n, \phi)$ is a syntactic rule of the AVG, then $(X_0, X_1, \ldots, X_n, \psi)$ is a syntactic rule of the HP-AVG, where

$$\begin{aligned} \psi &= \phi|\text{GRAMMAR} \\ &\wedge\ x_0\text{COUNTER SUMS} = \Sigma\, x_0\text{COUNTER SLIST} \\ &\wedge\ x_0\text{COUNTER SIZE} = x_0\text{COUNTER SUMS} \\ &\wedge\ x_0\text{COUNTER SUMP} = \Sigma\, x_0\text{COUNTER PLIST} \\ &\wedge\ x_0\text{COUNTER SUMP} = y \\ &\wedge\ x_0\text{COUNTER POLY} = y + 1 \\ &\wedge\ x_0\text{COUNTER SLIST R}^i\text{ F} \doteq x_i\text{COUNTER SIZE}\ \ (0 \leq i < n) \\ &\wedge\ x_0\text{COUNTER SLIST R}^n \doteq + \\ &\wedge\ x_0\text{COUNTER PLIST R}^i\text{ F} \doteq x_i\text{COUNTER POLY}\ \ (0 \leq i < n) \\ &\wedge\ x_0\text{COUNTER PLIST R}^n \doteq + \end{aligned}$$

Now a derivation for the HP-AVG starts with a nondeterministic construction of a counter SIZE with value n and a counter POLY with value n^k. Then the derivation of the original AVG is simulated, such that (i) the mother nonterminal produces a string of size n iff the daughter nonterminals together produce a string of size n, and (ii) the mother nonterminal makes $n^k + 1$ derivation steps iff the daughter nonterminals together make n^k derivation steps.

Chapter 3

Complexity of Categorial Unification Grammar

3.1 Introduction

In this and the three subsequent chapters, we consider the complexity of the recognition problem of unification-based grammatical theories. Especially, we consider the recognition problem of fixed grammars. We will present NP-hard lower bounds for the *fixed recognition problems* of primitive fragments of Categorial Unification Grammar (CUG), Functional Unification Grammar (FUG), Head-driven Phrase Structure Grammar (HPSG), Lexical Functional Grammar (LFG). As a consequence, the more general *universal recognition problems* of these fragments are also NP-hard.

In Section 3.3, the NP-hard lower bound for the fixed recognition problem of CUG is proven by a polynomial time, many-one reduction from the NP-complete problem 3-SATISFIABILITY (3SAT). We will show that there is a grammar G such that (i) for each formula φ there exists a string w such that G generates w iff φ is a satisfiable 3SAT formula; (ii) for each formula φ it takes polynomial time to compute the string w. In Chapter 4, 5, and 6, the NP-hard lower bounds for the remaining three fragments are proven by means of simulations of CUG. We will show that each CUG-grammar can be simulated by an FUG-, HPSG-, and LFG-grammar. Hence if G is a CUG-grammar, then there are FUG-, HPSG-, and LFG-grammars G' such that G generates string w iff G' generates string w'. Moreover, we will show that the simulations are computed in polynomial time. Hence the simulations are polynomial time many-one reductions, and thus the fixed recognition problems of FUG, HPSG and LFG are NP-hard.

Furthermore, we will show that derivations in each of the four grammatical theories have polynomial length and derivation steps are computable in polynomial time. Thus the universal recognition problems of these grammatical theories and the fixed recognition problems are NP-complete. The simulations and the complexity results also yield lower and upper bounds on the weak generative capacity of these grammatical theories.

3.2 Preliminaries

This section contains the preliminaries on complexity theory and feature theory. First we present the preliminaries on complexity theory: in particular the notions "*P*," "*NP*," "*NP*-hard," "*NP*-complete," "3-SATISFIABILITY," and "reduction" are introduced. Then we present the preliminaries on feature theory: in particular the notions "attribute," "value," "attribute-value matrix," "reentrance," and "unification" are introduced.

3.2.1 Complexity Theory

Some complexity classes. In complexity theory one tries to determine the complexity of problems. The complexity is measured by the amount of time and space needed to solve a problem. Usually, one considers *decision problems*: problems that are answered "Yes" or "No." Often we are interested in the distinction between tractable and intractable problems. A problem is *tractable* if its solution requires an amount of steps that is polynomial in the size of the input: we say that the problem requires *polynomial time*. Likewise, we speak of *linear time*, quadratic time, etcetera. The class of all tractable problems is called *P*. Occasionally we will refer to the tractable problems as *P problems*. The intractable problems are called *NP-hard problems*. The easiest intractable problems are the *NP-complete problems*. The *NP*-complete problems are contained in the complexity class *NP*. Solutions for problems in *NP* can be guessed and checked in polynomial time. It is strongly believed that the complexity class *P* and the complexity class *NP* are different, although this has not been proven. In fact this is the major open problem in complexity theory. (See any textbook on complexity theory for more formal definitions (e.g., Hopcroft and Ullman 1979, Garey and Johnson 1979)).

Reductions & 3-Satisfiability. There is a direct manner to determine the upper bound complexity of a problem, if there is an algorithm that solves the problem: determine the complexity of that algorithm. An indirect way to determine the lower bound complexity of a problem is the reduction. A reduction from some problem A to some problem B maps instances of problem A onto instances of problem B.

The reductions that we will consider are known as *polynomial time, many-one reductions*. These many-one reductions are subject to two conditions: (i) the reductions are easy to compute, and (ii) the reductions preserve the answers. A reduction from A to B is *easy* to compute, if the mapping takes polynomial time. A reduction *preserves answers* if the answer to the instance of A is the same as the answer to the instance of B. That is, the answer to the instance of A is "Yes" iff the answer to the instance of B is also "Yes."

A reduction is an elegant way to classify a problem as intractable. Suppose problem B is a problem with unknown complexity. Let there be a reduction f from an *NP*-hard problem A to problem B. Furthermore, let f conform to the two conditions above. By means of an indirect proof, it follows from this reduction that B is at least as hard as A. Hence B is also an *NP*-hard problem. If we also prove that we can guess a solution for B and check that guessed solution in polynomial time, then B is an NP-complete problem.

A well-known NP-complete problem is 3-SATISFIABILITY (3SAT).

3.2.1. DEFINITION. 3-SATISFIABILITY
INSTANCE: A formula φ, from propositional logic, in 3-conjunctive normal form.
QUESTION: Is there an assignment of truth-values to the propositional variables of φ, such that φ is true?

The instances of 3-SATISFIABILITY are formulas in *3-conjunctive normal form*, i.e., the formulas are conjunctions of *clauses*. The clauses are disjunctions of exactly three literals, and the *literals* are positive and negative occurrences of propositional variables. We call formula φ a *satisfiable* formula if an assignment exists that makes formula φ true.

An assignment assigns either the value true or the value false to each propositional variable. Given such an assignment, we can determine the truth-value of a formula. The formula $\varphi = (\gamma_1 \wedge \ldots \wedge \gamma_m)$ is true iff each clause, γ_i, is true. A clause $\gamma = (l_1 \vee \ldots \vee l_m)$ is true iff at least one literal, l_i, is true. A positive literal, $l_i = p_j$, is true iff the variable p_j is assigned the value true. A negative literal, $l_i = \overline{p_j}$, is true iff the variable p_j is assigned the value false.

Recognition problems. In this and the three subsequent chapters, we consider the complexity of the recognition problem of grammatical theories. Especially, we consider the recognition problem of fixed grammars. The recognition problem is the problem whether a string, w, is a string of the language, $L(G)$, generated by a grammar, G. There are multiple versions of the question whether a grammar generates a string. In the general case, the question concerns an arbitrary string and an arbitrary grammar. This version of the recognition problem is called the *universal recognition problem* (URP). A more specific version of this question results when one considers a fixed grammar instead of an arbitrary one. We will call this version the *fixed recognition problem* (FRP).

The universal recognition problem (URP) is defined as follows.

3.2.2. DEFINITION. Universal Recognition Problem
INSTANCE: A pair (w, G), where w a string and G a grammar.
QUESTION: Is the string w in the language generated by G ($w \in L(G)$)?

The fixed recognition problem (FRP) is defined as follows.

3.2.3. DEFINITION. Fixed Recognition Problem
Let there be a fixed grammar G.
INSTANCE: A string w.
QUESTION: Is the string w in the language generated by G ($w \in L(G)$)?

The universal recognition problem is at least as difficult as the fixed recognition problem. The universal recognition problem may be more difficult. For instance, an algorithm that solves the universal recognition problem efficiently also solves the fixed recognition problem efficiently. However, an efficient algorithm for the fixed recognition problem may take an amount of time that is exponential with respect to the size of the grammar. Clearly, such an algorithm does not solve the universal recognition problem efficiently.

3.2.2 The Standard Feature Theory

In this section we introduce a "standard" feature theory, which expresses the general principles of the four unification-based grammatical formalisms described in this thesis. This feature theory serves as the starting point in the definitions of the grammatical formalisms in the next chapters. Our objective is to make a comparison between the various formalisms easy. Therefore, we will use a consistent notations and terminology throughout this thesis. As a consequence, our notations and terminology will occasionally conflict with those usually applied by these various formalisms.

This section consists of five parts.

- In the first part of this section, we will formalize the notion of a feature-graph.
- In the second part of this section, we will present an intuitive way to describe feature-graphs: attribute-value matrices.
- In the third part of this section, we will present a more formal way to describe feature-graphs: F_L-formulas.
- In the fourth part of this section, we will formalize the notion of unification and the related notion subsumption.
- In the fifth and final part of this section, we will present two algorithms. The first algorithm, called BOX2PATH, transforms an attribute-value matrix that contains box-labels into an attribute-value matrix that contains path-equations. This algorithm takes quasi-linear time, i.e., a logarithmic factor more than linear time. The second algorithm, called F_L-SATISFIABILITY, is an efficient unification algorithm. The algorithm solves the unification problem for feature-graphs in quadratic time.

Feature-graphs. Although a universal feature theory does not exist, there is a general understanding of its objects. The objects of feature theories are abstract linguistic objects, e.g., an object "sentence," an object "masculine third person singular," an object "verb," an object "noun phrase." These abstract objects have properties, like "tense," "number," "predicate," "subject." The values of these properties are either atomic, like "present" and "singular," or abstract objects, like "verb" and "noun phrase." The abstract objects can be represented as rooted graphs: feature-graphs.

The nodes of these feature-graphs stand for abstract objects, like "singular" and "noun phrase." The edges represent properties of these objects, like "tense" and "number." To be more precise, a feature-graph is a finite, rooted, connected, directed, and acyclic graph whose edges are labeled with attributes. For every node, the labels of the edges departing from it must be pair-wise distinct. A node represents an atomic abstract object only if no edges depart from that node.

3.2.4. DEFINITION. A *feature-graph* is either

1. a pair $(a, \emptyset)$, where a is an atomic value and $\emptyset$ is the empty set, or
2. a pair (x, E), where x is a root node and E is a finite, possibly empty set of edges such that
   (a) for each property and all nodes there is at most one edge that represents the property departing from the node,
   (b) if there is an edge in E from node y to node z, then node y is a non-atomic value and z is an atomic or non-atomic value,
   (c) if there is an edge in E from node y to node z, then there is no path in E from node z to node y, and
   (d) if there is an edge in E from node y to node z, then there is a path in E leading from the root node x to node y.

As an example consider the following three abstract objects and the simplified feature-graph in Figure 3.1.

3.2.5. EXAMPLE.

- The abstract object temptingly labeled *Sentence: A man walks* has three properties: TENSE, SUBJECT, and PREDICATE. The value of property TENSE is *present*, the value of property SUBJECT is the abstract object labeled *Noun phrase: A man*, and value of property PREDICATE is the abstract object labeled *Verb: walks*.
- The abstract object labeled *Noun phrase: A man* has property NUMBER with value *singular*.
- The abstract object labeled *Verb: walks* also has property NUMBER with value *singular*.

Note that it is not a coincidence that the abstract objects labeled by *Noun phrase: A man* and *Verb: walks* have the same value *singular* for property NUMBER. These two abstract objects have the same value for property NUMBER, because the two edges with label NUMBER lead to the same node.

The description of feature-graphs by attribute-value matrices. A familiar, intuitive way to describe feature-graphs is the attribute-value matrix notation. An *attribute-value matrix* (AVM) is a collection of attribute-value pairs. The AVMs are written as matrices between squared brackets. An *attribute* is a string of letters without internal structure. In this thesis attributes are capitalized, like SUBJECT and NUMBER. A *value* is either atomic, i.e., a string of letters without internal structure, or compound, e.g., an attribute-value matrix. Atomic values will be written in italics, like *singular*. Because values may be compound, sequences of attributes

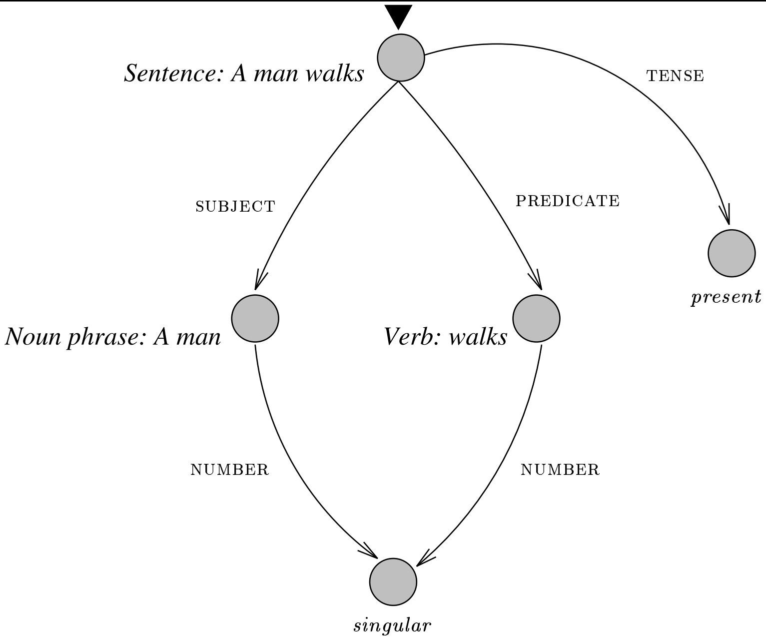


Figure 3.1: The feature-graph for *A man walks*.

exist. A sequence of attributes is called a *path*. We say that the value of the final attribute in a path is the *value of a path*. Paths are enclosed in angled brackets, like ⟨SUBJECT NUMBER⟩. These notions are illustrated by (1) in Example 3.2.6.

An extension to the basic AVMs that all formalisms have in common is *reentrance*, also known as information sharing. Reentrance enables one to state that two paths have the same value; we say that these paths are *reentrant*. Two slightly different notations (cf., Blackburn and Spaan 1993) and the next section, for reentrance are *box-labels* and *path-equations*. Box-labels are natural numbers within a frame, e.g., $\boxed{1}$. Path-equations are pairs of paths, conjoined by an dotted equality-sign, like ⟨SUBJECT⟩ $\doteq$ ⟨HEAD SUBJECT⟩. The distinction between box-labels and path-equations is illustrated by (2) in Example 3.2.6.

In this feature theory there is no essential difference in representing reentrance by box-labels or path-equations. Attribute-value matrices that use box-labels to represent reentrance are easily transformed into equivalent attribute-value matrices that use path-equations. At the end of this section we will present an efficient algorithm, BOX2PATH, that performs this transformation.

3.2.6. EXAMPLE.

1. The value of the attribute NUMBER in the attribute-value matrix

$$\left[\ \text{NUMBER} \quad \textit{singular}\ \right]$$

is the atomic value *singular*.
The compound value of the attribute SUBJECT in the attribute-value matrix

$$\left[\ \text{SUBJECT} \quad [\text{NUMBER}\ \textit{singular}]\ \right]$$

is the attribute-value matrix

$$\left[\ \textsc{number}\ \textit{singular}\ \right].$$

We say that the value of the path ⟨SUBJECT NUMBER⟩ is *singular*.

2. The two occurrences of the box-label $\boxed{1}$ in the attribute-value matrix

$$\begin{bmatrix} \textsc{subject} & \boxed{1}\begin{bmatrix} \textsc{number} & \textit{singular} \\ \textsc{person} & \textit{3rd} \end{bmatrix} \\ \textsc{head} & \begin{bmatrix} \textsc{subject} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

denote the reentrance of the paths ⟨SUBJECT⟩ and ⟨HEAD SUBJECT⟩. The path-equation in the following attribute-value matrix expresses also that the paths ⟨SUBJECT⟩ and ⟨HEAD SUBJECT⟩ are reentrant.

$$\begin{bmatrix} \textsc{subject} & \begin{bmatrix} \textsc{number} & \textit{singular} \\ \textsc{person} & \textit{3rd} \end{bmatrix} \\ \langle \textsc{subject} \rangle & \doteq \langle \textsc{head subject} \rangle \end{bmatrix}$$

Both notations state that the value of the attribute SUBJECT is the same as the value of the sequence of the attributes HEAD and SUBJECT.

The AVM notation is intuitive because AVMs strongly resemble feature-graphs. We can view the opening brackets and the atomic values of an AVM as nodes. The outermost bracket is the root node. The attributes of the AVM can be view as edges with the attribute as their label. The box-labels and path-equations identify nodes in the feature-graph. The feature-graph given in Figure 3.1 could be represented by the following two attribute-value matrices.

$$\begin{bmatrix} \textsc{subject} & \begin{bmatrix} \textsc{number} & \boxed{1}\,\textit{singular} \end{bmatrix} \\ \textsc{predicate} & \begin{bmatrix} \textsc{number} & \boxed{1} \end{bmatrix} \\ \textsc{tense} & \textit{present} \end{bmatrix} \quad \begin{bmatrix} \textsc{subject} & \begin{bmatrix} \textsc{number} & \textit{singular} \end{bmatrix} \\ \textsc{tense} & \textit{present} \\ \multicolumn{2}{l}{\langle \textsc{subject number} \rangle \doteq \langle \textsc{predicate number} \rangle} \end{bmatrix}$$

The description of feature-graphs by F_l-formulas. The abstract linguistic objects are fully described by their properties and their values. Thus another way to describe feature-graphs is a language for these properties and values of the abstract linguistic objects. In this thesis we use a redefinition (Rounds To appear) of Smolka's (1992) sublanguage of predicate logic with equality: F_L.

Assume three pair-wise disjunct sets of symbols: the set of constants A, the set of variables V, and the set of attributes L. The *attributes* (denoted by f, g, h or capitalized strings) correspond to the properties of the abstract objects, the *variables* (denoted by x, y, z) correspond to the abstract objects, and the *constants* (denoted by a, b, c or italicized strings) correspond to the atomic values. Let s, t denote variables or constants, and let a *path* (denoted by p, q) be a finite, possible empty sequence of attributes.

3.2.7. DEFINITION. The *terms* of the *description language* F_L are the elements from V and A. The *formulas* of the description language (F_L-formulas) are equations, and conjunctions:

$$sp \doteq tq \text{ and } \varphi \wedge \psi$$

if φ, ψ are formulas, p, q are paths, and s, t are terms. The formulas of the following form are called *primitive* formulas:

$$s \doteq t \text{ and } sf \doteq t.$$

We will interpret the formulas from the description language F_L as feature-graphs. The formula $s \doteq t$ is interpreted as: the terms s and t denote the same node in the feature-graph. The formula $sf \doteq t$ is interpreted as: there is an edge with label f from the node denoted by s to the node denoted by t in the feature-graph.

As an example, consider the feature-graph given in Figure 3.1. The following formula describes the feature-graph, provided that the proper sets A, V and L are given.

$$x\,\text{SUBJECT} \doteq y \;\wedge\; x\,\text{PREDICATE} \doteq z \;\wedge\; y\,\text{NUMBER} \doteq z\,\text{NUMBER} \wedge$$
$$x\,\text{SUBJECT NUMBER} \doteq \textit{singular} \;\wedge\; x\,\text{TENSE} \doteq \textit{present}$$

Subsumption and unification. Subsumption and unification are closely related. Subsumption describes a measure of the information content. The unification of two feature-graphs is the smallest feature-graph that contains all information both feature-graphs

3.2.8. DEFINITION. A feature-graph $A = (x_A, E_A)$ *subsumes* a feature-graph $B = (x_B, E_B)$ ($A \sqsubseteq B$) iff there is total function h from the set of nodes in A to the set of nodes in B, such that

1. $h(x_A) = x_B$,
2. if there is an edge labeled f from y_A to z_A in A, then there is an edge labeled f from $h(y_A)$ to $h(z_A)$ in B, and $h(y_A) \neq h(z_A)$.

Intuitively, the unification of two feature-graphs G_1 and G_2 combines all the information that is expressed by each of the feature-graphs, provided that the information is not contradictory. The unification cannot occur if the feature-graphs contain contradictory information. Two feature-graphs G_1 and G_2 contain contradicting information in three cases:

1. if the value of some path p in G_1 is atomic, whereas the value of p in G_2 is compound;
2. if the values of some path p are atomic in both G_1 and G_2, but the atomic values differ;

3. if two paths p and q lead to the same node in G_1, whereas in G_2 p lead to a node v, q leads to node w, and from node v there is a non-empty path p' that leads to the node w.

Formally, the unification of feature-graphs is defined as follows.

3.2.9. DEFINITION. A feature-graph C is the *unification* of feature-graph A and feature-graph B ($C = A \sqcup B$) iff

1. $A \sqsubseteq C$ and $B \sqsubseteq C$, and
2. for all C' such that $A \sqsubseteq C'$ and $B \sqsubseteq C'$, $C \sqsubseteq C'$.

The following examples may illustrate the unification-operations. We use the description of feature-graphs by means of attribute-value matrices because the attribute-value matrices strongly resemble feature-graphs.

3.2.10. EXAMPLE.

1. Unifying the non-contradicting attribute-value matrices $[\text{NUMBER}\,\textit{singular}]$ and $[\text{PERSON}\ \textit{3rd}]$ yields

$$\begin{bmatrix} \text{NUMBER} & \textit{singular} \\ \text{PERSON} & \textit{3rd} \end{bmatrix}.$$

2. The value of path $\langle\text{SUBJECT}\rangle$ in the attribute-value matrix $\begin{bmatrix} \text{SUBJECT}\ \textit{John} \end{bmatrix}$ is atomic, whereas the value of path $\langle\text{SUBJECT}\rangle$ in the attribute-value matrix

$$\begin{bmatrix} \text{SUBJECT} & \begin{bmatrix} \text{NUMBER}\ \textit{singular} \end{bmatrix} \end{bmatrix}$$

is compound. Hence these two attribute-value matrices do not unify.
3. The attribute-value matrices $\begin{bmatrix} \text{SUBJECT}\ \textit{singular} \end{bmatrix}$ and $\begin{bmatrix} \text{SUBJECT}\ \textit{John} \end{bmatrix}$ do not unify, because the atomic values *singular* and *John* differ.
4. According to the attribute-value matrix

$$\begin{bmatrix} \text{SUBJECT} & \boxed{1}\begin{bmatrix} \text{PERSON} & \textit{3rd} \end{bmatrix} \\ \text{HEAD} & \boxed{1} \end{bmatrix}$$

the paths $\langle\text{SUBJECT}\rangle$ and $\langle\text{HEAD}\rangle$ are reentrant, whereas in the attribute-value matrix

$$\begin{bmatrix} \text{SUBJECT} & \boxed{2} \\ \text{HEAD} & \begin{bmatrix} \text{SUBJECT} & \boxed{2} \end{bmatrix} \end{bmatrix}$$

the path $\langle\text{SUBJECT}\rangle$ is reentrant with the path $\langle\text{HEAD SUBJECT}\rangle$. Consequently, these attribute-value matrices do not unify.

Unification in F_L is efficient. Let A and B be abstract linguistic objects, or feature-graphs, that are described by the F_L-formulas φ and ψ, respectively. The unification of A and B is described by F_L-formula $\varphi \wedge \psi$ iff $\varphi \wedge \psi$ describes a feature-graph. In the final part of this section we will present an efficient algorithm, called F_L-SATISFIABILITY, that determines whether an F_L-formula describes a feature-graph. Hence we can view the algorithm as a unification algorithm.

Unification of AVMs is efficient. Let A and B be abstract linguistic objects, or feature-graphs, that are described by the AVMs $[F]$ and $[H]$, respectively. The unification of A and B is denoted by $[F] \sqcup [H]$. The algorithm, F_L-SATISFIABILITY, in the final part of this section can be used to compute the AVM $[F] \sqcup [H]$ efficiently, in the following way.

First, there is a linear time algorithm that transforms AVMs into F_L formulas. Second, the algorithm F_L-SATISFIABILITY can easily be modified such that it also outputs the feature-graph that is described by an F_L-formula. Since the modified algorithm will remain efficient, the feature-graph will be small. Finally there is a trivial, linear time, algorithm that transforms feature-graphs into AVMs.

Two algorithms. In this final part of the section, we will present two algorithms. The first algorithm, called BOX2PATH, transforms an attribute-value matrix that contains box-labels into an attribute-value matrix that contains path-equations. This algorithm takes quasi-linear time. The second algorithm, called F_L-SATISFIABILITY, is an efficient unification algorithm. The algorithm solves the unification problem for feature-graphs in an quadratic time.

Algorithm BOX2PATH. In the standard feature theory there is no essential difference in representing reentrance by box-labels or path-equations. Attribute-value matrices that use box-labels to represent reentrance are easily transformed into equivalent attribute-value matrices that use path-equations. The following algorithm presents a quasi-linear, i.e., a logarithmic factor more than linear time, transformation from AVM $[F]$ into AVM $[F]^\circ$.

By inspection of the algorithm, it is clear that $[F]$ and $[F]^\circ$ describe exactly the same feature-graphs. The path-equations in $[F]^\circ$ are collected in the outermost attribute-value matrix. The non-linearity is due to the sorting of the table TABLE.

ALGORITHM BOX2PATH
INPUT: AVM $[F]$ containing box-labels
OUTPUT: AVM $[F]^\circ$ containing path-equations
Comment: Create a sorted table of (box-label, path)-pairs.
  Set READY := false.
  Set TABLE := empty.
  Set PATH := empty.
  Do until READY
    Do while there is an unmarked attribute f departing from PATH in $[F]$
      Mark f.

Push f onto PATH.
If f has box-label $\boxed{\text{i}}$ as value,
then add pair (i, PATH) to TABLE before all pairs with first coordinate $j > i$.
End while
If PATH = empty,
then set READY := true
else pop PATH.
End until
Comment: Remove the box-labels from $[F]^\circ$.
Set $[F]^\circ := [F]$.
Set READY := false.
Set PATH := empty.
Do until READY
Do while there is an unmarked attribute f departing from PATH in $[F]^\circ$
Mark f.
Push f onto PATH.
End while
If PATH = empty,
then set READY := true, **else**
if PATH has only box-label $\boxed{\text{i}}$ as value,
then remove top of PATH and box-label $\boxed{\text{i}}$ from $[F]^\circ$, **else**
if PATH has box-label $\boxed{\text{i}}$ plus an atomic or non-atomic value,
then remove box-label $\boxed{\text{i}}$ from $[F]^\circ$.
pop PATH.
End until
Comment: Add to $[F]^\circ$ the path-equations indicated by TABLE.
For first to last entry of TABLE **do**
if (i, p) is the first entry in TABLE with coordinate i,
then for all other entries (i, q) **do**
add path-equation $p \doteq q$ to $[F]^\circ$.

Algorithm F_L-SATISFIABILITY. In the remainder of this section we will present the algorithm F_L-SATISFIABILITY, which determines whether a formula of the description language F_L describes a feature-graph. The algorithm is a slight modification of the constraint-solving algorithm in (Smolka 1992, Section 5).

The algorithm F_L-SATISFIABILITY can be used to determine whether two abstract objects can be unified: if the formulas φ and ψ describe abstract objects, then $\varphi \wedge \psi$ describes their unification iff the unification exists. So we may say that the algorithm solves the unification problem.

The algorithm F_L-SATISFIABILITY below determines syntactically whether a formula is satisfiable in some feature algebra. Because there is a 1–1 correspondence between satisfiable formulas and feature-graphs, the algorithm determines whether a formula describes a feature-graph. The algorithm first transforms any formula by means of syntactic simplification rules into a normal form. Then this normal form is checked syntactically in order to see whether the formula is satisfiable.

The correctness and the complexity of the algorithm F_L-SATISFIABILITY follow from (Smolka 1992, Section 5). The function TRANSFORM, the procedure SIMPLIFY, the clash-freeness test and the acyclicity test can all be computed in an amount of time that is quadratic in the size of the formula φ. Hence the algorithm F_L-SATISFIABILITY takes quadratic time. Moreover, the function TRANSFORM, the procedure SIMPLIFY, the two tests use an amount of space that is linear in the size of the formula φ. Hence the algorithm F_L-SATISFIABILITY takes linear space.

ALGORITHM F_L-SATISFIABILITY
INPUT: Formula $\varphi = \bigwedge_i \varphi_i$ from the description language F_L.
OUTPUT: 1) "Yes" if φ describes an acyclic feature-graph, and
2) "No" otherwise.
Begin Algorithm
Each φ_i is of the form $sp \doteq tq$, where p, q are paths, s, t are terms.
TRANSFORM φ into a set of primitive formulas:
$P = \{\psi_i \mid \psi_i = sf \doteq t, \text{ or } \psi_i = s \doteq t\}$.
SIMPLIFY the set P, yielding set S, until no further simplification is possible.
If set S is clash-free and acyclic,
then
Exit with answer "Yes,"
else
Exit with answer "No."
End Algorithm

FUNCTION TRANSFORM
INPUT: Formula $\varphi = \bigwedge_i \varphi_i$ from the description language F_L.
OUTPUT: A set of primitive formulas $P = \{\psi_i | \psi_i = sf \doteq t, \text{ or } \psi_i = s \doteq t\}$.
Begin Function
P := TRANSFORM(φ), where
Step 0. TRANSFORM($\bigwedge_i \varphi_i$) := $\bigcup_i$ TRANSFORM(φ_i)
Step 1. TRANSFORM($sp \doteq tq$) := TRANSFORM($sp \doteq y$)$\cup$ TRANSFORM($tq \doteq y$), where y is a fresh variable
Step 2. TRANSFORM($s\, f_1 \ldots f_n \doteq y$) := $\{s \doteq y_0, y_n \doteq y\} \cup \{y_{i-1} f_i \doteq y_i \mid 1 \leq i \leq n\}$, where y_i $(1 \leq i \leq n)$ are fresh variables, and y is a variable introduced in step 1.
End Function

In the procedure SIMPLIFY we will use the following notations. We use $[x/s]P$ to denote the set that is obtained from P by replacing every occurrence of variable x by term s, and $s \doteq t \,\&\, P$ to denote the set $\{s \doteq t\} \cup P$, provided that $s \doteq t \notin P$.

PROCEDURE SIMPLIFY (cf., Smolka 1992)
INPUT: Set of primitive formulas P.

OUTPUT: Simplified set of primitive formulas S.
Begin Procedure
  Do while one of the following four simplification rules is applicable
    1. $(x \doteq s) \,\&\, P \;\rightarrow\; (x \doteq s) \,\&\, [x/s]P$ if x occurs in P and $x \neq s$
    2. $(a \doteq x) \,\&\, P \;\rightarrow\; (x \doteq a) \,\&\, P$
    3. $(xf \doteq s) \,\&\, (xf \doteq t) \,\&\, P \;\rightarrow\; (xf \doteq s) \,\&\, (s \doteq t) \,\&\, P$
    4. $(s \doteq s) \,\&\, P \;\rightarrow\; P$
  End while
Exit with the simplified form of set P: set S.
End Procedure

Now it remains to show what the final test "set S is clash-free and acyclic" in the algorithm means.

3.2.11. LEMMA. *A simplified set of primitive formulas S is clash-free if*

1. *S contains no formula $af \doteq s$, and*
2. *S contains no formula $a \doteq b$ such that $a \neq b$.*

Proof. From (Smolka 1992, Proposition 5.4). □

3.2.12. LEMMA. *A simplified set of primitive formulas S is acyclic iff S does not contain a sequence of formulas $x_i f_i \doteq x_{i+1}$ and $x_n f_n \doteq x_1$ $(1 \leq i \leq n)$.*

Proof. By induction on the length of a cycle. □

3.3 Categorial Unification Grammar

In this section we will present a formulation of Categorial Unification Grammar (CUG). This formulation serves as a starting point for the chapters on Functional Unification Grammar, Head-driven Phrase Structure Grammar, and Lexical Functional Grammar. Bach (1983a, 1983b) introduced Categorial Unification Grammar. The formalism is studied further in articles like (Uszkoreit 1986) and (Bouma 1988).

In Section 3.3.1 we will present the classical Categorial Grammar and Categorial Unification Grammar. In Section 3.3.2 we will present an *NP*-hard lower bound for the fixed recognition problem of CUG. This *NP*-hard lower bound is proven by a polynomial time, many-one reduction from the *NP*-complete problem 3-SATISFIABILITY. In Section 3.3.3 we will present an *NP* upper bound for the universal recognition problem of CUG. In Section 3.3.4 we will discuss the weak generative capacity of CUG.

3.3.1 The Grammatical Formalisms

Categorial Unification Grammar (CUG) is a classical Categorial Grammar (CG) in which feature theory is incorporated. Various extensions of Categorial Grammar are studied, e.g., Lambek Categorial Grammar (Lambek 1988, Moortgat 1988), Zielonka's Categorial Grammar (Zielonka 1981), but we will not consider them in this chapter. First we will define the classical Categorial Grammar. Then we will present the Categorial Unification Grammar.

Classical Categorial Grammar

Classical Categorial Grammar (Buszkowski 1988a, Buszkowski 1988b, Partee, ter Meulen and Wall 1990) is defined by a finite set of primitive categories, which contains one distinguished primitive category, a finite set of strings, a lexicon, and combinatory rules to combine these categories.

Thus we define a *CG-grammar* as a tuple $<\textsc{pcat}, \mathrm{S}, \mathrm{Lex}, \mathrm{Lexicon}, C>$, where PCAT is a finite set of primitive categories, S is a distinguished primitive category, Lex is a finite set of strings, Lexicon is a lexicon, and C is a set of combinatory rules.

Categories. An infinite set of categories is defined inductively from the set of primitive categories.

3.3.1. DEFINITION. Given a set of primitive categories, PCAT, the set of *categories*, CAT, is the smallest set such that

1. PCAT $\subset$ CAT.
2. if $\mathrm{X} \in$ CAT and $\mathrm{Y} \in$ CAT, then both $(\mathrm{X}\backslash\mathrm{Y}) \in$ CAT and $(\mathrm{Y}/\mathrm{X}) \in$ CAT.

The outermost parentheses of a category are omitted, when it causes no confusion.

Lexicon. The lexicon is a relational mapping from strings to categories. Such a mapping from a string to a category is called an *entry* of the lexicon. An entry for string w and category X is usually denoted by $w : \mathrm{X}$.

3.3.2. DEFINITION. Given a finite set of strings, Lex, and a finite set of primitive categories, PCAT. Let CAT be the set of categories defined from PCAT. The *lexicon*, Lexicon, is a finite subset of Lex $\times$ CAT.

Combinatory rules. The combinatory rules combine categories with each other to form a new category. The set of *combinatory rules*, C, in CG consists of two *application rules*:

$$\begin{array}{rll} \mathrm{X} \quad (\mathrm{X}\backslash\mathrm{Y}) & \rightarrow \mathrm{Y} & \text{(left application)} \\ (\mathrm{Y}/\mathrm{X}) \quad \mathrm{X} \qquad & \rightarrow \mathrm{Y} & \text{(right application)} \end{array}$$

The category X in the application rules is called the *argument*. The other category, $(\mathrm{X}\backslash\mathrm{Y})$ or (Y/X), is called the *functor*. We will refer to X in the functors $(\mathrm{X}\backslash\mathrm{Y})$ and (Y/X) as the *argument category*, and to Y as the *resulting category*.

It is clear that the category that results from the application of these combinatory rules is equal to the resulting category of the functor. This property will be generalized into the subcategory-property (Definition 3.3.6).

The language generated. Now that we know what a CG-grammar is, let us define the language that it generates. In order to define the language that a CG-grammar G generates, we introduce the notions "derive," "produce," and "generate."

3.3.3. DEFINITION. A category A_0 *derives* a sequence of categories $A_1 \dots A_n$ iff

1. $n = 2$, and A_1 and A_2 combine into A_0 by a left or right application rule ($A_1A_2 \rightarrow A_0$), or
2. A_i and A_{i+1} combine into B by a left or right application rule ($A_iA_{i+1} \rightarrow B$), and A_0 derives the sequence of categories $A_1 \dots A_{i-1}BA_{i+2} \dots A_n$.

3.3.4. DEFINITION. Given a CG-grammar G. A category A_0 *produces* a string $w = w_1 \dots w_n$ iff category A_0 derives a sequence of categories $A_1 \dots A_n$, and the lexicon of G contains the entries $w_i : A_i$ $(1 \leq i \leq n)$.

3.3.5. DEFINITION. CG-grammar G *generates* string w iff the distinguished category S produces w. The *language* that grammar G generates ($L(G)$) is defined as the set of all strings that are produced by S ($L(G) = \{w \mid \text{S produces } w\}$).

A computationally attractive property of CG is the subcategory-property. It follows easily from the combinatory rules that the subcategory-property holds for classical Categorial Grammars.

3.3.6. DEFINITION. Let PCAT be a finite set of primitive categories, and CAT be the set of categories defined from PCAT. The *subcategories* of a category A are defined as:

- If A $\in$ PCAT, then A has no subcategories.
- If A $=$ (X\Y) $\in$ CAT, or A $=$ (Y/X) $\in$ CAT, then the subcategories of A are the category Y plus all the subcategories of this category Y.

The *subcategory-property* states that if category A_0 derives a sequence of categories $A_1 \dots A_n$, then A_0 is a subcategory of one of the A_i's.

From the subcategory-property the following fact follows. This fact shows that it suffices to describe a CG-grammar by its lexicon, its distinguished primitive category, and its combinatory rules.

3.3.7. FACT. Given a lexicon Lexicon and a string $w = w_1 \dots w_n$. All categories A that appear in a derivation for w, are subcategories of some A_i $(1 \leq i \leq n)$, such that $w_i : A_i$ is an entry of Lexicon.

Categorial Unification Grammar

As mentioned above, Categorial Unification Grammar (CUG) is a Categorial Grammar in a which feature theory is incorporated. This is accomplished by associating attribute-value matrices (AVMs) with the categories. The feature theory that we will incorporate is the standard feature theory from Section 3.2. In Bouma's (1988) definition of CUG, AVMs are allowed to range over both primitive and non-primitive categories. We adopt this idea and copy the positioning of the AVM of the non-primitive category: it is positioned under the slash connectives. CUG is now obtained from CG by altering the set of categories, the lexicon, and the combinatory rules.

Categories in CUG. The next definition shows that the AVM of the resulting category of the non-primitive category is positioned under the slash connectives.

3.3.8. DEFINITION. Given the set of primitive categories in classical Categorial Grammar, PCAT, and the standard feature theory, $\mathcal{F}$. Let $[F], [H]$ be, possibly empty, AVMs from $\mathcal{F}$. The set of *CUG-categories*, CUGCAT, is defined as the smallest set such that

1. if X is a primitive category ($\mathrm{X} \in$ PCAT), then $\mathrm{X}[F]$ is a primitive CUG-category ($\mathrm{X}[F] \in$ CUGCAT);
2. $(\mathrm{X}[F] \underset{[H]}{\backslash} \mathrm{Y}), (\mathrm{Y} \underset{[H]}{/} \mathrm{X}[F])$ are CUG-categories ($(\mathrm{X}[F] \underset{[H]}{\backslash} \mathrm{Y}), (\mathrm{Y} \underset{[H]}{/} \mathrm{X}[F])$ $\in$ CUGCAT) if $\mathrm{X}[F]$ and $\mathrm{Y}[H]$ are CUG-categories ($\mathrm{X}[F], \mathrm{Y}[H] \in$ CUGCAT).

3.3.9. DEFINITION. Given two CUG-categories $X[F]$ and $Y[H]$. The CUG-categories $X[F]$ and $Y[H]$ *unify* iff

1. $X = Y$, where $X, Y \in$ PCAT and $[F]$ unifies with $[H]$, or
2. $X[F] = (U'[F'] \underset{[F]}{\backslash} U)$ and $Y[H] = (V'[H'] \underset{[H]}{\backslash} V)$, and $U'[F]$ unifies with $V'[H]$ while $U[F]$ unifies with $V[H]$, or
3. $X[F] = (U \underset{[F]}{/} U'[F'])$ and $Y[H] = (V \underset{[H]}{/} V'[H'])$, and $U[F]$ unifies with $V[H]$ while $U'[F]$ unifies with $V'[H]$.

Lexicon in CUG. Again the lexicon in CUG is a relational mapping from strings to categories. The only difference with the lexicon in CG is that the lexicon in CUG is a mapping to CUG-categories. As before, an *entry* for string w and CUG-category $\mathrm{X}\,[H]$ is usually denoted by $w : \mathrm{X}\,[H]$.

3.3.10. DEFINITION. Given a finite set of strings, Lex, and a set of CUG-categories CUGCAT. A *lexicon* is a finite subset of Lex $\times$ CUGCAT.

Combinatory rules in CUG. The application rules of CG are adapted for the CUG-categories. The *left* and *right application rules* of CUG become

$$\mathrm{X}[F] \qquad (\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}) \;\rightarrow\; \mathrm{Y}[H]$$

$$(\mathrm{V} \underset{[H']}{/} \mathrm{U}[F']) \qquad \mathrm{X}[F] \;\rightarrow\; \mathrm{Y}[H]$$

Provided that $X[F]$ unifies with $U[F']$.

The unification of X$[F]$ and U$[F']$ may effect the CUG-category V$[H']$: changing V$[H']$ into Y$[H]$. When V$[H']$ is changed into Y$[H]$, this change is caused by reentrances between U$[F']$ and V$[H']$.

We can now describe a *CUG-grammar* by its lexicon, the set of two application rules, C, and a distinguished primitive CUG-category without an AVM, S: $<\mathrm{Lexicon}, C, \mathrm{S}>$,

The language generated. In order to define the language generated by a CUG-grammar, we define the notions "derive," "produce," and "generate." These notions are very similar for CUG and CG.

3.3.11. Definition. A category A_0 with AVM $[H]$ *derives* a sequence of categories $\mathrm{A}_1 \ldots \mathrm{A}_n$ with AVMs $[F_1], \ldots, [F_n]$, respectively, iff

1. $n = 2$ and the application of $\mathrm{A}_1[F_1]$, and $\mathrm{A}_2[F_2]$, yields $\mathrm{A}_0[H]$, or
2. the application of $\mathrm{A}_i\,[F_i]$ and $\mathrm{A}_{i+1}\,[F_{i+1}]$ yields $\mathrm{B}[H']$, and $\mathrm{A}_0\,[H]$ derives the sequence of categories $\mathrm{A}_1 \ldots \mathrm{A}_{i-1}\mathrm{B}\mathrm{A}_{i+2} \ldots \mathrm{A}_n$ with AVMs $[F_1], \ldots, [F_{i-1}]$, $[H'], [F_{i+2}], \ldots, [F_n]$, respectively.

3.3.12. Definition. Given a CUG-grammar G. A category A_0 with AVM $[H]$ *produces* a string $w = w_1 \ldots w_n$ iff category A_0 with AVM $[H]$ derives a sequence of categories $\mathrm{A}_1 \ldots \mathrm{A}_n$, with AVMs $[F_1] \ldots [F_n]$, and the lexicon of G contains the entries $w_i :\ \mathrm{A}_i[H_i]$, where $[H_i] \sqsubseteq [F_i]$.

3.3.13. Definition. CUG-grammar G *generates* string w iff the distinguished category S with some AVM $[F]$ produces w. The *language* that grammar G generates $(L(G))$ is defined as the set of all strings that are produced by S with some AVM $[F]$ $(L(G) = \{w \mid \mathrm{S}\,[F] \text{ produces } w\})$.

If we disregard that CUG-categories contain AVMs, then the subcategory-property also holds for CUG.

3.3.14. Definition. The *subcategory-property* for CUG states that if CUG-category $\mathrm{A}_0\,[F_0]$ derives a sequence of CUG-categories $\mathrm{A}_1\,[F_1]\ldots\mathrm{A}_n\,[F_n]$, then A_0 is a subcategory of one of the A_i's.

We conclude this section with an example. The example illustrates the role unification plays in the application rules.

3.3.15. Example. Let there be a CUG-grammar with the left and right application rules, the distinguished category S and the following lexicon with five entries:

$$\text{the: } \underset{\boxed{1}[\textsc{def}\ +]}{\mathrm{N}} \;/\; \mathrm{N}\boxed{1} \qquad \text{a: } \underset{\boxed{1}[\textsc{def}\ -]}{\mathrm{N}} \;/\; \mathrm{N}\boxed{1}[\textsc{num}\ \textit{sing}] \qquad \text{kisses: } (\mathrm{N}\boxed{1}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \underset{\boxed{1}}{\backslash} \mathrm{S})/\mathrm{N}$$

$$\text{boy: } \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \qquad \text{girl: } \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix}$$

A CUG-category N, with some appropriate AVM, produces the string "the boy" from the entries for "the" and "boy." This production follows from applying the right application rule to the corresponding categories. The box-label $\boxed{1}$ of the argument category and the AVM of the argument can be unified successfully, yielding

$$\underset{\boxed{1}[\textsc{def}\ +]}{\mathrm{N}} \;/\; \mathrm{N}\boxed{1} \qquad \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \rightarrow \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\\ \textsc{def} & +\end{bmatrix}$$

In a similar way, a CUG-category N, with some appropriate AVM, produces the string "a girl", because the value of attribute NUM in the entry for "girl" is indeed *sing*.

$$\underset{\boxed{1}[\textsc{def}\ -]}{\mathrm{N}} \;/\; \mathrm{N}\boxed{1}[\textsc{num}\ \textit{sing}] \qquad \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \rightarrow \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\\ \textsc{def} & -\end{bmatrix}$$

The right application of the CUG-category for the lexicon entry "kisses" and the CUG-category for the string "a girl" is successful simply because the string "a girl" is a N.

$$(\mathrm{N}\boxed{1}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \underset{\boxed{1}}{\backslash} \mathrm{S})/\mathrm{N} \qquad \mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\\ \textsc{def} & -\end{bmatrix} \rightarrow \mathrm{N}\boxed{1}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \underset{\boxed{1}}{\backslash} \mathrm{S}$$

Let us now consider the left application of the CUG-categories for the string "the boy" and the string "kisses a girl." The result is the distinguished CUG-category S, with some AVM. Hence the string "the boy kisses a girl" is a sentence from the language generated by G.

$$\mathrm{N}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\\ \textsc{def} & +\end{bmatrix} \qquad \mathrm{N}\boxed{1}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \underset{\boxed{1}}{\backslash} \mathrm{S} \rightarrow \mathrm{S}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\\ \textsc{def} & +\end{bmatrix}$$

3.3.2 Fixed Recognition is *NP*-hard

In this section we will prove that the fixed recognition problem of CUG, "FRP-CUG," is *NP*-hard, cf., Definition 3.2.3. As an immediate consequence, the universal recognition problem of CUG is also *NP*-hard, a result that was obtained before in (Trautwein 1992). We will present a reduction from 3-SATISFIABILITY (3SAT) to the recognition problem of some fixed CUG-grammar G. The reduction maps formulas φ in 3-conjunctive normal form to strings w. We will prove that φ is satisfiable iff G generates w (Lemmas 3.3.20 and 3.3.21). Hence we can solve the 3SAT problem by solving the FRP-CUG problem. Furthermore, we will show that the reduction from 3SAT to FRP-CUG takes polynomial time (Lemma 3.3.16). Hence a deterministic polynomial time solution for FRP-CUG would imply a deterministic polynomial time solution for 3SAT, which proves the *NP*-hardness of the fixed and universal recognition problems of CUG (Theorems 3.3.22 and 3.3.23).

The Reduction and the Grammar

In this part we will first present the reduction and then the fixed grammar G. The reduction maps formulas, φ, in 3-conjunctive normal form onto strings, w, in the following way.

Let formula φ be a formula in 3-conjunctive normal form: $\varphi = \gamma_1 \wedge \ldots \wedge \gamma_n$, where $\gamma_i = (l_1^i \vee l_2^i \vee l_3^i)$, and l_k^i is p_j or $\overline{p_j}$. We may assume, without loss of generality, that the indices of the variables, p_j, are in binary representation. String w is obtained from φ by removing all parentheses, marking the even occurrences of the symbol $\vee$ with a prime, and preceding the literals by their indices.

Thus the reduction is defined by the following mapping from formulas in 3-conjunctive normal form onto strings:

$$\begin{array}{rcll}
f(\gamma_1 \wedge \ldots \wedge \gamma_n) & = & f(\gamma_1) \wedge \ldots \wedge f(\gamma_n) & (\gamma_i \text{ a clause}) \\
f(l_1 \vee l_2 \vee l_3) & = & f(l_1) \vee f(l_2) \vee' f(l_3) & (l_i \text{ a literal}) \\
f(p_i) & = & i\,p & (p_i \text{ a positive literal, } i \text{ in} \\
 & & & \text{binary representation}) \\
f(\overline{p_i}) & = & i\,\overline{p} & (\overline{p_i} \text{ a negative literal, } i \text{ in} \\
 & & & \text{binary representation})
\end{array}$$

Now let us define the CUG-grammar G. The fixed CUG-grammar G is defined by its distinguished primitive category, its combinatory rules, and its lexicon. The distinguished primitive category S. The combinatory rules, the left and right application rules, were given in Section 3.3.1. The lexicon of grammar G is given in Table 3.1. The entries of the lexicon are string-category pairs. A colon separates the string and the category.

Now that we have defined the reduction and the fixed grammar we turn to the next part of this section. In this second part we will prove that the reduction from 3SAT to FRP-CUG is computed in polynomial time.

String	Category	String	Category
$\vee$:	$\mathrm{T}\boxed{1} \underset{\boxed{1}}{\backslash} (\mathrm{PT} / \mathrm{T}\boxed{1})$	$\vee'$:	$(\mathrm{PT}\boxed{1} \backslash \mathrm{S}) \underset{\boxed{1}}{/} \mathrm{T}\boxed{1}$
$\vee$:	$\mathrm{T}\boxed{1} \underset{\boxed{1}}{\backslash} (\mathrm{PT} / \mathrm{F}\boxed{1})$	$\vee'$:	$(\mathrm{PT}\boxed{1} \backslash \mathrm{S}) \underset{\boxed{1}}{/} \mathrm{F}\boxed{1}$
$\vee$:	$\mathrm{F}\boxed{1} \underset{\boxed{1}}{\backslash} (\mathrm{PT} / \mathrm{T}\boxed{1})$	$\vee'$:	$(\mathrm{PF}\boxed{1} \backslash \mathrm{S}) \underset{\boxed{1}}{/} \mathrm{T}\boxed{1}$
$\vee$:	$\mathrm{F}\boxed{1} \underset{\boxed{1}}{\backslash} (\mathrm{PF} / \mathrm{F}\boxed{1})$	$\wedge$:	$(\mathrm{S}\boxed{1} \backslash \mathrm{S}) \underset{\boxed{1}}{/} \mathrm{S}\boxed{1}$
0 :	$\mathrm{T} \underset{[0\ \boxed{1}]}{/} \mathrm{T}\boxed{1}$	1 :	$\mathrm{T} \underset{[1\ \boxed{1}]}{/} \mathrm{T}\boxed{1}$
0 :	$\mathrm{F} \underset{[0\ \boxed{1}]}{/} \mathrm{F}\boxed{1}$	1 :	$\mathrm{F} \underset{[1\ \boxed{1}]}{/} \mathrm{F}\boxed{1}$
p :	$\mathrm{T}[\,\mathrm{v}\ \ +\,]$	$\overline{p}$:	$\mathrm{T}[\,\mathrm{v}\ \ -\,]$
p :	$\mathrm{F}[\,\mathrm{v}\ \ -\,]$	$\overline{p}$:	$\mathrm{F}[\,\mathrm{v}\ \ +\,]$

Table 3.1: Lexicon of fixed grammar G.

Cost of the Reduction

It follows straightforwardly from the next lemma that the reduction takes polynomial time.

3.3.16. LEMMA. *The reduction from 3SAT to FRP-CUG is computed in linear time.*

Proof. The construction of CUG-grammar G does not depend on the formula φ because G is the same for all formulas φ. Therefore, the cost to construct CUG-grammar G is neglected. The mapping from formulas φ onto strings w causes the substantial cost of the reduction from 3SAT to FRP-CUG.

Three operations are performed to obtain string w from formula φ. First, all parentheses in φ are removed. Second, the even occurrences of the symbol $\vee$ are marked with a prime. And third, the literals are preceded by their indices. These three operations can be performed simultaneously. Thus it suffices to go through the formula φ once: w is obtained from φ in a linear amount of steps, with respect to the size of the formula. So the total reduction takes a linear amount of steps, and can be computed in linear time. $\square$

The Reduction Preserves Answers

In this part we will prove that the reduction preserves answers. That is, the reduction maps all, and only all, satisfiable 3SAT formulas onto strings that are generated by G.

The subcategory-property of CUG ensures that the following two facts hold. Combining these two fact results in Lemma 3.3.19.

3.3.17. FACT. The lexicon of the grammar G contains 16 different categories, which give rise to 16 different subcategories.

1. Category S with AVM $[H]$ derives always one of the following four series of categories:
   (a) $\mathrm{PT}[F]\ \ (\mathrm{PT}\boxed{1}\backslash \mathrm{S})\ \ /_{\boxed{1}}\ \ \mathrm{T}\boxed{1}\ \ \mathrm{T}[F']$, where $[H] = [F] \sqcup [F']$
   (b) $\mathrm{PT}[F]\ \ (\mathrm{PT}\boxed{1}\backslash \mathrm{S})\ \ /_{\boxed{1}}\ \ \mathrm{F}\boxed{1}\ \ \mathrm{F}[F']$, where $[H] = [F] \sqcup [F']$
   (c) $\mathrm{PF}[F]\ \ (\mathrm{PF}\boxed{1}\backslash \mathrm{S})\ \ /_{\boxed{1}}\ \ \mathrm{T}\boxed{1}\ \ \mathrm{T}[F']$, where $[H] = [F] \sqcup [F']$
   (d) $\mathrm{S}[F]\ \ (\mathrm{S}\boxed{1}\backslash \mathrm{S})\ \ /_{\boxed{1}}\ \ \mathrm{S}\boxed{1}\ \ \mathrm{S}[F']$, where $[H] = [F] \sqcup [F']$
2. Category PT with AVM $[H]$ derives always one of the following three series of categories:
   (a) $\mathrm{T}[F]\ \ (\mathrm{T}\boxed{1}\backslash \mathrm{PT})\ \ /_{\boxed{1}}\ \ \mathrm{T}\boxed{1}\ \ \mathrm{T}[F']$, where $[H] = [F] \sqcup [F']$
   (b) $\mathrm{F}[F]\ \ (\mathrm{F}\boxed{1}\backslash \mathrm{PT})\ \ /_{\boxed{1}}\ \ \mathrm{T}\boxed{1}\ \ \mathrm{T}[F']$, where $[H] = [F] \sqcup [F']$
   (c) $\mathrm{T}[F]\ \ (\mathrm{T}\boxed{1}\backslash \mathrm{PT})\ \ /_{\boxed{1}}\ \ \mathrm{F}\boxed{1}\ \ \mathrm{F}[F']$, where $[H] = [F] \sqcup [F']$
3. Category PF with AVM $[H]$ derives always the series of categories:
   $\mathrm{F}[F]\ \ (\mathrm{F}\boxed{1}\backslash \mathrm{PF})\ \ /_{\boxed{1}}\ \ \mathrm{F}\boxed{1}\ \ \mathrm{F}[F']$, where $[H] = [F] \sqcup [F']$

3.3.18. FACT. Let w be a string from the language described by the regular expression $(0 \cup 1)^*(p \cup \overline{p})$. This string w is produced by

1. category T with AVM $[H]$ iff
   (a) $w = b_1 \ldots b_n p$ and $[H] = \begin{bmatrix} b_1 \ldots & \begin{bmatrix} b_n & \begin{bmatrix} \mathrm{V} & + \end{bmatrix} \end{bmatrix} \ldots \end{bmatrix}$, or
   (b) $w = b_1 \ldots b_n \overline{p}$ and $[H] = \begin{bmatrix} b_1 \ldots & \begin{bmatrix} b_n & \begin{bmatrix} \mathrm{V} & - \end{bmatrix} \end{bmatrix} \ldots \end{bmatrix}$
2. category F with AVM $[H]$ iff
   (a) $w = b_1 \ldots b_n p$ and $[H] = \begin{bmatrix} b_1 \ldots & \begin{bmatrix} b_n & \begin{bmatrix} \mathrm{V} & - \end{bmatrix} \end{bmatrix} \ldots \end{bmatrix}$, or
   (b) $w = b_1 \ldots b_n \overline{p}$ and $[H] = \begin{bmatrix} b_1 \ldots & \begin{bmatrix} b_n & \begin{bmatrix} \mathrm{V} & + \end{bmatrix} \end{bmatrix} \ldots \end{bmatrix}$

3.3.19. LEMMA. *Category* S *with AVM* $[H]$ *produces string* w *iff*

1. $w = b_1 \dots b_n q \vee b'_1 \dots b'_n q' \vee' b''_1 \dots b''_n q''$ *and* $(b_i, b'_i, b''_i \in \{0,1\}, q, q', q'' \in \{p, \overline{p}\})$
   $[H] = \left[\, b_1 \dots b_n \quad [\mathrm{V}\ x] \dots \,\right] \sqcup \left[\, b'_1 \dots b'_n \quad [\mathrm{V}\ y] \dots \,\right] \sqcup \left[\, b''_1 \dots b''_n \quad [\mathrm{V}\ z] \dots \,\right]$
   for $x, y, z \in \{+, -\}$ *such that at least one of the pairs* $(x, q), (y, q'), (z, q'')$ *equals* $(+, p)$ *or* $(-, \overline{p})$, *or*
2. $w = w_1 \wedge w_2$, *where the strings* w_1, w_2 *are produced by* $\mathrm{S}[F], \mathrm{S}[F']$, *and the AVMs* $[F]$ *and* $[F']$ *do not contradict each other.*

Proof.
Proof of the first part of the lemma: combine the lexicon entries for $\vee$ and $\vee'$ with Fact 3.3.18 and 3.3.17 (1a-1b, 2), and with Fact 3.3.18 and 3.3.17 (1c, 3).
Proof of the second part of the lemma: combine Fact 3.3.17 (1d) with the lexicon entries for $\wedge$. □

The next two lemmas together prove that the reduction is answer-preserving. The first lemma shows that the reduction maps all satisfiable formulas onto strings that the fixed CUG-grammar G generates. The second lemma shows that the reduction maps only satisfiable formulas onto strings that the fixed CUG-grammar G generates.

3.3.20. LEMMA. *Given a formula φ in 3-conjunctive normal form. Let the reduction above map formula φ onto the string w. If formula φ is a satisfiable formula, then the fixed grammar G generates w.*

Proof. Formula $\varphi = \gamma_1 \wedge \dots \wedge \gamma_n$ is mapped onto string $w = w_1 \wedge \dots \wedge w_n$, where $w_i = l^i_1 \vee l^i_2 \vee' l^i_3$ and each l^i_j in the language described by $(0 \cup 1)^*(p \cup \overline{p})$.

There is an assignment g that satisfies formula φ. Let string αp be produced by category $\mathrm{T}\,[H]$ iff g assigns value true to literal p_α. From Fact 3.3.18, it follows that $[H] = \left[\, \alpha \quad \left[\, \mathrm{V} \quad + \,\right] \,\right]$ $(\alpha \in (0 \cup 1)^*)$. Likewise, let string $\alpha\overline{p}$ be produced by category $\mathrm{T}\,[H]$ iff g assigns value true to literal $\overline{p_\alpha}$, where $[H] = \left[\, \alpha \quad \left[\, \mathrm{V} \quad - \,\right] \,\right]$.

As a result, string αp is produced by category T with AVM $[H]$ iff string $\alpha\overline{p}$ is produced by category F with AVM $[H]$. Furthermore, all occurrences of strings αp and $\alpha\overline{p}$ are produced by the same categories.

Combining Fact 3.3.18 and the first part of Lemma 3.3.19 shows that category $\mathrm{S}\,[F_i]$ produces w_i, where $[F_i]$ encodes the assignment g restricted to the literals that appear in clause γ_i. The second part of Lemma 3.3.19 shows that $\mathrm{S}\,[H]$ produces w, where $[H]$ is the unification of the AVMs $[F_i]$. This unification yields an encoding of the entire assignment g. Hence category S produces string w: w is generated by G. □

3.3.21. LEMMA. *Given a formula φ in 3-conjunctive normal form. Let the reduction above map formula φ onto the string w. If the fixed grammar G generates w, then formula φ is a satisfiable formula.*

Proof. Formula $\varphi = \gamma_1 \wedge \dots \wedge \gamma_n$ is mapped onto string $w = w_1 \wedge \dots \wedge w_n$, where $w_i = l^i_1 \vee l^i_2 \vee' l^i_3$ and each l^i_j in the language described by $(0 \cup 1)^*(p \cup \overline{p})$.

String w is generated by G iff w is produced by category $\mathrm{S}\,[H]$. According to the second part of Lemma 3.3.19, the strings $w_1 \dots w_n$ are produced by the categories

$\mathrm{S}\,[F_1]\ldots\mathrm{S}\,[F_n]$ and $[H]$ is the unification of $[F_1]\ldots[F_n]$. Obviously, $[F_1]\ldots[F_n]$ do not contradict one another.

According to the first part of Lemma 3.3.19, the string w_i is produced by $\mathrm{S}\,[F_i]$. Furthermore, there is at least one substring l^i in w^i such that $(\alpha \in (0 \cup 1)^*)$

- l^i corresponds to 3SAT literal p_α and $[F_i]$ contains $\begin{bmatrix} \alpha & \begin{bmatrix} \mathrm{V} & + \end{bmatrix} \end{bmatrix}$, or
- l^i corresponds to 3SAT literal $\overline{p_\alpha}$ and $[F_i]$ contains $\begin{bmatrix} \alpha & \begin{bmatrix} \mathrm{V} & - \end{bmatrix} \end{bmatrix}$.

Clearly $[F_i]$ encodes a satisfying assignment for the clause γ_i. Hence each string w_i is produced by a category $\mathrm{S}\,[F_i]$ and $[F_i]$ encodes a satisfying assignment for the clause γ_i. Moreover, $\mathrm{S}\,[H]$ produces string $w = w_1 \wedge \ldots \wedge w_n$ and AVM $[H] = [F_1] \sqcup \ldots \sqcup [F_n]$ encodes a consistent satisfying assignment for φ. So the formula φ is satisfiable under the assignment encoded by AVM $[H]$. $\square$

3.3.22. THEOREM. *The recognition problem of CUG for a fixed grammar (FRP-CUG) is NP-hard.*

Proof. We presented a reduction f from 3SAT to the recognition problem of CUG for a fixed grammar G. According to Lemma 3.3.16, this reduction is computed in polynomial time. Furthermore, according to Lemmas 3.3.20 and 3.3.21, the reduction is answer-preserving. Hence the recognition problem of CUG for a fixed grammar is as hard as 3SAT, i.e., is *NP*-hard. $\square$

3.3.23. THEOREM. *The universal recognition problem of CUG is NP-hard.*

Proof. The universal recognition problem is at least as hard as the recognition problem for a fixed grammar. Theorem 3.3.22 shows that the recognition problem for a fixed grammar is *NP*-hard. Hence the universal recognition problem of CUG is *NP*-hard. $\square$

3.3.3 Universal Recognition is in *NP*

Theorems 3.3.22 and 3.3.23 present lower bounds on the complexity of the recognition problems of CUG. Obviously, we would also like to have an upper bound on the complexity of the recognition problems.

Now we will prove an *NP* upper bound on the complexity of the universal recognition problem of CUG. As a direct consequence, the fixed recognition problem is computable in nondeterministic polynomial time, as well. The input of the universal recognition problem is a CUG-grammar G, and a string w. As far as this section is concerned, it suffices to describe the CUG-grammar by the lexicon, the combinatory rules, and the distinguished category.

The proof of the *NP* upper bound is based on the following two observations. First, a derivation for a string w and a CUG-grammar G consists of a polynomial amount of steps. This first observation is proven in Lemma 3.3.24. Second, the reverse of a derivation step is computable in polynomial time. This second observation is proven in Lemma 3.3.25.

3.3.24. LEMMA. *A derivation for a string w and a CUG-grammar G has linear size, with respect to the sizes of w and G.*

Proof. Recall that a CUG-grammar contains two combinatory rules: the left and right application rules. Both rules combine two CUG-categories. Hence a derivation for a string of size n consists of at most a linear amount of steps, with respect to the size of the string and the size of the grammar. □

3.3.25. LEMMA. *Given two CUG-categories* $\mathrm{X}[F]$ *and* $\mathrm{Z}[H']$*, the result of the application,* $\mathrm{Y}[H]$*, is computable in polynomial time, with respect to the size of the categories* $\mathrm{X}[F]$ *and* $\mathrm{Z}[H']$*.*

Proof. Two CUG-categories $\mathrm{X}[F]$ and $\mathrm{Z}[H']$ are combined by left or right application. Hence $\mathrm{Z}[H'] = (\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ or $\mathrm{Z}[H'] = (\mathrm{V} \underset{[H']}{/} \mathrm{U}[F'])$, and $\mathrm{X}[F]$ combines with $\mathrm{Z}[H']$ to the left or to the right. The result of unifying $\mathrm{X}[F]$ with $\mathrm{U}[F']$ is computed in polynomial time using linear space, because $[F]$ and $[F']$ are feature structures from the standard feature theory (Section 3.2). Therefore, the effect of that unification on $\mathrm{V}[H']$ is computable in polynomial time. Hence the result of the application, $\mathrm{Y}[H]$, is computable in polynomial time. □

Now given a string w and a CUG-grammar G, we guess a linear amount of entries from the lexicon, and a polynomial sized sequence of rules that encode the derivation for w. The previous two lemmas then show that both guesses can be checked in polynomial time. Hence the following lemma and theorem hold.

3.3.26. LEMMA. *The universal recognition problem of CUG for string w and CUG-grammar G is computable in nondeterministic polynomial time.*

Proof. A derivation for the string w is described by a sequence of lexicon entries and a sequence of application rules.

We guess a sequence of lexicon entries of G for the string w. These entries have a total size that is linear in the size of the string and the grammar. The check that this sequence of entries forms the string w takes linear time.

Next we guess a sequence of application rules, which completes the description of the derivation for w. This sequence consists of a linear amount of rules, by Lemma 3.3.24. Lemma 3.3.25 showed that each application step is computable in polynomial time. So given the guessed entries and application rules, we can compute the total derivation in polynomial time. Given the total derivation it is easy to check that the distinguished CUG-category produces the string w. □

3.3.27. THEOREM.
(*i*) *The fixed recognition problem of CUG is NP-complete.*
(*ii*) *The universal recognition problem of CUG is NP-complete.*

Proof. Theorem 3.3.22 and Lemma 3.3.26 prove the *NP*-completeness of the fixed recognition problem of CUG and the universal recognition problem of CUG. □

3.3.4 Weak Generative Capacity

In this section we will discuss the weak generative capacity of CUG. A lower bound on the generative capacity follows from known results on the generative capacity of the classical Categorial Grammar. We will also determine an upper bound on the weak generative capacity of CUG. This upper bound results from the upper bound of the complexity of the fixed recognition problem. These two bounds on the weak generative capacity are represented in Theorem 4.4.8.

A lower bound on the weak generative capacity. We know that the languages generated by classical Categorial Grammar are exactly all context-free languages. It is evident that each classical Categorial Grammar can be simulated by a CUG-grammar. Hence the set of languages generated by a CUG-grammar, the *CUG-languages*, is at least context-free.

An upper bound on the weak generative capacity. The upper bound on the weak generative capacity of CUG results from the upper bound of the complexity of the recognition problem for a fixed CUG-grammar.

Chapter 2 connects the complexity of the recognition problem and weak generative capacity of restricted attribute-value grammars. These restricted attribute-value grammars respect a liberal variation of the so-called *off-line parsability constraint*. This constraint (Johnson 1988) is a well-known generalization of the "Definition of a Valid Derivation" from Lexical Functional Grammar (Kaplan and Bresnan 1982, p. 266). The off-line parsability constraint relates the amount of "work" done by the grammar to produce a string linearly to the number of terminal symbols produced. It is therefore a sort of honesty constraint that is common in complexity theory.

In Chapter 2 a variation on the off-line parsability constraint is introduced: the *honest parsability constraint*. This honest parsability constraint is a more liberal constraint than the off-line parsability constraint. The honest parsability constraint relates the amount of "work" done by the grammar to produce a string polynomially to the number of terminal symbols produced. For convenience, we restate the definition of the honest parsability constraint, Definition 2.5.1, below.

> *A grammar G satisfies the honest parsability constraint iff there exists a polynomial p such that for each string w in the language generated by G there exists a derivation with at most $p(|w|)$ steps.*

A nice property of the restricted attribute-value grammars that respect the honest parsability constraint is that they generate exactly all languages in the complexity class *NP*. Or, as we can restate Theorem 2.5.3:

> *Let L be a language that has an NP recognition algorithm. Then there exists a restricted attribute-value grammar G, that respects the honest parsability constraint, such that G generates the language L.*

By Theorem 3.3.27 CUG has an *NP* recognition problem. Hence any CUG-grammar can be simulated by a restricted attribute-value grammar. Thus the languages generated by restricted attribute-value grammars that respect the honest parsability constraint provide an upper bound on the weak generative capacity of CUG-grammars. A close look at the proof of Lemma 3.3.26 shows that the an application step requires linear space. Hence the fixed recognition problem of CUG is computable in nondeterministic linear space. This implies that the CUG-languages are recognized by linear bounded automata (LBAs). Due to the equivalence of LBAs and context-sensitive grammars, CUG generates only context-sensitive languages (CSLs). Hence the collection of CSLs that are recognized in nondeterministic polynomial time form an upper bound for the weak generative capacity of CUG (cf., Book 1978). Or stated differently:

3.3.28. THEOREM. *The weak generative capacity of CUG is enclosed between the context-free languages and the collection of context-sensitive languages generated by restricted attribute-value grammars that respect the honest parsability constraint.*

Chapter 4

Functional Unification Grammar

In this chapter we will present a formulation of Functional Unification Grammar (FUG). The main purpose of this chapter is to show that FUG-grammars can simulate any arbitrary CUG-grammar that was presented in Section 3.3. This simulation provides a way of comparing the different grammatical formalisms. For instance, the simulation shows that the generative capacity of FUG is at least as high as the generative capacity of CUG. Moreover, because the simulation is computable in polynomial time, it is a polynomial time many-one reduction. Therefore the recognition problem of FUG is at least as hard as the NP-complete recognition problem of CUG.

The next section (Section 4.1) contains an informal introduction in FUG and discusses which part of FUG are needed for a simulation of CUG. In Section 4.2 we present the simulation of CUG. In Section 4.3 we prove the correctness of the simulation. We conclude with Section 4.4, in which we discuss the weak generative capacity of FUG and the complexity of its recognition problem.

4.1 Introduction in FUG

This section contains an informal introduction in Functional Unification Grammar (FUG). The formalism is briefly introduced in several articles, e.g., Kay (1984, 1985), Ritchie (1984, 1985), and Shieber (1986). We will only consider the well understood basics of FUG described by these articles.

We will introduce FUG in two steps. In the first step, we compare FUG with the standard feature theory from Section 3.2. We will also indicate which parts of FUG are needed for the simulation of CUG. In the second step, we will compare FUG and CUG. This comparison will reveal some implicit notions of FUG that are important for the simulation.

4.1.1 Comparison with the Standard Feature Theory

Let us first introduce a syntactic shorthand in attribute-value matrices (AVMs) in order to avoid needlessly large descriptions of AVMs. Occasionally, we use the following shorthand in AVMs:

$$\left[\text{ATTR}_1 \mid \ldots \mid \text{ATTR}_n \quad [F]\right] \text{ stands for } \left[\text{ATTR}_1 \quad \left[\ldots \quad [\text{ATTR}_n\,[F]\,]\ldots\right]\right].$$

When we compare Functional Unification Grammar and the standard feature theory, they differ in the following three aspects.

- The AVMs in FUG are not restricted to conjunctions of attribute-value pairs, but can be any conjunctive or disjunctive combination of attribute-value pairs. This disjunctive combination of attribute-value pairs is not the only extension with respect to the AVMs of the standard feature theory. FUG distinguishes four kinds of values: the familiar atomic and compound values, and the additional "special" and "structured" values. The special values are the values *any* and *none*, which stand for any and no value. The structured values are sets and lists of, among other things, paths. These structured values are restricted to the special attributes, which are discussed in a moment. In the simulation, however, no use is made of the disjunctive AVMs, or the special values. The simulation will only use the structured values set and list. These structured values are illustrated by 1 in Example 4.1.1.
- FUG distinguishes three special attributes: LEX, PATTERN, and C-SET. The attribute LEX is restricted to the AVMs that describe entries in lexicon. The value of the attribute LEX is a string, i.e., the form of a word as it appears in the lexicon.
  The purpose of C-SET, which abbreviates CONSTITUENT-SET, and PATTERN is to identify constituents in an AVM and to state constraints on the order of their occurrence. The value of the attribute C-SET is a set of paths that lead to the constituents. The attribute C-SET, however, does not specify the order of the constituents. The value of the attribute PATTERN determines the order of the constituents. This value is a list whose members are, among other things, paths leading to an AVM. A full description of the values of PATTERN is given by Kay (1985, p. 264). For the simulation it suffices to limit the values of PATTERN to lists whose members are paths leading to an AVM. The attributes C-SET and LEX play a minor role in the simulation. The three special attributes are explained by 2 in Example 4.1.1.
- The special attributes C-SET and PATTERN govern the unification in FUG. In addition to the standard way, an AVM $[F]$ can unify with the value of some path denoted by the attribute C-SET or PATTERN of another AVM $[H]$. As a result, there are multiple ways to unify two AVMs. The various possibilities to combine three AVMs are discussed in detail in Example 4.1.1 by 3. These additional ways to unify AVMs are necessary to build arbitrary constituents. Because CUG-grammars, in general, can construct arbitrary constituents, these additional ways to unify AVMs are used by the simulation.

4.1.1. EXAMPLE.

1. The structured values list and set are both indicated by parenthesis. Their elements are separated by spaces. Thus structured value $(a\,b\,c)$ is either a list

or a set with three elements. The attribute of the value denotes whether the value is a list or a set.

2. The following AVM is a simplified description of an entry in a lexicon for the verb form "saw."

$$\begin{bmatrix} \textsc{cat} & \mathit{VP} \\ \textsc{lex} & \mathit{see} \\ \textsc{tense} & \mathit{past} \end{bmatrix}$$

According to the attribute C-SET, the following AVM of category S has two constituents. These constituents are the values this AVM has for the attributes HEAD and SUBJ. The attribute PATTERN states that the leftmost constituent of this AVM is the value this AVM has for the attribute SUBJ. Its rightmost constituent is the value this AVM has for the attribute HEAD.

$$\begin{bmatrix} \textsc{cat} & S \\ \textsc{c-set} & (\langle\textsc{head}\rangle\,\langle\textsc{subj}\rangle) \\ \textsc{pattern} & (\langle\textsc{subj}\rangle\,\langle\textsc{head}\rangle) \end{bmatrix}$$

3. Let us see what happens when the following three AVMs, $[A]$, $[B]$, and $[C]$, are unified.

$$[A] = \begin{bmatrix} \textsc{c-set} & (\langle\textsc{verb}\rangle\,\langle\textsc{subj}\rangle) \\ \textsc{pattern} & (\langle\textsc{subj}\rangle\,\langle\textsc{verb}\rangle) \\ \textsc{cat} & S \\ \textsc{subj} & \begin{bmatrix} \textsc{cat} & \mathit{NP} \end{bmatrix} \end{bmatrix}, [B] = \begin{bmatrix} \textsc{cat} & \mathit{NP} \\ \textsc{lex} & \mathit{John} \end{bmatrix}, [C] = \begin{bmatrix} \textsc{cat} & \mathit{VP} \\ \textsc{lex} & \mathit{see} \\ \textsc{tense} & \mathit{past} \end{bmatrix}$$

When we try to unify AVM $[A]$ with AVM $[B]$, there are three possibilities. First, unify $[A]$ and $[B]$ in the standard way. This fails because the AVMs $[A]$ and $[B]$ have different values for the attribute CAT. Second, we can unify $[B]$ with the value of the attribute SUBJ in $[A]$, resulting in $[B^s]$. Third, unify $[B]$ with the value of the attribute VERB, resulting in $[B^v]$. Similarly, there are three ways to unify $[C]$ with $[A]$ of which one succeeds: $[C^v]$.

$$[B^s] = \begin{bmatrix} \textsc{c-set} & (\langle\textsc{verb}\rangle\,\langle\textsc{subj}\rangle) \\ \textsc{pattern} & (\langle\textsc{subj}\rangle\,\langle\textsc{verb}\rangle) \\ \textsc{cat} & S \\ \textsc{subj} & \begin{bmatrix} \textsc{cat} & \mathit{NP} \\ \textsc{lex} & \mathit{John} \end{bmatrix} \end{bmatrix}, [B^v] = \begin{bmatrix} \textsc{c-set} & (\langle\textsc{verb}\rangle\,\langle\textsc{subj}\rangle) \\ \textsc{pattern} & (\langle\textsc{subj}\rangle\,\langle\textsc{verb}\rangle) \\ \textsc{cat} & S \\ \textsc{subj} & \begin{bmatrix} \textsc{cat} & \mathit{NP} \end{bmatrix} \\ \textsc{verb} & \begin{bmatrix} \textsc{cat} & \mathit{NP} \\ \textsc{lex} & \mathit{John} \end{bmatrix} \end{bmatrix}$$

$$[C^v] = \begin{bmatrix} \textsc{c-set} & (\langle\textsc{verb}\rangle\,\langle\textsc{subj}\rangle) \\ \textsc{pattern} & (\langle\textsc{subj}\rangle\,\langle\textsc{verb}\rangle) \\ \textsc{cat} & S \\ \textsc{subj} & \begin{bmatrix} \textsc{cat} & \mathit{NP} \end{bmatrix} \\ \textsc{verb} & \begin{bmatrix} \textsc{cat} & \mathit{VP} \\ \textsc{lex} & \mathit{see} \\ \textsc{tense} & \mathit{past} \end{bmatrix} \end{bmatrix}$$

Because the AVMs $[B^v]$ and $[C^v]$ do not unify, there are only three ways to continue the unification successfully. The AVM $[B^s]$ can be unified with $[C]$

at the attribute VERB; $[B^s]$ can also be unified with $[C^v]$ in the standard way; and the AVM $[C^v]$ can be unified with $[B]$ at the attribute SUBJ. All three possibilities result in the following AVM for the sentence "*John saw.*"

$$\begin{bmatrix} \text{C-SET} & (\langle \text{VERB} \rangle \, \langle \text{SUBJ} \rangle) \\ \text{PATTERN} & (\langle \text{SUBJ} \rangle \, \langle \text{VERB} \rangle) \\ \text{CAT} & S \\ \text{SUBJ} & \begin{bmatrix} \text{CAT} & NP \\ \text{LEX} & John \end{bmatrix} \\ \text{VERB} & \begin{bmatrix} \text{CAT} & VP \\ \text{LEX} & see \\ \text{TENSE} & past \end{bmatrix} \end{bmatrix}$$

4.1.2 Comparison with CUG

Let us now compare the description of Functional Unification Grammar given this far and the definition of Categorial Unification Grammar (CUG) from Section 3.3.1. This comparison reveals some implicit notions of FUG that are important for the simulation. FUG and CUG differ in the following three points.

- The most salient difference between FUG and CUG is the least essential difference. FUG and CUG differ in the way reentrance is expresses. The former expresses reentrance by means of path-equations; the latter by means of box-labels. As Smolka (1992) indicates and is exemplified in Section 3.2.2, this difference is accommodated in quasi-linear time.
- The descriptions that state how words are combined differ in CUG and FUG. CUG one the one hand is a lexicon based formalism. That is, the entries in the lexicon determine how words are combined, whereas the rules of CUG-grammars are fixed. Consequently, a CUG-grammar assigns binary branching constituent structure trees to strings. FUG on the other hand is a rule based formalism. That is, the lexicon is largely fixed, whereas the rules of an FUG-grammar have an arbitrary form. A computationally unattractive property of these rules is that arbitrary constituent structure trees may be assigned to strings. As Ritchie (1985) shows, these arbitrary constituent structure trees result in an undecidable recognition problem. Ritchie (1985) suggests that a decidable recognition problem is obtained if the branching of constituent structure trees is constrained. In order to make the simulation computationally interesting, we restrict our attention to decidable fragments of FUG. Therefore we assume that the number of unary branchings is bounded, e.g., with respect to the size of the grammar.
- CUG distinguishes arbitrary, incomplete, strings from complete strings, usually called sentences. The former strings are formed only to serve as subparts of the latter strings. This distinction is explicitly made clear in CUG-grammars: one primitive category is distinguished in the description of a CUG-grammar. All and only all strings that result from this distinguished category are the sentences, i.e., the strings in the language that the particular grammar describes.

This distinction between incomplete strings and complete strings only comes to front in FUG when parsing is discussed. In general, parsers determine the constituent structure tree that corresponds to some string. The nodes in a constituent structure tree of FUG are AVMs. A parser for FUG-grammars has to determine whether the root node of a constituent structure tree has the form of some special AVM (cf., Kay 1985, Karttunen and Kay 1985). For the benefit of the simulation, we make this special AVM explicit. Thus we introduce a distinguished AVM in the description of an FUG-grammar.

Informally, we can think of an FUG-grammar as a set of AVMs plus a distinguished AVM. The set of AVMs may be partitioned into two disjoint subsets. One subset, the lexicon, consists of only AVMs that contain the special attribute LEX. The other subset is the set of combinatory rules. This subset consists of only AVMs that contain the special attributes C-SET and PATTERN. The values of these two attributes are simply sets and lists of paths. Hence the additional ways to unify AVMs are limited to the situation exemplified by 3 in Example 4.1.1. The distinguished AVM is used to identify the sentences that the FUG-grammar generates. Finally we require the number of unary branchings in derivations of the FUG-grammar to be bounded.

As an example of a simple FUG-grammar, consider 3 in Example 4.1.1. The AVMs $[B]$ and $[C]$ denote the lexicon, AVM $[A]$ is the only combinatory rule. We can take $\left[\ \textsc{cat} \quad \mathrm{S}\ \right]$ as the distinguished AVM.

4.2 The Simulation

In this section we will present a simulation f of Categorial Unification Grammar by Functional Unification Grammar In Section 4.3 we will prove the correctness of f. That is, CUG-grammar G generates string w iff FUG-grammar $f(G)$ generates w. In Section 4.4 we will show that f is a polynomial time many-one reduction. Hence the recognition problem of FUG is NP-hard.

We recall that a CUG-grammar is described by its lexicon, its distinguished primitive category, and its combinatory rules to combine categories. This simulation resembles strongly the way in which CUG is often defined (cf., Uszkoreit 1986, Bouma 1993). We avoid unnecessary complex AVMs by removing the attribute C-SET from our descriptions. The information encoded in this attribute can also be retrieved from the attribute PATTERN.

First we will show how the simulation maps CUG-categories onto AVMs in FUG. This shows how the distinguished primitive category is mapped onto the distinguished AVM of the FUG-grammar. Then we will explain how the lexicon of CUG is mapped onto a lexicon in FUG. Finally we will show how the combinatory rules of the CUG-grammar are mapped onto the grammar rules of the FUG-grammar.

Categories. The AVMs in CUG express reentrance by box-labels, whereas AVMs in FUG express reentrance by means of path-equalities. We recall from Section 3.2.2

that an AVM with box-labels, $[F]$, can be transformed into an equivalent AVM with path-equalities, $[F]^\circ$, in quasi-linear time. The simulation f maps CUG-categories onto AVMs. Because the set of categories in CUG is defined inductively, the simulation is defined inductively, as described below. Let the values x, y be either atomic or AVMs with attributes VAL, DIR and ARG.

1. If A $[F]$ is a primitive CUG-category, then the simulation f maps this CUG-category onto the following AVM.

$$\begin{bmatrix} \textsc{cat} & \mathrm{A} \\ \textsc{feat} & [F] \end{bmatrix}^\circ$$

2. Let the simulation f map CUG-category X $[F]$ and CUG-category Y $[H]$ onto the following two AVMs, respectively.

$$\begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix}^\circ \text{ and } \begin{bmatrix} \textsc{cat} & y \\ \textsc{feat} & [H] \end{bmatrix}^\circ$$

   Then f maps the CUG-categories $\mathrm{X}[F] \underset{[H]}{\backslash} \mathrm{Y}$ and $\mathrm{Y} \underset{[H]}{/} \mathrm{X}[F]$ onto the following two AVMs.

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} & \begin{bmatrix} \textsc{cat} & y \end{bmatrix} \\ \textsc{dir} & \textit{left} \\ \textsc{arg} & \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix} \end{bmatrix} \\ \textsc{feat} & [H] \end{bmatrix}^\circ \text{ and } \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} & \begin{bmatrix} \textsc{cat} & y \end{bmatrix} \\ \textsc{dir} & \textit{right} \\ \textsc{arg} & \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix} \end{bmatrix} \\ \textsc{feat} & [H] \end{bmatrix}^\circ$$

Distinguished AVM. The distinguished category in CUG identifies derivations that produce sentences. The sentences that an FUG-grammar produces are identified by the distinguished AVM. The simulation maps the distinguished primitive category S onto the distinguished AVM $\begin{bmatrix} \textsc{cat} & \mathrm{S} \end{bmatrix}$.

The lexicon. The simulation of the lexicon is straightforward. All entries of the CUG-lexicon are mapped onto AVMs in the FUG-lexicon, and the FUG-lexicon contains no other AVMs. This mapping is defined in the following way. Let $w : \mathrm{X}[F]$ be an entry of the CUG-lexicon and let CUG-category X $[F]$ be mapped onto the AVM $[F_1]$. Then the simulation f maps the entry of the CUG-lexicon onto the entry $[H_1]$ in the FUG-lexicon, where

$$[F_1] = \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix}^\circ \text{ and } [H_1] = \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \\ \textsc{lex} & w \end{bmatrix}^\circ.$$

Combinatory rules. CUG contains two combinatory rules: left and right application. The rules can be viewed as descriptions of schemas for constituent structures. The schemas denote that a constituent consists of one mother category and

$$\begin{bmatrix} \text{PATTERN} & (\langle\text{ARG-DTR}\rangle\,\langle\text{FUN-DTR}\rangle) \\ \text{FUN-DTR} & \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{DIR} & \textit{left} \end{bmatrix} \end{bmatrix} \\ \langle\text{CAT}\rangle & \doteq \langle\text{FUN-DTR CAT VAL CAT}\rangle \\ \langle\text{FEAT}\rangle & \doteq \langle\text{FUN-DTR FEAT}\rangle \\ \langle\text{ARG-DTR}\rangle & \doteq \langle\text{FUN-DTR CAT ARG}\rangle \end{bmatrix} \quad \text{(Referred to as } [F_{la}]\text{)}$$

$$\begin{bmatrix} \text{PATTERN} & (\langle\text{FUN-DTR}\rangle\,\langle\text{ARG-DTR}\rangle) \\ \text{FUN-DTR} & \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{DIR} & \textit{right} \end{bmatrix} \end{bmatrix} \\ \langle\text{CAT}\rangle & \doteq \langle\text{FUN-DTR CAT VAL CAT}\rangle \\ \langle\text{FEAT}\rangle & \doteq \langle\text{FUN-DTR FEAT}\rangle \\ \langle\text{ARG-DTR}\rangle & \doteq \langle\text{FUN-DTR CAT ARG}\rangle \end{bmatrix} \quad \text{(Referred to as } [F_{ra}]\text{)}$$

Table 4.1: Attribute-value matrices corresponding to the application rules.

two daughter categories. One of these daughter categories is the compound of the mother category and the other daughter category.

The simulation f maps these schemas onto the two AVMs given in Table 4.1. The left application rule is simulated by the AVM $[F_{la}]$, the right application rule by the AVM $[F_{ra}]$. The attribute PATTERN is the special attribute that identifies the constituents and their order. The attributes CAT, DIR, FEAT, VAL and the values *left*, *right* are familiar from the inductive definition of categories. The attributes ARG-DTR and FUN-DTR are used to refer to the daughters that correspond to the argument and functor, respectively, in the application rule.

4.2.1. DEFINITION. Given a CUG-grammar G, the *simulation* f maps this grammar onto the FUG-grammar $f(G)$, where $f(G)$ is a set of AVMs plus a distinguished AVM. The set of AVMs can be divided in two disjunct subsets: a lexicon, G_L, and a set of rules, G_P. The *lexicon* consists of all AVMs in $f(G)$ that contain the special attribute LEX. The set of *combinatory rules* consists of all AVMs in $f(G)$ that contain the special attribute PATTERN.

- The *lexicon* is defined as the set of AVMs

  $$G_L = \{[H] \mid f(w : X[F]) = [H],\ w : X[F] \text{ an entry from the lexicon of } G\}$$

  We recall that $[H]$ has the following form.

  $$\begin{bmatrix} \text{CAT} & x \\ \text{FEAT} & [F] \\ \text{LEX} & w \end{bmatrix}^{\circ}$$

- The *set of combinatory rules* is defined as the set of AVMs from Table 4.1:

  $$G_P = \{[F_{la}], [F_{ra}]\}.$$

- The *distinguished AVM* is defined as $\begin{bmatrix} \text{CAT} & S \end{bmatrix}$ if S is the distinguished category of G.

Let us now consider a simple example of the simulation. We will assume some small CUG-grammar and show for some CUG-categories how they are mapped onto AVMs. Then we present some examples of entries in the lexicon of the simulating FUG-grammar. We conclude with some examples of AVMs that may be constructed by the simulating FUG-grammar.

4.2.2. EXAMPLE. Let there be a CUG-grammar with the ordinary combinatory rules, the distinguished category S and the following lexicon with five entries:

the: $\mathrm{N} \;\underset{\boxed{1}[\textsc{def}\ +]}{/}\; \mathrm{N}\boxed{1}$ a: $\mathrm{N} \;\underset{\boxed{1}[\textsc{def}\ -]}{/}\; \mathrm{N}\boxed{1}[\textsc{num}\ \mathit{sing}]$ kisses: $(\mathrm{N}\boxed{1}\begin{bmatrix}\textsc{pers} & \mathit{3rd}\\ \textsc{num} & \mathit{sing}\end{bmatrix} \underset{\boxed{1}}{\backslash} \mathrm{S})/\mathrm{N}$

boy: $\mathrm{N}\begin{bmatrix}\textsc{pers} & \mathit{3rd}\\ \textsc{num} & \mathit{sing}\end{bmatrix}$ girl: $\mathrm{N}\begin{bmatrix}\textsc{pers} & \mathit{3rd}\\ \textsc{num} & \mathit{sing}\end{bmatrix}$

Now the simulation maps the CUG-category $\mathrm{N} \;\underset{\boxed{1}[\textsc{def}\ -]}{/}\; \mathrm{N}\boxed{1}[\textsc{num}\ \mathit{sing}]$ for the string “a” onto the following AVM.

$$\begin{bmatrix}
\textsc{cat} & \begin{bmatrix}\textsc{val} & [\textsc{cat}\ N]\\ \textsc{dir} & \mathit{right}\\ \textsc{arg} & \begin{bmatrix}\textsc{cat} & N\\ \textsc{feat} & [\textsc{num}\ \mathit{sing}]\end{bmatrix}\end{bmatrix}\\
\textsc{feat} & [\textsc{def}\ -]\\
\langle\textsc{feat}\rangle & \doteq \langle\textsc{cat arg feat}\rangle
\end{bmatrix}$$

Similarly, the CUG-category $\mathrm{N}\boxed{1}\begin{bmatrix}\textsc{pers} & \mathit{3rd}\\ \textsc{num} & \mathit{sing}\end{bmatrix} \underset{\boxed{1}}{\backslash} \mathrm{S}$ is mapped onto the following AVM.

$$\begin{bmatrix}
\textsc{cat} & \begin{bmatrix}\textsc{val} & [\textsc{cat}\ S]\\ \textsc{dir} & \mathit{left}\\ \textsc{arg} & \begin{bmatrix}\textsc{cat} & N\\ \textsc{feat} & \begin{bmatrix}\textsc{pers} & \mathit{3rd}\\ \textsc{num} & \mathit{sing}\end{bmatrix}\end{bmatrix}\end{bmatrix}\\
\langle\textsc{feat}\rangle & \doteq \langle\textsc{cat arg feat}\rangle
\end{bmatrix}$$

The lexicon of the CUG-grammar is mapped onto the lexicon of the FUG-grammar. From the previous examples it is clear that the FUG-grammar contains the following two entries for the words “the” and “kisses.”

$$\begin{bmatrix}
\textsc{cat} & \begin{bmatrix}\textsc{val} & [\textsc{cat}\ N]\\ \textsc{dir} & \mathit{right}\\ \textsc{arg} & [\textsc{cat}\ N]\end{bmatrix}\\
\textsc{feat} & [\textsc{def}\ +]\\
\textsc{lex} & \mathit{the}\\
\langle\textsc{feat}\rangle & \doteq \langle\textsc{cat arg feat}\rangle
\end{bmatrix}$$

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} \mid \textsc{cat} & \begin{bmatrix} \textsc{val} & [\textsc{cat}\ S] \\ \textsc{dir} & \textit{left} \\ \textsc{arg} & \begin{bmatrix} \textsc{cat} & N \\ \textsc{feat} & \begin{bmatrix} \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \textsc{dir} & \textit{right} \\ \textsc{arg} & [\textsc{cat}\ N] \end{bmatrix} \\ \textsc{lex} & \textit{kisses} \\ \multicolumn{2}{l}{\langle \textsc{cat val feat} \rangle \doteq \langle \textsc{cat val cat arg feat} \rangle} \end{bmatrix}$$

Given the entries in the lexicon of the FUG-grammar, we can construct complex AVMs for constituents. These complex AVMs are constructed by unifying the lexical AVMs with the values of the attributes ARG-DTR and FUN-DTR in the AVM $[F_{la}]$ or $[F_{ra}]$.

Let us consider the AVMs that belong to the constituents "the boy" and "a girl." The AVM for "the boy" will be specified in its full detail. The AVM for the determiner "the" is unified with the value of the attribute FUN-DTR in the AVM $[F_{ra}]$. This unification is possible because the value of the path $\langle \textsc{cat dir} \rangle$ in the AVM for "the" is *right*.

In a similar way, the AVM for the noun "boy" is unified with the value of the attribute ARG-DTR in $[F_{ra}]$. This unification is possible because the AVM for "boy" unifies with the path $\langle \textsc{cat arg} \rangle$ in the AVM for "the."

$$\begin{bmatrix} \textsc{pattern} & (\langle \textsc{fun-dtr} \rangle\ \langle \textsc{arg-dtr} \rangle) \\ \textsc{fun-dtr} & \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} & [\textsc{cat}\ N] \\ \textsc{dir} & \textit{right} \\ \textsc{arg} & [\textsc{cat}\ N] \end{bmatrix} \\ \textsc{feat} & [\textsc{def}\ +] \\ \textsc{lex} & \textit{the} \end{bmatrix} \\ \textsc{arg-dtr} & \begin{bmatrix} \textsc{cat} & N \\ \textsc{feat} & \begin{bmatrix} \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{bmatrix} \\ \textsc{lex} & \textit{boy} \end{bmatrix} \\ \langle \textsc{cat} \rangle & \doteq \langle \textsc{fun-dtr cat val cat} \rangle \\ \langle \textsc{feat} \rangle & \doteq \langle \textsc{fun-dtr feat} \rangle \\ \langle \textsc{fun-dtr feat} \rangle & \doteq \langle \textsc{fun-dtr cat arg feat} \rangle \\ \langle \textsc{arg-dtr} \rangle & \doteq \langle \textsc{fun-dtr cat arg} \rangle \end{bmatrix}$$

We confine to a simplified AVM for the constituent "a girl" because it is rather similar to the AVM for "the boy."

$$\begin{bmatrix} \textsc{pattern} & (\langle \textsc{fun-dtr} \rangle\ \langle \textsc{arg-dtr} \rangle) \\ \textsc{cat} & N \\ \textsc{fun-dtr} & [\textsc{lex}\ \textit{a}] \\ \textsc{arg-dtr} & [\textsc{lex}\ \textit{girl}] \\ \textsc{feat} & \begin{bmatrix} \textsc{det} & - \\ \textsc{num} & \textit{sing} \end{bmatrix} \end{bmatrix}$$

Let us now consider simplified AVMs that belong to the constituents "kisses a girl" and "the boy kisses a girl." In these cases the AVMs for the constituents "kisses," "a girl," "the boy" and "kisses a girl" are unified with the values of the attributes FUN-DTR and ARG-DTR.

$$\left[\begin{array}{ll}
\text{PATTERN} & (\langle\text{FUN-DTR}\rangle\ \langle\text{ARG-DTR}\rangle) \\
\text{CAT} & \left[\begin{array}{ll}
\text{VAL} & [\text{CAT}\ S] \\
\text{DIR} & \textit{left} \\
\text{ARG} & \left[\begin{array}{ll}
\text{CAT} & N \\
\text{FEAT} & \left[\begin{array}{ll} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{array}\right]
\end{array}\right]
\end{array}\right] \\
\text{FUN-DTR} & [\text{LEX}\ \textit{kisses}] \\
\text{ARG-DTR} & \left[\begin{array}{ll} \text{FUN-DTR} & [\text{LEX}\ \textit{a}] \\ \text{ARG-DTR} & [\text{LEX}\ \textit{girl}] \end{array}\right] \\
\langle\text{FEAT}\rangle & \doteq \langle\text{CAT ARG FEAT}\rangle
\end{array}\right]$$

$$\left[\begin{array}{ll}
\text{PATTERN} & (\langle\text{ARG-DTR}\rangle\ \langle\text{FUN-DTR}\rangle) \\
\text{CAT} & S \\
\text{FUN-DTR} & \left[\begin{array}{ll}
\text{FUN-DTR} & [\text{LEX}\ \textit{kisses}] \\
\text{ARG-DTR} & \left[\begin{array}{ll} \text{FUN-DTR} & [\text{LEX}\ \textit{a}] \\ \text{ARG-DTR} & [\text{LEX}\ \textit{girl}] \end{array}\right]
\end{array}\right] \\
\text{ARG-DTR} & \left[\begin{array}{ll} \text{FUN-DTR} & [\text{LEX}\ \textit{the}] \\ \text{ARG-DTR} & [\text{LEX}\ \textit{boy}] \end{array}\right] \\
\text{FEAT} & \left[\begin{array}{ll} \text{DEF} & + \\ \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{array}\right]
\end{array}\right]$$

The reader should notice that the AVM for "the boy kisses a girl" has the form that corresponds to the distinguished CUG-category S. Hence this AVM describes a complete string.

4.3 Correctness of the Simulation

The main purpose of this section is to prove the correctness of the simulation f. That is, CUG-grammar G and FUG-grammar $f(G)$ generate the same language. We will start with some definitions that will be convenient in the proofs of the correctness of the simulation.

4.3.1 Definitions

The various articles on FUG contain clear definitions about attributes, values, unification, etcetera. However, no definitions about the production process are provided. The production process is only explained in the text (cf., Kay 1985, Karttunen and Kay 1985) roughly in the following way.

Suppose a grammar G which consists of a set of rules G_P, and a lexicon G_L. Given an AVM $[F]$, which constitutes the specification of a string to be uttered, generation is explained as follows. The AVM $[F]$ is unified with an AVM in the set G_P or G_L. If this unification fails, then the grammar does not generate strings that meet the specification given by the AVM. If the unification is successful, the result will be to add detail to the AVM $[F]$. If this result of the unification contains constituents, the process is repeated unifying each constituent in turn with the grammar. An AVM that has no constituents is a terminal that must unify with some entry in the lexicon.

We define these ideas on generation by means of the three notions "derives," "produces," and "generates." The definitions of these notions are rather similar to Definition 3.3.11, 3.3.12, and 3.3.13

4.3.1. DEFINITION. Let G be an FUG-grammar with set of rules G_P and lexicon G_L, and let $[F]$ be an AVM. The AVM $[F]$ *derives* AVM $[H]$ iff the unification of $[F]$ with some $[G'] \in G_P \cup G_L$ yields $[H']$, where

1. $[H] = [H']$, or
2. $[H']$ contains attribute PATTERN with structured value $(p_1\, p_2)$, and p_i has value $[F_i']$ in $[H']$, where $[F_i']$ derives $[H_i']$ for some i $(1 \leq i \leq 2)$, and the unification of $[H']$ with $[p_i\; [H_i']]$ yields $[H]$.

4.3.2. DEFINITION. Let G be an FUG-grammar, $[F]$ be an AVM, and w a string. The AVM $[F]$ *produces* string w iff $[F]$ derives some AVM $[H]$ that has yield w.

4.3.3. DEFINITION. Let $[F]$ be an AVM, and w a string. AVM $[F]$ has *yield* w iff

1. AVM $[F]$ contains attribute LEX with value w, but does not contain attribute PATTERN; or
2. AVM $[F]$ does not contain attribute LEX, but contains attribute PATTERN with structured value $(p_1\, p_2)$, and the value of path p_i is $[F_i]$, where $[F_i]$ has yield w_i, and $w = w_1\, w_2$.

4.3.4. DEFINITION. FUG-grammar G *generates* string w iff the distinguished AVM of G produces string w. The *language* that grammar G generates $(L(G))$ is defined as the set of all strings that are produced by the distinguished AVM of G.

4.3.2 Proof of Correctness

Given the definitions above, we can prove the correctness of the simulation in Theorem 4.3.9. Lemmas 4.3.5 and 4.3.6 show that one single application of a combinatory rule for a CUG-grammar G can be simulated by the FUG-grammar $f(G)$. Lemma 4.3.8 shows that CUG-categories and the corresponding AVMs in FUG produce the same strings.

The following lemma states that an FUG-constituent with mother $[F_3]$, left complement-daughter $[F_1]$, and right head-daughter $[F_2]$ can be constructed iff a CUG-constituent can be constructed, with mother $\mathrm{Y}\,[H]$, left daughter $\mathrm{X}\,[F]$, and right daughter $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$.

4.3.5. LEMMA. *Let the simulation map* $\mathrm{X}\,[F]$ *onto* $[F_1]^\circ$, $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ *onto* $[F_2]^\circ$, *and* $\mathrm{Y}\,[H]$ *onto* $[F_3]^\circ$. *CUG-category* $\mathrm{Y}\,[H]$ *derives the sequence* $\mathrm{X}\,[F]\ (\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ *iff the AVM* $[F_0]$ *derives* $[F_{app}]$, *where*

$$[F_0] = \begin{bmatrix} \textsc{fun-dtr} & [F_2] \\ \textsc{arg-dtr} & [F_1] \\ \textsc{cat} & y \\ \textsc{feat} & [H] \end{bmatrix}^\circ \quad \textit{and} \quad [F_{app}] = \begin{bmatrix} \textsc{pattern} & (\langle \textsc{arg-dtr} \rangle\ \langle \textsc{fun-dtr} \rangle) \\ \textsc{fun-dtr} & [F_2] \\ \textsc{arg-dtr} & [F_1] \\ \textsc{cat} & y \\ \textsc{feat} & [H] \\ \langle \textsc{cat} \rangle & \doteq \langle \textsc{fun-dtr cat val cat} \rangle \\ \langle \textsc{feat} \rangle & \doteq \langle \textsc{fun-dtr feat} \rangle \\ \langle \textsc{arg-dtr} \rangle & \doteq \langle \textsc{fun-dtr cat arg} \rangle \end{bmatrix}^\circ .$$

Proof. The simulation maps $\mathrm{X}\,[F]$ onto $[F_1]^\circ$, $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ onto $[F_2]^\circ$, and $\mathrm{Y}\,[H]$ onto $[F_3]^\circ$, where

$$\begin{aligned} [F_1]^\circ &= \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix}^\circ \\ [F_2]^\circ &= \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} & [\textsc{cat}\ v] \\ \textsc{dir} & \textit{left} \\ \textsc{arg} & \begin{bmatrix} \textsc{cat} & u \\ \textsc{feat} & [F'] \end{bmatrix} \end{bmatrix} \\ \textsc{feat} & [H'] \end{bmatrix}^\circ \\ [F_3]^\circ &= \begin{bmatrix} \textsc{cat} & y \\ \textsc{feat} & [H] \end{bmatrix}^\circ \end{aligned}$$

Only if: Assume CUG-category $\mathrm{Y}\,[H]$ derives the sequence $\mathrm{X}\,[F]\ (\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$. Then the left application of $\mathrm{X}\,[F]$ and $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ yields $\mathrm{Y}\,[H]$. So both $\mathrm{X}\,[F]$ and $\mathrm{U}[F']$, and $\mathrm{V}\,[H']$ and $\mathrm{Y}\,[H]$ are unifiable.

The unification of AVM $[F_{la}] \in G_P$ with $[F_0]$ results in AVM $[F_{app}]$, provided that the three path-equations in $[F_{app}]$ are satisfiable. So y should unify with v, and $[H]$ with $[H']$, while simultaneously the AVMs

$$\begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix} \text{ and } \begin{bmatrix} \textsc{cat} & u \\ \textsc{feat} & [F'] \end{bmatrix}$$

should unify. This simultaneous unification follows by the assumption.

If: Suppose that AVM $[F_0]$ derives AVM $[F_{app}]$. The left application of $\mathrm{X}\,[F]$ and $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ yields $\mathrm{Y}\,[H]$ iff the unification of $\mathrm{X}\,[F]$ with $\mathrm{U}\,[F']$ turns $\mathrm{V}\,[H']$ into $\mathrm{Y}\,[H]$.

From the equation $\langle \text{ARG-DTR} \rangle \doteq \langle \text{FUN-DTR CAT ARG} \rangle$ it follows that the AVMs

$$\begin{bmatrix} \text{CAT} & x \\ \text{FEAT} & [F] \end{bmatrix} \text{ and } \begin{bmatrix} \text{CAT} & u \\ \text{FEAT} & [F'] \end{bmatrix}$$

unify. So $\mathrm{X}\,[F]$ unifies with $\mathrm{U}\,[F']$. Furthermore, because $[F_0]$ derives AVM $[F_{app}]$ and the equations $\langle \text{CAT} \rangle \doteq \langle \text{FUN-DTR CAT VAL CAT} \rangle$ and $\langle \text{FEAT} \rangle \doteq \langle \text{FUN-DTR FEAT} \rangle$ are satisfied, it follows that both $[H']$ and v can be extended such that $v = y$ and $[H'] = [H]$. So $\mathrm{V}\,[H']$ turns into $\mathrm{Y}\,[H]$. □

The following lemma states roughly that an FUG-constituent with mother $[F_3]$, left head-daughter $[F_2]$, and right complement-daughter $[F_1]$ can be constructed iff a CUG-constituent with mother $\mathrm{Y}\,[H]$, left daughter $\mathrm{V}\underset{[H']}{/}\mathrm{U}[F']$ and right daughter $\mathrm{X}\,[F]$ can be constructed.

4.3.6. LEMMA. *Let the simulation map* $\mathrm{Y}\,[H]$ *onto* $[F_3]^\circ$, $\mathrm{V}\underset{[H']}{/}\mathrm{U}[F']$ *onto* $[F_2]^\circ$, *and* $\mathrm{X}\,[F]$ *onto* $[F_1]^\circ$. *CUG-category* $\mathrm{Y}\,[H]$ *derives the sequence* $(\mathrm{V}\underset{[H']}{/}\mathrm{U}[F'])\ \mathrm{X}\,[F]$ *iff the AVM* $[F_0]$ *derives* $[F_{app}]$, *where*

$$[F_0] = \begin{bmatrix} \text{FUN-DTR} & [F_2] \\ \text{ARG-DTR} & [F_1] \\ \text{CAT} & y \\ \text{FEAT} & [H] \end{bmatrix}^\circ \quad \textit{and} \quad [F_{app}] = \begin{bmatrix} \text{PATTERN} & (\langle \text{FUN-DTR} \rangle\, \langle \text{ARG-DTR} \rangle) \\ \text{FUN-DTR} & [F_2] \\ \text{ARG-DTR} & [F_1] \\ \text{CAT} & y \\ \text{FEAT} & [H] \\ \langle \text{CAT} \rangle & \doteq \langle \text{FUN-DTR CAT VAL CAT} \rangle \\ \langle \text{FEAT} \rangle & \doteq \langle \text{FUN-DTR FEAT} \rangle \\ \langle \text{ARG-DTR} \rangle & \doteq \langle \text{FUN-DTR CAT ARG} \rangle \end{bmatrix}^\circ$$

Proof. Similar to the proof of Lemma 4.3.5. □

4.3.7. FACT. AVM $[F]$ produces string $w_0 \ldots w_n$ iff $[F]$ derives AVM $[H]$ with yield $w_0 \ldots w_n$ iff there are sequences of AVMs, $[G_0] \ldots [G_k]$, and pair-wise distinct paths, $p_0 \ldots p_k$ (with p_0 is the empty path, $k \geq n$) where $[G_{k-i}]$ is the entry in the lexicon for the string w_i, such that the unification of $[F]$ with all $[p_i\,[G_i]]$ results in $[H]$.

4.3.8. LEMMA. *Given a CUG-grammar G. Let* w *be a string and* $\mathrm{Y}\,[H]$ *be a CUG-category. Let the simulation map* $\mathrm{Y}\,[H]$ *onto* $[F_3]^\circ$. *CUG-category* $\mathrm{Y}\,[H]$ *produces string* w *in G iff the AVM* $[F_3]^\circ$ *produces string* w *in FUG-grammar* $f(G)$.

Proof. (Sketch of the proof.) By induction on the length of the string w.

String w is in the lexicon. The CUG-lexicon contains entry $w : \mathrm{Y}\,[H]$ iff the FUG-lexicon contains entry $\begin{bmatrix} \text{CAT} & y \\ \text{FEAT} & [H] \\ \text{LEX} & w \end{bmatrix}^\circ$. Hence $\mathrm{Y}\,[H]$ produces w iff the unification of $[F_3]^\circ$ with the entry for w has yield w.

String w is not in the lexicon. Let the simulation map Y $[H]$ onto $[F_3]^\circ$, U $[F']$ onto $[F_1]^\circ$, and V $[H']$ onto $[F_2]^\circ$. String $w = w_1 w_2$, and Y $[H]$ derives sequence U $[F']$ V $[H']$ iff AVM $[F_0]$ derives $[F_{app}]$, as in Lemmas 4.3.5 and 4.3.6. By induction U $[F']$ produces w_1 iff $[F_1]^\circ$ produces w_1, and V $[H']$ produces w_2 iff $[F_2]^\circ$ produces w_2. Hence $[F_1]^\circ$ derives some AVM $[H_1']$ with yield w_1, and $[F_2]^\circ$ derives some AVM $[H_2']$ with yield w_2.

Assume that $[F_0]$ and $[F_{app}]$ are as presented in Lemma 4.3.5. Then $[F_0]$ derives the following AVM $[H_0']$.

$$[H_0'] = \begin{bmatrix} \text{PATTERN} & (\langle \text{FUN-DTR} \rangle \; \langle \text{ARG-DTR} \rangle) \\ \text{FUN-DTR} & [H_2'] \\ \text{ARG-DTR} & [H_1'] \\ \text{CAT} & y \\ \text{FEAT} & [H] \\ \langle \text{CAT} \rangle & \doteq \langle \text{FUN-DTR CAT VAL CAT} \rangle \\ \langle \text{FEAT} \rangle & \doteq \langle \text{FUN-DTR FEAT} \rangle \\ \langle \text{ARG-DTR} \rangle & \doteq \langle \text{FUN-DTR CAT ARG} \rangle \end{bmatrix}^\circ$$

By the previous fact, there are sequences $[G_0'] \ldots [G_k']$ and $p_0' \ldots p_k'$ such that $[F_1]^\circ$ unified with each $[p_i' \, [G_i']]$ results in $[H_1']$; and there are sequences $[G_0''] \ldots [G_l'']$ and $p_0'' \ldots p_l''$ such that $[F_2]^\circ$ unified with each $[p_i'' \, [G_i'']]$ results in $[H_2']$. By Lemma 4.3.5: Given the sequence of AVMs $[G_0] \ldots [G_{k+l+2}] = [F_{la}][G_0'] \ldots [G_k'][G_0''] \ldots [G_l'']$, and the sequence of path $q_0 \ldots q_{k+l+2} = q_0 \langle \text{ARG-DTR}\, p_0' \rangle \ldots \langle \text{ARG-DTR}\, p_k' \rangle \langle \text{FUN-DTR}\, p_0'' \rangle \ldots \langle \text{FUN-DTR}\, p_l'' \rangle$ (p_0', p_0'', q_0 empty paths), the unification of $[F_3]^\circ$ with $[q_i \, [G_i]]$ results in AVM $[H_0']$ with yield $w_1 w_2$,

Similar argumentation holds if $[F_0]$ and $[F_{app}]$ are as presented in Lemma 4.3.6. Hence $[F_3]^\circ$ produces $w = w_1 w_2$. □

4.3.9. THEOREM. *Given a CUG-grammar G and string w. Let the simulation f map G onto FUG-grammar f(G). CUG-grammar G generates string w iff FUG-grammar f(G) generates string w.*

Proof. Straightforward generalization of Lemma 4.3.8. □

4.4 Formal Properties

In this section we consider two formal properties. First, we handle the complexity of the recognition problems of FUG. Second, we handle the weak generative capacity of FUG.

4.4.1 Complexity of the Recognition Problems

We determine the complexity of the recognition problems in two steps. First, we will provide an *NP*-hard lower bound on the complexity of the recognition problem for a fixed grammar. Second, we will provide an *NP* upper bound on the complexity of the universal recognition problem. These two bounds together determine the complexity of the recognition problems exactly.

A lower bound on the complexity. A lower bound on the complexity of the recognition problems results from the simulation by the following argumentation. Given any CUG-grammar, the simulation f provides us with an FUG-grammar that recognizes the same language as the CUG-grammar. Thus the mapping from a CUG-grammar G onto an FUG-grammar $f(G)$ is a many-one reduction. Moreover, Lemma 4.4.1 proves that the simulation is a polynomial time, many-one reduction. By Theorem 3.3.22, the recognition problem of CUG for a fixed grammar is *NP*-hard. Hence the recognition problems of FUG are *NP*-hard, as indicated by the Theorems 4.4.2 and 4.4.3.

4.4.1. LEMMA. *The FUG-grammar $f(G)$ that simulates a CUG-grammar G is computed in quasi-linear time, with respect to the size of the CUG-grammar G.*

Proof. The mapping from the lexicon of the CUG-grammar onto the lexicon of the FUG-grammar has the largest impact on the cost of the computation of the FUG-grammar. This mapping depends mainly on the mapping from CUG-categories onto AVMs in FUG. This latter mapping costs an amount of time that is quasi-linear in the size of the CUG-category. So the mapping from the CUG-lexicon costs quasi-linear time with respect to the size of the CUG-grammar. Hence the FUG-grammar is computed in quasi-linear time, with respect to the size of the CUG-grammar. □

4.4.2. THEOREM. *The recognition problem of FUG for a fixed grammar is NP-hard.*

Proof. According to Theorem 4.3.9, the FUG-grammar $f(G)$ recognizes the same language as the CUG-grammar G presented in Section 3.3.2. Lemma 4.4.1 shows that the FUG-grammar $f(G)$ is computable in polynomial time, with respect to the size of G. Hence the recognition problem of FUG for a fixed grammar is as hard as the recognition problem of CUG for a fixed grammar. Theorem 3.3.22 proves that the fixed recognition problem of CUG is *NP*-hard. Hence the recognition problem of FUG for a fixed grammar is *NP*-hard. □

4.4.3. THEOREM. *The universal recognition problem of FUG is NP-hard.*

Proof. The universal recognition problem is at least as hard as the recognition problem for a fixed grammar. Theorem 4.4.2 shows that the recognition problem for a fixed grammar is *NP*-hard. Hence the universal recognition problem of FUG is *NP*-hard. □

An upper bound on the complexity. Theorems 4.4.2 and 4.4.3 present lower bounds on the complexity of the recognition problems of FUG. Obviously, we would also like to have an upper bound on the complexity of the recognition problems. In the ideal situation we would provide an upper bound for general versions of FUG. A prerequisite for such an upper bound is that a complete description of FUG is available. Unfortunately, no complete description of FUG exists. So we have to settle for the practical situation and consider fragments of FUG. Let us first consider the fragment of FUG presented in Section 4.1. Then we will consider polynomial extensions of this fragment, and argue that these extensions suffice to describe full FUG-grammars.

We can easily prove an *NP* upper bound on the complexity of the universal recognition problem of this fragment of FUG. The fragment of FUG is based on the standard feature theory from Section 3.2. This standard feature theory has a nice computational property: the unification problem is tractable. We indicated in Section 3.2.2 that minor changes to Smolka's (1992) constraint solving algorithm yield a polynomial time unification algorithm for AVMs. To be more precise, the unification of two AVMs $[F]$ and $[H]$ is computable in polynomial time, with respect to the sizes of $[F]$ and $[H]$. Moreover, the amount of space used by the unification is linear with respect to the sizes of $[F]$ and $[H]$.

The instances of the universal recognition problem consist of an FUG-grammar G and a string w (see Definition 3.2.2). The FUG-grammar G consists of a set of AVMs that forms the lexicon, G_L, a set of AVMs that forms the set of combinatory rules, G_P, and a distinguished AVM.

We call an AVM in G_P a k-ary rule iff the AVM contains the attribute PATTERN, and the value of this attribute is a list with k elements. A derivation contains a *detour* if the derivation contains a sequence of unary rules, and some rule occurs twice in this sequence. If detours are allowed in derivations, then the recognition problem of FUG is undecidable, (cf., Ritchie 1984, Ritchie 1985). In order to obtain a decidable fragment of FUG we forbid detours in derivations. This ban was expressed in Section 4.1 as a bound on the number of unary branchings.

The proof of the *NP* upper bound is based on the following two observations. First, a derivation for a string w and an FUG-grammar G consists of a polynomial amount of steps. This first observation is proven in Lemma 4.4.4. Second, the reverse of a derivation step is computable in polynomial time. This second observation is proven in Lemma 4.4.5.

4.4.4. LEMMA. *A derivation for a string w and a grammar G has polynomial size, with respect to the sizes of w and G.*

Proof. Because we postulated a ban on detours in derivations, every derivation must combine two AVMs after at most an amount of steps that is linear with respect to the size of G_P. Hence the total derivation for a string w consists of at most a linear amount of steps with respect to the size of the string and the size of the grammar. □

4.4.5. LEMMA. *Given an AVM $[G']$ from G_P of which the attribute* PATTERN *has structured value $(p_1 \ldots p_k)$ and k AVMs, $[F_1], \ldots, [F_k]$. The corresponding, reversed, derivation step is computable in polynomial time, with respect to the sizes of $[F_1]$, $\ldots$, $[F_k]$, and $[G']$.*

Proof. Given an AVM $[G']$ from G_P of which the attribute PATTERN has a list with k elements as value. The result of applying this rule $[G']$ is a constituent with k daughters and one mother. The reversed derivation step consists of the unification of the k AVMs $\begin{bmatrix} p_i & [F_i] \end{bmatrix}$ with $[G']$. Hence this reversed derivation step takes an amount of time that is polynomial in the size of the rule and the sizes of the k daughter AVMs. □

Now given a string w and an FUG-grammar G, we guess a linear amount of entries from the lexicon, and a polynomial sized sequence of rules that encode the derivation for w. The previous two lemmas show that both guesses can be checked in polynomial time. Hence the following lemma holds.

4.4.6. LEMMA. *The universal recognition problem of FUG for string w and FUG-grammar G is computable in nondeterministic polynomial time.*

Proof. A derivation for the string w is described by a sequence of entries from the lexicon and a sequence of applied rules. We will guess both sequences and check in polynomial time that the described derivation indeed yields w.

So we guess a sequence of entries from the lexicon of G for the string w. The total size of the sequence is linear in the size of the string and the grammar. Next we guess a sequence of combinatory rules that complete the description of the derivation for w. This sequence consists of a polynomial amount of rules, by Lemma 4.4.4. Now in order to check whether the guesses result in a derivation for the string w, we compute, in reversed order, the derivation from these guesses. Lemma 4.4.5 states that this computation takes polynomial time, with respect to the size of guessed entries and rules, which is polynomial in the size of string w and FUG-grammar G.

All that remains is to check that the distinguished AVM starts the derivation and the string w results from the derivation. Clearly, these final checks only require linear time. □

Let us now consider extensions of the fragment of FUG that we presented in Section 4.1. We consider "polynomial" extensions in which the reversed application of derivation steps remain computable in polynomial time, and in which the derivations remain of polynomial size. Clearly, the recognition problems of these extensions are also computable in nondeterministic polynomial time. We claim that these polynomial extensions may contain

1. disjunctive AVMs;
2. the special values *any* and *none*; and
3. most uses of the attributes C-SET and PATTERN.

We therefore think that it is legitimate to demand that full FUG-grammars are polynomial extensions of the fragment presented in Section 4.1. Hence by Theorems 4.4.2 and 4.4.3:

4.4.7. THEOREM.
(*i*) *The fixed recognition problem of FUG is NP-complete.*
(*ii*) *The universal recognition problem of FUG is NP-complete.*

In addition to the recognition problem of FUG, complexity results of the generation problem of FUG are known. Ritchie (1986) showed that given some FUG-grammar, the question whether an arbitrary AVM produces any string is *NP*-hard. The fragment of FUG that Ritchie (1986) considers is essentially the same as the fragment of FUG presented in Section 4.1. Therefore Lemmas 4.4.4 and 4.4.5 hold also for Ritchie's fragment.

Hence we strengthen Ritchie's (1986) result with an additional *NP* upper bound. We formulate the generation problem as: "Given an FUG-grammar G and an AVM $[F]$, does AVM $[F]$ produce any string?" The *NP* upper bound is proven as follows. By Lemma 4.4.4, the size of a string is polynomial with respect to the size of the AVM $[F]$. So the size of the derivation is polynomial with respect to the sizes of the AVM $[F]$ and the grammar G. Hence guessed sequences of applied rules and entries from the lexicon for $[F]$ can be checked in polynomial time.

4.4.2 Weak Generative Capacity

We will now determine a lower bound and an upper bound on the weak generative capacity of FUG. The lower bound results from the simulation. The upper bound results from the upper bound of the complexity of the fixed recognition problem. These two bounds on the weak generative capacity are represented in Theorem 4.4.8.

A lower bound on the weak generative capacity. Theorem 4.3.9 proves that for every CUG-grammar there exists an FUG-grammar that recognizes the same language. Hence the set of languages recognized by a CUG-grammar, the *CUG-languages*, is a subset of the set of languages recognized by an FUG-grammar, *FUG-languages*. So if we can prove the location of the CUG-languages in the Chomsky hierarchy, we know that the FUG-languages take the same or a higher location in the Chomsky hierarchy. For instance, it is known that the set of languages that the classical categorial grammars generate is the set of context-free languages. Therefore the CUG-languages generate all context-free languages. Hence the FUG-languages generate all context-free languages.

An upper bound on the weak generative capacity. Once again, we are unable to provide an upper bound for general versions of FUG. A prerequisite for such an upper bound is a complete description of FUG, which is not available. So we settle for the fragment of FUG presented in Section 4.1.

Chapter 2 connects the complexity of the recognition problem and weak generative capacity of restricted attribute-value grammars. These restricted attribute-value grammars respect a liberal variation of the so-called *off-line parsability constraint*. This off-line parsability constraint (Johnson 1988) is a well-known generalization of the "Definition of a Valid Derivation" from Lexical Functional Grammar (Kaplan and Bresnan 1982). This off-line parsability constraint relates the amount of "work" done by the grammar to produce a string linearly to the number of terminal symbols produced. It is therefore a sort of honesty constraint that is common in complexity theory.

In Chapter 2 a variation on the off-line parsability constraint is introduced: the *honest parsability constraint*. This honest parsability constraint is a more liberal constraint than the off-line parsability constraint. The honest parsability constraint relates the amount of "work" done by the grammar to produce a string polynomially to the number of terminal symbols produced. For convenience, we state the definition of the honest parsability constraint, Definition 2.5.1, below.

> *A grammar G satisfies the honest parsability constraint iff there exists a polynomial p such that for each string w in the language generated by G there exists a derivation with at most $p(|w|)$ steps.*

A nice property of the restricted attribute-value grammars that respect the honest parsability constraint is that they generate exactly all languages in the complexity class *NP*. Or, as we can restate Theorem 2.5.3:

> *Let L be a language that has an NP recognition algorithm. Then there exists a restricted attribute-value grammar G, that respects the honest parsability constraint, such that G generates the language L.*

By Theorem 4.4.7 the restricted fragment of FUG has an *NP* recognition problem. Hence any restricted FUG-grammar can be simulated by a restricted attribute-value grammar. Thus the languages generated by restricted attribute-value grammars that respect the honest parsability constraint provide an upper bound on the weak generative capacity of restricted FUG-grammars. Hence the following theorem holds.

4.4.8. THEOREM. *The weak generative capacity of the restricted fragment of FUG presented in this chapter is enclosed between the context-free languages and the collection of languages generated by restricted attribute-value grammars that respect the honest parsability constraint.*

Moreover, we conjecture a stronger upper bound for the weak generative capacity of FUG. A close look at the proof of Lemma 4.4.6 shows that the fixed recognition problem of FUG is computable in nondeterministic linear space. We conjecture that the polynomial extensions of the fragment presented in Section 4.1 are also computable in nondeterministic linear space. This implies that the FUG-languages are recognized by linear bounded automata (LBAs). Due to the equivalence of LBAs and context-sensitive grammars, FUG generates only context-sensitive languages (CSLs). Hence we conjecture that the collection of CSLs that are recognized in nondeterministic polynomial time form an upper bound for the weak generative capacity of FUG (cf., Book 1978). Or stated differently:

4.4.9. CONJECTURE. *The collection of context-sensitive languages that the restricted attribute-value grammars that respect the honest parsability constraint generate form an upper bound for the weak generative capacity of FUG.*

Chapter 5

Head-driven Phrase Structure Grammar

In this chapter we will present a formalization of Head-driven Phrase Structure Grammar (HPSG). The main purpose of this chapter is to show that HPSG-grammars can simulate any arbitrary CUG-grammar that was presented in Section 3.3. This simulation provides a way of comparing the different grammatical formalisms. For instance, the simulation shows that the generative capacity of HPSG is at least as high as the generative capacity of CUG. Moreover, because the simulation is computable in polynomial time, it is a polynomial time many-one reduction. Therefore the recognition problem of HPSG is at least as hard as the *NP*-complete recognition problem of CUG.

The next section (Section 5.1) contains an informal introduction in HPSG and discusses which part of HPSG are needed for a simulation of CUG. In Section 5.2 we present the simulation of CUG. In Section 5.3 we prove the correctness of the simulation. We conclude with Section 5.4, in which we discuss the weak generative capacity of HPSG and the complexity of its recognition problem.

5.1 Introduction in HPSG

This section contains an informal introduction in Head-driven Phrase Structure Grammar (HPSG), for a full description see (Pollard and Sag 1987, Pollard and Sag 1994). King (1994) presents a logic which seems close to the logical formalism underlying HPSG. The name "Head-driven Phrase Structure Grammar" covers the main important ideas behind HPSG. The last part of the name states that HPSG uses phrase structures, or, as we call them, constituents. The slogan head-driven exemplifies that the constituents discriminate one of their substructures as the *head*. The head of a constituent is the most important substructure of that constituent. The other substructures of a constituent form the *complement*. The development of the HPSG theory, in particular the constituent component, was influenced by Categorial Grammar.

We will now introduce HPSG in an informal way in two steps. We take the standard feature theory from Section 3.2 as a starting point for the first step. We partly introduce HPSG by comparing it with this standard feature theory. We will also indicate which parts of HPSG are needed for the simulation of CUG. In the second step we compare HPSG and CUG. This comparison will reveal some implicit notions of HPSG that are important for the simulation.

5.1.1 Comparison with the Standard Feature Theory

Let us first introduce a syntactic shorthand in attribute-value matrices (AVMs) in order to avoid needlessly large descriptions of AVMs. Occasionally, we use the following shorthand in AVMs:

$$\Big[\textsc{attr}_1 \mid \ldots \mid \textsc{attr}_n \quad [F]\Big] \text{ stands for } \Big[\textsc{attr}_1 \quad \Big[\ldots \quad [\textsc{attr}_n\, [F]\,]\ldots\Big]\Big].$$

Head-driven Phrase Structure Grammar differs from the standard feature theory in the following four aspects.

- The AVMs in HPSG are not restricted to conjunctions of attribute-value pairs, but can be any conjunctive or disjunctive combination of attribute-value pairs. HPSG has these disjunctive AVMs in common with other unification grammars, for instance, FUG. The disjunctive combination of attribute-value pairs in HPSG is not the only extension with respect to the AVMs of the standard feature theory. HPSG also distinguishes disjunctive values and structured values, besides the familiar atomic and compound values. A disjunctive value is an abbreviation for a disjunctive AVM. The structured values are sets of values and lists of values. In the simulation, however, no use is made of the disjunctive AVMs, the disjunctive values, or the sets. The simulation will only use the structured value list, which is illustrated by 1 in Example 5.1.1.
- HPSG is introduced as a system of sorted AVMs. The AVMs in HPSG are classified in two distinct sorts: phrasal AVMs and lexical AVMs. An AVM is *phrasal* if the AVM contains an attribute DAUGHTERS; otherwise the AVM is *lexical*. Although the fact that the AVMs are sorted does not play an important role in the simulation, the simulation takes this fact into account. We discuss the standard sort-hierarchy of HPSG in 2 of Example 5.1.1.
- HPSG treats three attributes specially: PHONOLOGY (abbreviated to PHON), SUBCATEGORIZATION (SUBCAT), and COMPLEMENT-DAUGHTERS (CDTRS). The attribute PHON has a list of strings as its value. The value of the attributes SUBCAT and CDTRS are lists of atomic and compound values. One might expect that HPSG would also assigns a special status to the attribute HEAD-DAUGHTER (HDTR), because the head-daughter of a constituent is distinguished from the other daughters. Nevertheless, the attribute HDTR is an ordinary attribute. The special status of the head-daughter is enforced by the HPSG-principles, which are discussed next. The three special attributes are illustrated by 3 in Example 5.1.1. All three attributes will play an important role in the simulation.
- HPSG contains *principles* and *schemas*, which control the unification operation. The combined action of principles and schemas state which AVMs

are *well-formed*. There are two kinds of principles: universal ones and language specific ones. The *universal principles* hold for all HPSG-grammars, the *language specific* principles, which might be instantiations of parameterized universal principles, hold for some HPSG-grammars. The three main universal principles are the *Head Feature Principle* (HFP), the *Subcategorization Principle* (SP), and the *Semantics Principle* (SemP). Two important language specific principles are the *Linear Precedence Principle* (LPP) and the *Immediate Dominance Principle* (IDP). The IDP specifies the schemas (ID-schemas) that an HPSG-grammar contains. These ID-schemas describe in which way constituents can be formed. These five principles are exemplified by 4 in Example 5.1.1. For the actual definition of the Linear Precedence Principle, see (Pollard and Sag 1987, Chapter 7). For the precise definition of the other four principles, see (Pollard and Sag 1994, Appendix A.2). The simulation of CUG only uses these five principles.

5.1.1. EXAMPLE.

1. The structured value list is denoted by a sequence of values, which are separated by comma's, and is enclosed in angled brackets. Thus, the list that contains the AVMs $[F]$ and $[H]$ is denoted by $\langle [F], [H] \rangle$.
2. The most general sort of AVM in the standard sort-hierarchy of HPSG is called `sign`. The sort `sign` is divided in two distinct sorts: `lexical` and `phrasal`. Lexical AVMs contain the attribute PHON (an abbreviation for PHONOLOGY) and the attribute SYNSEM (an abbreviation for SYNTAX&SEMANTICS). The value of the attribute PHON is a list of strings, The value of the attribute SYNSEM is of sort `synsem`. This sort `synsem` contains an attribute LOCAL whose value is an AVM that contains the attributes CONT and CAT. The value of attribute CONT, which abbreviates CONTENT, denotes semantic relations. Because semantic relations fall beyond the scope of this syntactic framework, the value of CONT is simply specified by the sort `content`. The value of attribute CAT, which abbreviates CATEGORY, is another AVM with the attributes HEAD and SUBCAT. The attribute HEAD has a value of sort `head`, the value of attribute SUBCAT, which abbreviates SUBCATEGORIZATION, is a list of AVMs of sort `synsem`. Summing up, we can say that a lexical AVM has the following form.

$$\begin{bmatrix} \text{PHON} & \mathtt{string} \\ \text{SYNSEM} & \begin{bmatrix} \text{LOCAL} & \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD} & \mathtt{head} \\ \text{SUBCAT} & \mathtt{synsem-list} \end{bmatrix} \\ \text{CONT} & \mathtt{content} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Phrasal AVMs extend lexical AVMs by an attribute DTRS (DAUGHTERS), whose value is an AVM with the two attributes HDTR (HEAD-DAUGHTER), and CDTRS (COMPLEMENT-DAUGHTERS). The value of the attribute HDTR is an AVM of sort `sign`. The value of the attribute CDTRS is a list of AVMs of sort `sign`. Thus a phrasal AVM has the following form.

$$\begin{bmatrix} \textsc{phon} & \texttt{string} \\ \textsc{synsem} & \begin{bmatrix} \textsc{local} & \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \texttt{head} \\ \textsc{subcat} & \texttt{synsem-list} \end{bmatrix} \\ \textsc{cont} & \texttt{content} \end{bmatrix} \end{bmatrix} \\ \textsc{dtrs} & \begin{bmatrix} \textsc{hdtr} & \texttt{sign} \\ \textsc{cdtrs} & \texttt{sign-list} \end{bmatrix} \end{bmatrix}$$

3. The attribute PHON encodes lexical information. The attribute PHON appears both in lexical and phrasal AVMs. In a lexical AVM, PHON encodes the lexical form of a word. In a phrasal AVM, PHON encodes the lexical form of the constituent, like in

$$\begin{bmatrix} \textsc{phon} & \langle \textit{big, red, book} \rangle \end{bmatrix}.$$

The attributes HDTR and CDTRS describe the constituent structure. The following AVM expresses a constituent that has a *VP* as head-daughter and two *NP*'s as complement-daughters

$$\begin{bmatrix} \textsc{hdtr} & \textit{VP} \\ \textsc{cdtrs} & \langle \textit{NP}, \textit{NP} \rangle \end{bmatrix}.$$

The attribute SUBCAT expresses which constituents are required to complete a constituent structure. A ditransitive verb would be expressed as a verb phrase (*VP*) that is incomplete, i.e., subcategorizes, for three noun phrases (*NP*'s), as indicated by the following AVM.

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \textit{VP} \\ \textsc{subcat} & \langle \textit{NP}, \textit{NP}, \textit{NP} \rangle \end{bmatrix} \end{bmatrix}$$

4. The Head Feature Principle states that in a constituent the mother and the head-daughter share the same syntactic properties, which are specified by the attribute HEAD. The Semantics Principle states that in a constituent the mother and the semantic-daughter, which is often the head-daughter, share the same semantic information. The Subcategorization Principle states that in a constituent the head-daughter subcategorizes for a list of constituents, which has an initial part L and a remaining part L', iff the mother subcategorizes for the list L, and the complement-daughters form the list L'. The Linear Precedence Principle describes the ordering of constituents, but unfortunately the actual description of this principle has remained unclear so far in HPSG. Therefore, for the time being, we postulate that a function on the daughter constituents, called *order-constituents*, accomplishes the right ordering of constituents. Two important ID-schemas for an HPSG-grammar for English are the head-subject ID-schema and the head-complement ID-schema. The head-subject ID-schema states that a constituent consists of a mother with an empty subcategorization-list, a head-daughter and one complement-daughter.

The head-complement ID-schema states that a constituent consists of a mother with a subcategorization-list that contains one element, a head-daughter that is a lexical entry, and some complement-daughters.

5.1.2 Comparison with CUG

We start this paragraph with a comparison between the production processes of HPSG and CUG. In CUG the production process is defined operationally, whereas in HPSG the production process is defined declaratively.

This operational way to define the production process in CUG is exemplified by the combinatory rules. These rules state that two categories can be combined into one new category, and determine the result of this combination. HPSG, however, does not contain such combinatory rules. The principles, which guide the production process in HPSG, are defined as well-formedness constraints on AVMs, i.e., in a declarative way. Thus the principles do not construct AVMs, but serve merely as a filter for the well-formed AVMs. That is, given some collection of AVMs, the principles and schemas discriminates the AVMs that are well-formed from the other AVMs.

In contrast with these declarative definition, an operational approach to the production process is often found in the HPSG-literature, Especially, in the description of examples an operational approach is frequently plied. In this operational approach the principles are not regarded as statements describing which AVMs are well-formed, but as descriptions for the construction of the well-formed AVMs.

Let us now compare the description of HPSG given this far and the definition of CUG from Section 3.3.1. This comparison results in two apparent mismatches that have to be solved before the simulation can be presented. Both mismatches are caused by the different approaches to the production process in HPSG and CUG. The apparent mismatches and their solutions are given below.

- CUG-grammars, on the one hand, contain combinatory rules, which determine the result of combining categories. HPSG-grammars, on the other hand, do not contain explicit rules that construct AVMs. However, we will show below how the principles of HPSG can be regarded as rules that construct AVMs.
  First, we assume that a set of minimal well-formed AVMs is available. All other AVMs are then constructed from these minimal AVMs. The set of minimal AVMs is in fact the lexicon. Second, we regard the principles as prescriptions how AVMs are combined. To be more precise, we regard the principles as descriptions of the form of well-formed AVMs, and we demand that well-formed AVMs have the form described by the principles.
- In CUG arbitrary, incomplete, strings are distinguished from complete strings, usually called sentences. The former strings are formed only to serve as a subpart of the latter strings. This distinction is explicit made clear in CUG-grammars: one primitive category is distinguished in the description of a CUG-grammar. All and only all strings that result from this distinguished category

are the sentences, i.e., the strings in the language that the particular grammar describes.

The distinction between incomplete strings and complete strings is not made explicitly in HPSG. However, an explicit distinction is needed, for the simulation. Therefore, we introduce a distinguished AVM in the description of an HPSG-grammar, like is done for CUG-grammars. This introduction should not be too controversial, because the notion of a special AVM already seems to exist in HPSG. That is, the AVMs that describe sentences all have a common form. The typical common form of these special AVMs describes a verb that has combined with all its arguments, i.e.,

$$\left[\textsc{synsem}\ \left[\textsc{local}\ \left[\textsc{cat}\ \begin{bmatrix}\textsc{head} & \textit{verb}\\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\right]\right]\right].$$

An example may illustrate the distinction between the declarative and the operational approach to production process of HPSG.

5.1.2. Example. Consider the following three simplified AVMs. A phrasal AVM, i.e., an AVM with the attribute DTRS, for the constituent "Kim walks."

$$\begin{bmatrix}\textsc{phon} & \langle \textit{Kim, walks}\rangle\\ \textsc{synsem} & \left[\textsc{local}\mid\textsc{cat}\ \begin{bmatrix}\textsc{head} & \boxed{1}\,\textit{verb}\\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\right]\\ \textsc{dtrs} & \begin{bmatrix}\textsc{hdtr} & \begin{bmatrix}\textsc{phon} & \langle\textit{walks}\rangle\\ \textsc{synsem}\mid\textsc{local}\mid\textsc{cat} & \begin{bmatrix}\textsc{head} & \boxed{1}\,\textit{verb}\\ \textsc{subcat} & \langle\boxed{2}\rangle\end{bmatrix}\end{bmatrix}\\ \textsc{cdtrs} & \left\langle\begin{bmatrix}\textsc{phon} & \langle\textit{Kim}\rangle\\ \textsc{synsem} & \boxed{2}\left[\textsc{local}\mid\textsc{cat}\ \begin{bmatrix}\textsc{head} & \textit{noun}\\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\right]\end{bmatrix}\right\rangle\end{bmatrix}\end{bmatrix} \tag{5.1}$$

A lexical AVM for the string "walks."

$$\begin{bmatrix}\textsc{phon} & \langle\textit{walks}\rangle\\ \textsc{synsem}\mid\textsc{local}\mid\textsc{cat} & \begin{bmatrix}\textsc{head} & \textit{verb}\\ \textsc{subcat} & \left\langle\left[\textsc{local}\mid\textsc{cat}\ \begin{bmatrix}\textsc{head} & \textit{noun}\\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\right]\right\rangle\end{bmatrix}\end{bmatrix} \tag{5.2}$$

A lexical AVM for the string "Kim."

$$\begin{bmatrix}\textsc{phon} & \langle\textit{Kim}\rangle\\ \textsc{synsem} & \left[\textsc{local}\ \left[\textsc{cat}\ \begin{bmatrix}\textsc{head} & \textit{noun}\\ \textsc{subcat} & \langle\,\rangle\end{bmatrix}\right]\right]\end{bmatrix} \tag{5.3}$$

Let the Head Feature Principle be stated as follows. "In phrasal AVMs the following two paths share the same value:

$\langle$DTRS HDTR SYNSEM LOCAL CAT HEAD$\rangle$ and $\langle$SYNSEM LOCAL CAT HEAD$\rangle$."

Let the Subcategorization Principle be stated in the following way: "In phrasal AVMs the concatenation of the values of the paths

$\langle$SYNSEM LOCAL CAT SUBCAT$\rangle$ and $\langle$DTRS CDTRS SYNSEM$\rangle$

equal the value of path ⟨DTRS HDTR SYNSEM LOCAL CAT SUBCAT⟩."

In both the declarative approach and the operational approach all three AVMs are well-formed. In the declarative approach the AVMs are well-formed because they satisfy all principles, i.e., the Head Feature Principle and the Subcategorization Principle. The AVMs in (5.2) and (5.3) satisfy the principles because they are not applicable. The AVM in (5.1) satisfies the principles because the box-labels appear on the right places. In this declarative approach no correlation between the AVMs is found.

In the operational approach we would say that the AVMs in (5.2) and (5.3) are well-formed because they come from the lexicon. The AVM in (5.1) is well-formed because it is constructed from the AVMs in (5.2) and (5.3) according to the prescription of the principles. That is, the AVM in (5.2) is the head-daughter of the AVM in (5.1). Moreover, the AVM in (5.2) subcategorizes for an AVM like the AVM in (5.3). This AVM in (5.3) is the only complement-daughter of the AVM in (5.1). Furthermore, the value of the attribute HEAD in the AVM in (5.2) is passed on to the attribute HEAD of the AVM in (5.1).

Informally, we can think of an HPSG-grammar as a set of AVMs, the lexicon, a distinguished AVM and a set of principles. The combined action of the principles describes forms of well-formed AVMs.

5.2 The Simulation

In this section we will present the *simulation* f of CUG by HPSG. In Section 5.3 we will prove the correctness of f. That is, CUG-grammar G generates string w iff HPSG-grammar $f(G)$ generates w. In Section 5.4 we will show that f is a polynomial time many-one reduction. Hence the recognition problem of HPSG is *NP*-hard.

This simulation f resembles the simulation of CUG by FUG. We recall that a CUG-grammar is described by its lexicon, its distinguished primitive category, and its combinatory rules to combine categories. Below we will first explain how the simulation maps CUG-categories onto AVMs in HPSG. This shows how the distinguished primitive category is mapped onto the distinguished AVM of the HPSG-grammar. Then we will show how the lexicon of CUG is mapped onto a lexicon in HPSG. Finally we show how the principles and schemas in HPSG can copy the combinatory rules of the CUG-grammar.

Unfortunately, the AVMs from HPSG tend to become cumbersomely large, very easily. In order to avoid such large AVMs, we will use a compact description of the AVMs. In these compact descriptions, we will not mention the attributes DTRS, SYNSEM and LOCAL.

Categories. The simulation f maps CUG-categories onto AVMs in HPSG. The AVMs that correspond to CUG functors subcategorize for the AVMs corresponding to the CUG arguments. As a consequence CUG functors are the head-daughters of constituents and CUG arguments are the complement-daughters.

The set of categories in CUG is defined inductively. Therefore, the simulation is defined inductively on the categories. First, we define how the simulation maps a CUG-category A$[F]$, where A is a primitive category, onto an AVM in HPSG. Second, given that we know how the simple CUG-categories are mapped onto AVMs, we define how more complex CUG-categories are mapped onto AVMs.

We can say roughly that the syntactic and semantic information in a CUG-category is located under different attributes of an AVM in HPSG. The attribute CAT of the AVM denotes the syntactic information that is present in a CUG-category. That is, the category information in the CUG-category. The attribute CONT of the AVM denotes the semantic information that is present in a CUG-category. That is, the feature information in the CUG-category.

Furthermore, notice that the AVMs in HPSG given below differ in two aspects, neither one essential, from the standard sort-hierarchy. First, the value of attribute CAT is augmented with an attribute DIR, whose value is either *left* or *right*. Second, the value of attribute HEAD is atomic in the simulation. The symbols x, y stand for atomic values; L_x, L_y stand for lists; and $\oplus$ stands for concatenation of lists.

1. Let $\mathrm{A}\,[F]$ be a primitive category in the CUG-grammar. Then the simulation f maps this CUG-category onto the AVM

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \mathrm{A} \\ \textsc{subcat} & \langle\rangle \end{bmatrix} \\ \textsc{cont} & [F] \end{bmatrix}.$$

2. Let the simulation f map CUG-category $\mathrm{X}\,[F]$ onto $[F']$ and CUG-category $\mathrm{Y}\,[H]$ onto $[H']$, where

$$[F'] = \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & x \\ \textsc{subcat} & L_x \end{bmatrix} \\ \textsc{cont} & [F] \end{bmatrix} \text{ and } [H'] = \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & y \\ \textsc{subcat} & L_y \end{bmatrix} \\ \textsc{cont} & [H] \end{bmatrix}.$$

Then f maps the CUG-category $\mathrm{X}[F] \underset{[H]}{\backslash} \mathrm{Y}$ onto

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & y \\ \textsc{subcat} & L_y \oplus \left\langle \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & x \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & L_x \end{bmatrix} \\ \textsc{cont} & [F] \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & [H] \end{bmatrix}$$

and $\mathrm{Y} \underset{[H]}{/} \mathrm{X}[F]$ onto

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & y \\ \textsc{subcat} & L_y \oplus \left\langle \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & x \\ \textsc{dir} & \textit{right} \\ \textsc{subcat} & L_x \end{bmatrix} \\ \textsc{cont} & [F] \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & [H] \end{bmatrix}.$$

The following fact holds because the atomic values *left* and *right* only unify with themselves, and the values x and y unify iff $x = y$.

5.2.1. FACT. The CUG-categories $\mathrm{X}\,[F]$ and $\mathrm{Y}\,[H]$ unify iff the AVMs $f(\mathrm{X}\,[F])$ and $f(\mathrm{Y}\,[H])$ unify.

The distinguished AVM. Let S be the distinguished category in CUG, then the distinguished AVM in HPSG is

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \mathrm{S} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \end{bmatrix}.$$

The lexicon. Let the simulation f map CUG-category $\mathrm{X}[F]$ onto the AVM

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & x \\ \textsc{subcat} & L_x \end{bmatrix} \\ \textsc{cont} & [F] \end{bmatrix}.$$

Then the simulation maps the lexicon of the CUG-grammar onto the lexicon of the HPSG-grammar in the following way. If the lexicon of the CUG-grammar G contains entry $w\,:\mathrm{X}[F]$, then the lexicon of the HPSG-grammar $f(G)$ contains the *entry*

$$f(w\,:\mathrm{X}[F]) = \begin{bmatrix} \textsc{phon} & w \\ \textsc{cat} & \begin{bmatrix} \textsc{head} & x \\ \textsc{subcat} & L_x \end{bmatrix} \\ \textsc{cont} & [F] \end{bmatrix},$$

and the lexicon of the HPSG-grammar contains no other entries.

The principles. The combined action of the universal Head Feature Principle (HPF), Subcategorization Principle (SP), Semantics Principle (SemP), and the language specific Immediate Dominance Principle (IDP) and Linear Precedence Principle (LPP) copies the combinatory rules of CUG-grammars. The three universal principles are restated below according to our compact notation. The two language specific principles are defined below in a way that suits the simulation.

Each principle will be represented in two way. The first representation states the principle as a well-formedness condition on AVMs. The second representation expresses the form of the AVM as it is described by the principle.

- The Head Feature Principle is stated as the following condition. If an AVM contains the attribute HDTR, then the value of the paths ⟨CAT HEAD⟩ and ⟨HDTR CAT HEAD⟩ are shared. Thus this principle describes that phrasal AVMs have the following form:

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \boxed{1} \end{bmatrix} \\ \textsc{hdtr} & \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}.$$

- The Subcategorization Principle is stated as the following condition. If an AVM contains the attributes HDTR and CDTRS, then the value of the path $\langle$HDTR CAT SUBCAT$\rangle$ is the concatenation of the value of path $\langle$CAT SUBCAT$\rangle$ and the list of attribute-value pairs with attributes CAT and CONT of the attribute CDTRS. Thus this principle describes that phrasal AVMs have the following form, for some values n and n':

$$\begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{SUBCAT} & \boxed{1} \end{bmatrix} \\ \text{HDTR} & \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{SUBCAT} & \boxed{1} \oplus \left\langle \begin{bmatrix} \text{CAT} & \boxed{2} \\ \text{CONT} & \boxed{3} \end{bmatrix}, \ldots, \begin{bmatrix} \text{CAT} & \boxed{n} \\ \text{CONT} & \boxed{n'} \end{bmatrix} \right\rangle \end{bmatrix} \end{bmatrix} \\ \text{CDTRS} & \left\langle \begin{bmatrix} \text{CAT} & \boxed{2} \\ \text{CONT} & \boxed{3} \end{bmatrix}, \ldots, \begin{bmatrix} \text{CAT} & \boxed{n} \\ \text{CONT} & \boxed{n'} \end{bmatrix} \right\rangle \end{bmatrix}$$

- The Semantics Principle is stated as the following condition. If an AVM contains the attribute HDTR, then the value of the attribute CONT and the path $\langle$HDTR CONT$\rangle$ are shared. Thus this principle describes that phrasal AVMs have the following form:

$$\begin{bmatrix} \text{CONT} & \boxed{1} \\ \text{HDTR} & \begin{bmatrix} \text{CONT} & \boxed{1} \end{bmatrix} \end{bmatrix}.$$

- This language specific version of the Immediate Dominance Principle states that there is only one ID-schema. According to this ID-schema every constituent consists of exactly one mother, one head-daughter and one complement-daughter. Thus this principle describes that constituents, i.e., phrasal AVMs, have the following form, for some AVMs $[F]$ and $[H]$:

$$\begin{bmatrix} \text{HDTR} & [F] \\ \text{CDTRS} & \langle [H] \rangle \end{bmatrix}.$$

- The Linear Precedence Principle describes the ordering of constituents. The actual description of this principle has remained unclear for a long time in HPSG. Lately, Manandhar (1995) provided a feature logic which contains constraints that expresses linear precedence as usually employed in HPSG. The following linear precedence constraints are easily stated in Manandhar's (1995) feature logic. The LPP expresses that a complement-daughter is ordered to the left of the head-daughter if the value of the path $\langle$CAT DIR$\rangle$ in the complement-daughter is *left*, and a complement-daughter is ordered to the right of the head-daughter if the value of the path $\langle$CAT DIR$\rangle$ in the complement-daughter is *right*. In the sequel we assume that a function on the daughters, called *order-constituents*, accomplishes the right ordering. Thus this principle describes that constituents, i.e., phrasal AVMs, have the following form:

$$\begin{bmatrix} \text{PHON} & \textit{order-constituents}(\text{HDTR}, \text{CDTRS}) \end{bmatrix}.$$

Each of the five principles above describes partly the form of a well-formed phrasal AVM. Since a well-formed phrasal AVM satisfies all five principles, the final form of a well-formed phrasal AVM is described by the five principles together. On the one hand, the combined action of the two language specific principles describe that a well-formed phrasal AVM has the following form:

$$\begin{bmatrix} \textsc{phon} & \textit{order-constituents}(\textsc{hdtr}, \textsc{cdtrs}) \\ \textsc{hdtr} & [F] \\ \textsc{cdtrs} & \langle [H] \rangle \end{bmatrix}.$$

On the other hand, the combined action of the three universal principles describe that a well-formed phrasal AVM has the following form, for some values n and n':

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \boxed{1} \\ \textsc{subcat} & \boxed{2} \end{bmatrix} \\ \textsc{cont} & \boxed{0} \\ \textsc{hdtr} & \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \boxed{1} \\ \textsc{subcat} & \boxed{2} \oplus \left\langle \begin{bmatrix} \textsc{cat} & \boxed{3} \\ \textsc{cont} & \boxed{4} \end{bmatrix}, \ldots, \begin{bmatrix} \textsc{cat} & \boxed{n} \\ \textsc{cont} & \boxed{n'} \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{0} \end{bmatrix} \\ \textsc{cdtrs} & \left\langle \begin{bmatrix} \textsc{cat} & \boxed{3} \\ \textsc{cont} & \boxed{4} \end{bmatrix}, \ldots, \begin{bmatrix} \textsc{cat} & \boxed{n} \\ \textsc{cont} & \boxed{n'} \end{bmatrix} \right\rangle \end{bmatrix}.$$

The combined action of the language specific and universal principles describe that a well-formed phrasal AVM has exactly one complement-daughter. Hence the five principles together describe that a well-formed phrasal AVM has the following form, which we will call $[F_P]$.

$$[F_P] = \begin{bmatrix} \textsc{phon} & \textit{order-constituents}(\textsc{hdtr}, \textsc{cdtrs}) \\ \textsc{cat} & \begin{bmatrix} \textsc{head} & \boxed{1} \\ \textsc{subcat} & \boxed{2} \end{bmatrix} \\ \textsc{cont} & \boxed{0} \\ \textsc{hdtr} & \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \boxed{1} \\ \textsc{subcat} & \boxed{2} \oplus \left\langle \begin{bmatrix} \textsc{cat} & \boxed{3} \\ \textsc{cont} & \boxed{4} \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{0} \end{bmatrix} \\ \textsc{cdtrs} & \left\langle \begin{bmatrix} \textsc{cat} & \boxed{3} \\ \textsc{cont} & \boxed{4} \end{bmatrix} \right\rangle \end{bmatrix}$$

This form of an AVM corresponds closely to the combinatory rules of CUG. As stated before, an operational approach is more convenient in the simulation than a declarative approach. Therefore from now on, we will mainly use the description of the form of AVMs, $[F_P]$, to express the effect of the principles.

Summarizing, the HPSG-grammar $f(G)$, which simulates the CUG-grammar G, is now defined in the following way.

5.2.2. Definition. Given a CUG-grammar G, the *simulation* f maps this grammar onto the HPSG-grammar $f(G)$, where $f(G)$ consists of a lexicon; a distinguished AVM, and a description of the form of AVMs that satisfy all the principles.

- The *lexicon* of $f(G)$ is defined as the set of AVMs:

$$\{[H] \mid f(w : X[F]) = [H],\ w : X[F] \text{ a entry from the lexicon of } G\}.$$

We recall that $[H]$ has the following form

$$\begin{bmatrix} \text{PHON} & w \\ \text{CAT} & \begin{bmatrix} \text{HEAD} & x \\ \text{SUBCAT} & L_x \end{bmatrix} \\ \text{CONT} & [F] \end{bmatrix}.$$

- The *distinguished AVM* is defined as

$$\begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD} & \mathrm{S} \\ \text{SUBCAT} & \langle\rangle \end{bmatrix} \end{bmatrix}$$

if S is the distinguished category of G.

- The *combinatory rule* of the grammar is expressed by the AVM $[F_P]$.

Let us now consider a simple example of the simulation. We assume some small CUG-grammar and show for some CUG-categories how they are mapped onto AVMs. Then we present some examples of entries in the lexicon of the simulating HPSG-grammar. We conclude with some examples of AVMs that may be constructed by the simulating HPSG-grammar.

5.2.3. Example. Let there be a CUG-grammar with the ordinary combinatory rules, the distinguished category S and the following lexicon with five entries:

the: $\mathrm{N}_{\boxed{1}[\text{DEF }+]} \;/\; \mathrm{N}\boxed{1}$ a: $\mathrm{N}_{\boxed{1}[\text{DEF }-]} \;/\; \mathrm{N}\boxed{1}[\text{NUM } \textit{sing}]$ kisses: $(\mathrm{N}\boxed{1}\begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{bmatrix} \backslash_{\boxed{1}} \mathrm{S})/\mathrm{N}$

boy: $\mathrm{N}\begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{bmatrix}$ girl: $\mathrm{N}\begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{bmatrix}$

Now the simulation maps the CUG-category $\mathrm{N}_{\boxed{1}[\text{DEF }-]} \;/\; \mathrm{N}\boxed{1}[\text{NUM}\textit{sing}]$ for the string "a" onto the following AVM.

$$\begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD} & \mathrm{N} \\ \text{SUBCAT} & \left\langle \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD} & \mathrm{N} \\ \text{DIR} & \textit{right} \\ \text{SUBCAT} & \langle\rangle \end{bmatrix} \\ \text{CONT} & \boxed{1}[\text{NUM } \textit{sing}] \end{bmatrix} \right\rangle \end{bmatrix} \\ \text{CONT} & \boxed{1}[\text{DEF }-] \end{bmatrix}$$

Similarly, the CUG-category $\mathrm{N}\boxed{1}\begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{bmatrix} \backslash_{\boxed{1}} \mathrm{S}$ is mapped onto the fol-

$$\begin{bmatrix} \textsc{phon} & \textit{the} \\ \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{N} \\ \textsc{subcat} & \left\langle \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{N} \\ \textsc{dir} & \textit{right} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1} \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1}\,[\textsc{def}\ +] \end{bmatrix}$$

$$\begin{bmatrix} \textsc{phon} & \textit{kisses} \\ \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{S} \\ \textsc{subcat} & \left\langle \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{N} \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1} \begin{bmatrix} \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{N} \\ \textsc{dir} & \textit{right} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1} \end{bmatrix}$$

Figure 5.1: Lexicon entries for the strings "the" and "kisses."

lowing AVM.

$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{S} \\ \textsc{subcat} & \left\langle \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \text{N} \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1} \begin{bmatrix} \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1} \end{bmatrix}$$

The lexicon of the CUG-grammar is mapped onto the lexicon of the HPSG-grammar. From the previous examples it is clear that the HPSG-grammar contains the entries given in Figure 5.1 for the words "the" and "kisses."

Given the entries in the lexicon of the HPSG-grammar, we can construct complex AVMs for constituents. These complex AVMs are constructed by unifying the lexical AVMs with the values of the attributes HDTR and CDTRS in the description $[F_P]$.

Let us consider the AVMs that belong to the constituents "the boy" and "a girl." The AVM for "the boy" will be specified in its full detail. The determiner "the" is unified with the value of the attribute HDTR in the description $[F_P]$. As a result of this unification the box-labels in $[F_P]$ receive some values. Box-label $\boxed{1}$ in $[F_P]$ receives the value N; box-label $\boxed{4}$ receives the value $[\textsc{def}\ +]$, and the box-labels $\boxed{0}$ and $\boxed{4}$ in $[F_P]$ are identified; box-label $\boxed{2}$ receives the empty list ($\langle\,\rangle$) as value; and box-labels $\boxed{3}$ receives the value

$$\begin{bmatrix} \textsc{head} & \text{N} \\ \textsc{dir} & \textit{right} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix}.$$

$$\begin{bmatrix}
\text{PHON} & \textit{the boy} \\
\text{CAT} & \begin{bmatrix} \text{HEAD} & \boxed{1}\,\mathrm{N} \\ \text{SUBCAT} & \langle\,\rangle \end{bmatrix} \\
\text{CONT} & \boxed{4}\begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \\ \text{DEF} & + \end{bmatrix} \\
\text{HDTR} & \begin{bmatrix} \text{PHON} & \textit{the} \\ \text{CAT} & \begin{bmatrix} \text{HEAD} & \boxed{1} \\ \text{SUBCAT} & \left\langle \begin{bmatrix} \text{CAT} & \boxed{3}\begin{bmatrix} \text{HEAD} & \mathrm{N} \\ \text{DIR} & \textit{right} \\ \text{SUBCAT} & \langle\,\rangle \end{bmatrix} \\ \text{CONT} & \boxed{4} \end{bmatrix} \right\rangle \end{bmatrix} \\ \text{CONT} & \boxed{4} \end{bmatrix} \\
\text{CDTRS} & \left\langle \begin{bmatrix} \text{PHON} & \textit{boy} \\ \text{CAT} & \boxed{3} \\ \text{CONT} & \boxed{4} \end{bmatrix} \right\rangle
\end{bmatrix}$$

Figure 5.2: Attribute-value matrix for the string "the boy."

In a similar way, the noun "boy" is unified with the AVM described in the value of the attribute CDTRS in the description $[F_P]$. This unification is possible because the value of the attribute CAT in the AVM for "boy" unifies with the box-label $\boxed{3}$. A visible result of the unification is that the box-label $\boxed{4}$ is augmented with the value

$$\begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{bmatrix}.$$

Thus the complete AVM for the string "the boy" is as given in Figure 5.2.

We confine to a simplified AVM for the constituent "a girl" because it is rather similar to the AVM for "the boy."

$$\begin{bmatrix}
\text{PHON} & \textit{a girl} \\
\text{CAT} & \begin{bmatrix} \text{HEAD} & \mathrm{N} \\ \text{SUBCAT} & \langle\,\rangle \end{bmatrix} \\
\text{CONT} & \begin{bmatrix} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \\ \text{DEF} & - \end{bmatrix} \\
\text{HDTR} & [\text{PHON}\ \textit{a}] \\
\text{CDTRS} & \langle[\text{PHON}\ \textit{girl}]\rangle
\end{bmatrix}$$

Let us now consider simplified AVMs that belong to the constituents "kisses a girl" and "the boy kisses a girl." In these cases the AVMs for the constituents "kisses," "a girl," "the boy" and "kisses a girl" are unified with the values of the attributes HDTR and CDTRS.

$$
\begin{bmatrix}
\textsc{phon} & \textit{kisses a girl} \\
\textsc{cat} & \begin{bmatrix} \textsc{head} & \mathrm{S} \\ \textsc{subcat} & \left\langle \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & \mathrm{N} \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \\ \textsc{cont} & \boxed{1} \begin{bmatrix} \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{bmatrix} \end{bmatrix} \right\rangle \end{bmatrix} \\
\textsc{cont} & \boxed{1} \\
\textsc{hdtr} & [\textsc{phon}\ \textit{kisses}] \\
\textsc{cdtrs} & \langle [\textsc{phon}\ \textit{a girl}] \rangle
\end{bmatrix}
\quad
\begin{bmatrix}
\textsc{phon} & \textit{the boy kisses a girl} \\
\textsc{cat} & \begin{bmatrix} \textsc{head} & \mathrm{S} \\ \textsc{subcat} & \langle\,\rangle \end{bmatrix} \\
\textsc{cont} & \begin{bmatrix} \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \\ \textsc{def} & + \end{bmatrix} \\
\textsc{hdtr} & [\textsc{phon}\ \textit{kisses a girl}] \\
\textsc{cdtrs} & \langle [\textsc{phon}\ \textit{the boy}] \rangle
\end{bmatrix}
$$

The reader should notice that the AVM for "the boy kisses a girl" has the form that corresponds to the distinguished CUG-category S. Hence this AVM describes a complete string.

5.3 Correctness of the Simulation

The main purpose of this section is to prove the correctness of the simulation. That is, CUG-grammar G and HPSG-grammar $f(G)$ generate the same language. We will now define the notions "derives," "produces," and "generates" for HPSG. These three notions are not discussed in HPSG, so we will have to define them ourselves. The definitions will be convenient in the proofs of the correctness of the simulation.

5.3.1 Definitions

We will assume that complex AVMs are built from simpler AVMs, where the simplest AVMs are the entries in the lexicon. The AVMs are built as prescribed by the principles of the grammar. Hence all complex AVMs have the form of AVM $[F_P]$. Now we come to the following definition of the notions "derives" and "produces," which are rather similar to Definition 3.3.11 and Definition 3.3.12.

5.3.1. Definition. Let G be an HPSG-grammar, and let $[F]$ be an AVM. The AVM $[F]$ *derives* AVM $[H]$ iff the unification of $[F]$ with $[F_P]$ yields $[H']$, where the values of the attributes HDTR and CDTRS in $[H']$ are $[F_0]$ and $\langle[F_1]\rangle$, and the values of the attributes HDTR and CDTRS in $[H]$ are $[F'_0]$ and $\langle[F'_1]\rangle$, where either $[F_i]$ derives $[F'_i]$, or $[F_i] = [F'_i]$.

5.3.2. Definition. Let G be an HPSG-grammar, w be a string, and $[F]$ be an AVM that contains attribute PHON with value w. The AVM $[F]$ *produces* string w iff $[F]$ derives some AVM $[H]$, and all terminal AVMs of $[H]$ are subsumed by entries in the lexicon of G.

5.3.3. Definition. The AVM $[H]$ is a *terminal AVM* of AVM $[F]$ iff

1. $[F] = [H]$, and $[F]$ does not contain attribute HDTR or CDTRS, or
2. $[F]$ contains attribute HDTR and CDTRS with values $[F_0]$ and $\langle [F_1] \rangle$, and $[H]$ is a terminal AVM of some $[F_i]$.

The HPSG-literature indicates that the strings of the language described by a given HPSG-grammar all stem from the same category. The introduction of a distinguished AVM in Section 5.1 plays the role of this category. The definitions of the notions "generates" and "language" resemble Definition 3.3.13.

5.3.4. DEFINITION. HPSG-grammar G *generates* string w iff the unification of the AVM [PHON w] with the distinguished AVM of G produces string w. The *language* that grammar G generates ($L(G)$) is defined as the set of all strings that G generates.

5.3.2 Proof of Correctness

Given the definitions above, we prove the correctness of the simulation in Theorem 5.3.9. Lemmas 5.3.5 and 5.3.6 show that one single application of a combinatory rule for a CUG-grammar G can be simulated by the HPSG-grammar $f(G)$. Lemma 5.3.8 shows that CUG-categories and the corresponding AVMs in HPSG produce the same strings.

The following lemma states roughly that an HPSG-constituent with mother $[F_3]$, left complement-daughter $[F_1]$ and right head-daughter $[F_2]$ can be constructed iff a CUG-constituent with mother $\mathrm{Y}\,[H]$, left daughter $\mathrm{X}\,[F]$ and right daughter $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ can be constructed.

5.3.5. LEMMA. *Let the simulation map* $\mathrm{X}\,[F]$ *onto* $[F_1]$, $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ *onto* $[F_2]$, *and* $\mathrm{Y}\,[H]$ *onto* $[F_3]$. *CUG-category* $\mathrm{Y}\,[H]$ *derives the sequence* $\mathrm{X}\,[F]\ (\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ *by left application iff the AVM* $[F_0]$ *derives* $[F_{app}]$, *where*

$$[F_0] = \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{HEAD} & y \\ \text{SUBCAT} & L_y \end{bmatrix} \\ \text{CONT} & [H] \\ \text{HDTR} & [F_2] \\ \text{CDTRS} & \langle [F_1] \rangle \end{bmatrix}$$

and

$$[F_{app}] = \left[\begin{array}{ll}
\textsc{phon} & \textit{order-constituents}(\textsc{hdtr}, \textsc{cdtrs}) \\
\textsc{cat} & \left[\begin{array}{ll} \textsc{head} & \boxed{1}\, y \\ \textsc{subcat} & \boxed{2} \end{array}\right] \\
\textsc{cont} & \boxed{0}\, [H] \\
\textsc{hdtr} & \left[\begin{array}{ll}
\textsc{cat} & \left[\begin{array}{ll} \textsc{head} & \boxed{1}\, v \\ \textsc{subcat} & \boxed{2} \oplus \left\langle \left[\begin{array}{ll} \textsc{cat} & \boxed{3} \left[\begin{array}{ll} \textsc{head} & u \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & L_u \end{array}\right] \\ \textsc{cont} & \boxed{4}\, [F'] \end{array}\right] \right\rangle \end{array}\right] \\
\textsc{cont} & \boxed{0}\, [H']
\end{array}\right] \\
\textsc{cdtrs} & \left\langle \left[\begin{array}{ll} \textsc{cat} & \boxed{3} \left[\begin{array}{ll} \textsc{head} & x \\ \textsc{subcat} & L_x \end{array}\right] \\ \textsc{cont} & \boxed{4}\, [F] \end{array}\right] \right\rangle
\end{array}\right]$$

Proof. The simulation maps $\mathrm{X}\,[F]$ onto $[F_1]$, $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ onto $[F_2]$, and $\mathrm{Y}\,[H]$ onto $[F_3]$, where

$$\begin{array}{lcl}
[F_1] & = & \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{head} & x \\ \textsc{subcat} & L_x \end{array}\right] \\ \textsc{cont} & [F] \end{array}\right] \\
[F_2] & = & \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{head} & v \\ \textsc{subcat} & L_v \oplus \left\langle \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{head} & u \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & L_u \end{array}\right] \\ \textsc{cont} & [F'] \end{array}\right] \right\rangle \end{array}\right] \\ \textsc{cont} & [H'] \end{array}\right] \\
[F_3] & = & \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{head} & y \\ \textsc{subcat} & L_y \end{array}\right] \\ \textsc{cont} & [H] \end{array}\right]
\end{array}$$

Only if: Assume that by means of a left application CUG-category $\mathrm{Y}\,[H]$ derives the sequence $\mathrm{X}\,[F]\ \ (\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$. Then the unification of $\mathrm{X}\,[F]$ with $\mathrm{U}[F']$ succeeds and turns $\mathrm{V}[H']$ into $\mathrm{Y}\,[H]$.

The CUG-categories $\mathrm{X}\,[F]$ and $\mathrm{U}[F']$ unify iff the AVMs $f(\mathrm{X}\,[F])$ and $f(\mathrm{U}[F'])$ unify. This implies that $[F]$ and $[F']$ unify, and the following two AVMs unify.

$$\left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{head} & x \\ \textsc{subcat} & L_x \end{array}\right] \\ \textsc{cont} & [F] \end{array}\right] \text{ and } \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{head} & u \\ \textsc{dir} & \textit{left} \\ \textsc{subcat} & L_u \end{array}\right] \\ \textsc{cont} & [F'] \end{array}\right]$$

Because $\mathrm{V}[H']$ turns into $\mathrm{Y}\,[H]$, $\mathrm{V}[H']$ unifies with $\mathrm{Y}\,[H]$. Therefore $f(\mathrm{V}\,[H'])$ and $f(\mathrm{Y}\,[H])$ unify. Hence the unification of $[F_0]$ with $[F_P]$ yields $[F_{app}]$.

If: Assume that the AVM $[F_0]$ derives AVM $[F_{app}]$. The left application of $\mathrm{X}\,[F]$ and $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ yields $\mathrm{Y}\,[H]$ iff the unification of $\mathrm{X}\,[F]$ with $\mathrm{U}\,[F']$ turns $\mathrm{V}\,[H']$ into $\mathrm{Y}\,[H]$.

From the box-labels $\boxed{3}$ and $\boxed{4}$ in the AVM $[F_{app}]$ it follows that x unifies with u, L_x unifies with L_u and $[F]$ unifies with $[F']$. So $\mathrm{X}\,[F]$ unifies with $\mathrm{U}\,[F']$. Furthermore, because $[F_0]$ derives AVM $[F_{app}]$ and by the box-labels $\boxed{0}$ and $\boxed{1}$, it follows that both $[H']$ and v can be extended such that $v = y$ and $[H'] = [H]$. So $\mathrm{V}\,[H']$ turns into $\mathrm{Y}\,[H]$. □

The following lemma states roughly that an HPSG-constituent with mother $[F_3]$, right complement-daughter $[F_1]$, and left head-daughter $[F_2]$ can be constructed. iff a CUG-constituent with mother $\mathrm{Y}\,[H]$, right daughter $\mathrm{X}\,[F]$, and left daughter $\mathrm{V} \underset{[H']}{/} \mathrm{U}[F']$ can be constructed

5.3.6. LEMMA. *Let the simulation map* $\mathrm{Y}\,[H]$ *onto* $[F_3]$, $\mathrm{V} \underset{[H']}{/} \mathrm{U}[F']$ *onto* $[F_2]$, *and* $\mathrm{X}\,[F]$ *onto* $[F_1]$. *CUG-category* $\mathrm{Y}\,[H]$ *derives the sequence* $(\mathrm{V} \underset{[H']}{/} \mathrm{U}[F'])\ \mathrm{X}\,[F]$ *by right application iff the AVM* $[F_0]$ *derives* $[F_{app}]$, *with* $[F_{app}]$ *as given in Lemma 5.3.5 and*

$$[F_0] = \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{head} & y \\ \textsc{subcat} & L_y \end{bmatrix} \\ \textsc{cont} & [H] \\ \textsc{hdtr} & [F_1] \\ \textsc{cdtrs} & \langle [F_2] \rangle \end{bmatrix}$$

Proof. Similar to the proof of Lemma 5.3.5 □

5.3.7. FACT. AVM $[F]$ produces string $w_1 \ldots w_n$ iff $[F]$ derives AVM $[H]$ whose terminal AVMs are subsumed by the entries for the strings $w_1 \ldots w_n$ iff there are lexicon entries for the strings $w_1 \ldots w_n$, $[G_1] \ldots [G_n]$, and there is a sequence of pair-wise distinct paths, $p_1 \ldots p_{2n-1}$ (where p_{2n-1} is the empty path), such that the unification of $[F]$ with all $[p_i\,[G_i]]$ and $[p_j\,[F_P]]$ $(1 \leq i \leq n,\ n < j \leq 2n-1)$ results in $[H]$.

5.3.8. LEMMA. *Given a CUG-grammar G. Let w be a string and* $\mathrm{Y}\,[H]$ *be a CUG-category. CUG-category* $\mathrm{Y}\,[H]$ *produces string w iff the AVM* $f(w : \mathrm{Y}\,[H])$ *produces string w.*

Proof. By induction on the length of the string w.

String w is in the lexicon. The CUG-lexicon contains entry $w : \mathrm{Y}\,[H]$ iff the HPSG-lexicon contains entry $f(w : \mathrm{Y}\,[H])$. Hence $\mathrm{Y}\,[H]$ produces w iff the AVM $f(w : \mathrm{Y}\,[H])$ produces string w.

String w is not in the lexicon. Let $w = w_1 w_2$, $w_1 = w'_1 \ldots w'_n$, and $w_2 = w''_1 \ldots w''_m$. By induction $\mathrm{U}\,[F']$ produces w_1 iff $f(w_1 : \mathrm{U}\,[F'])$ produces w_1, and $\mathrm{V}\,[H']$ produces w_2 iff $f(w_2 : \mathrm{V}\,[H'])$ produces w_2. Moreover, CUG-category $\mathrm{Y}\,[H]$ derives sequence $\mathrm{U}\,[F']\ \ \mathrm{V}\,[H']$ iff AVM $[F_0]$ derives $[F_{app}]$, as in Lemmas 5.3.5 and 5.3.6.

By the previous fact, there are sequences $[G'_1] \ldots [G'_n]$ and $p'_1 \ldots p'_{2n-1}$ such that $f(w_1 : \mathrm{U}\,[F'])$ unified with $[p'_i\,[G'_i]]$ and $[p'_j\,[F_P]]$ results in $[H'_1]$ whose terminal AVMs are subsumed by the entries for the strings $w'_1 \ldots w'_n$. Likewise, there are sequences $[G''_1] \ldots [G''_m]$ and $p''_1 \ldots p''_{2m-1}$ such that $f(w_2 : \mathrm{V}\,[H'])$ unified with $[p''_i\,[G''_i]]$ and $[p''_j\,[F_P]]$ results in $[H'_2]$ whose terminal AVMs are subsumed by the entries for the strings $w''_1 \ldots w''_m$. From either Lemma 5.3.5 or Lemma 5.3.6 it follows that given the sequence of AVMs $[G'_1] \ldots [G'_n], [G''_1] \ldots [G''_m]$ and the sequence of paths $q'_1 \ldots q'_n$ $q''_1 \ldots q''_m$ $q_1 \ldots q_{n+m-1}$ the unification of $f(w : \mathrm{Y}\,[H])$ with all $[q'_i\,[G'_i]]$, $[q''_i\,[G''_i]]$ and $[q_i\,[F_P]]$ results in $[H'_0]$ whose terminal AVMs are subsumed by the entries for the strings $w'_1 \ldots w'_n\, w''_1 \ldots w''_m$. And hence $\mathrm{Y}\,[H]$ produces w iff $f(w : \mathrm{Y}\,[H])$ produces w.

According to Lemma 5.3.5 the path q_{n+m-1} is the empty path, $q'_i = \langle \textsc{cdtrs}\, p'_i \rangle$, $q''_i = \langle \textsc{hdtr}\, p''_i \rangle$, $q_j = \langle \textsc{cdtrs}\, p'_{n+j} \rangle$ $(1 \le j < n)$, and $q_{n+j} = \langle \textsc{hdtr}\, p''_{m+j+1} \rangle$ $(0 \le j < m-1)$.

According to Lemma 5.3.6 the path q_{n+m-1} is the empty path, $q'_i = \langle \textsc{hdtr}\, p'_i \rangle$, $q''_i = \langle \textsc{cdtrs}\, p''_i \rangle$, $q_j = \langle \textsc{hdtr}\, p'_{n+j} \rangle$ $(1 \le j < n)$, and $q_{n+j} = \langle \textsc{cdtrs}\, p''_{m+j+1} \rangle$ $(0 \le j < m-1)$. □

5.3.9. Theorem. *Given a CUG-grammar G and string w. Let the simulation map G onto HPSG-grammar $f(G)$. CUG-grammar G generates string w iff HPSG-grammar $f(G)$ generates string w*

Proof. Straightforward generalization of Lemma 5.3.8. □

5.4 Formal Properties

We consider two formal properties. First, we handle the complexity of the recognition problems of HPSG. Second, we handle the weak generative capacity of HPSG.

5.4.1 Complexity of the Recognition Problems

We determine the complexity of the recognition problems in two steps. First, we will provide an *NP*-hard lower bound on the complexity of the recognition problem for a fixed grammar. Second, we will provide an *NP* upper bound on the complexity of the universal recognition problem. These two bounds together determine the complexity of the recognition problems exactly.

A lower bound on the complexity. A lower bound on the complexity of the recognition problems results easily from the simulation by the following argumentation. Given any CUG-grammar, the simulation f provides us with an HPSG-grammar that recognizes the same language as the CUG-grammar. Thus the mapping from a CUG-grammar G onto an HPSG-grammar $f(G)$ is a many-one reduction. Moreover, the next lemma proves that the mapping is a polynomial time, many-one reduction. Hence Theorems 5.4.2 and 5.4.3 follow immediately.

5.4.1. LEMMA. *The HPSG-grammar $f(G)$ that simulates a CUG-grammar G is computed in an amount of steps that is linear with respect to the size of the CUG-grammar.*

Proof. The mapping from the lexicon of the CUG-grammar onto the lexicon of the HPSG-grammar has the largest impact on the cost of the computation of the HPSG-grammar. This mapping depends mainly on the mapping from CUG-categories onto AVMs in HPSG. This latter mapping costs an amount of time that is linear in the size of the CUG-category. So the mapping from the CUG-lexicon costs linear time with respect to the size of the CUG-grammar. Hence the HPSG-grammar is computed in an amount of steps that is linear with respect to the size of the CUG-grammar. $\square$

5.4.2. THEOREM. *The recognition problem of HPSG for a fixed grammar is NP-hard.*

Proof. In Section 3.3.2 we presented a CUG-grammar G whose recognition problem is *NP*-hard (Theorem 3.3.22). The simulation maps this CUG-grammar onto a HPSG-grammar $f(G)$. Lemma 5.4.1 shows that the HPSG-grammar $f(G)$ is computed in polynomial time. Furthermore, Theorem 5.3.9 shows that the CUG-grammar G and the HPSG-grammar $f(G)$ recognize the same language. Hence the recognition problem of the HPSG-grammar $f(G)$ is *NP*-hard. $\square$

5.4.3. THEOREM. *The universal recognition problem of HPSG is NP-hard.*

Proof. The universal recognition problem is at least as hard as the recognition problem for a fixed grammar. Theorem 5.4.2 shows that the recognition problem for a fixed grammar is *NP*-hard. Hence the universal recognition problem of HPSG is *NP*-hard. $\square$

An upper bound on the complexity. Theorems 5.4.2 and 5.4.3 present lower bound on the complexity of the recognition problems of HPSG. Obviously, we would also like to have an upper bound on the complexity of the recognition problems. In the ideal situation we would provide an upper bound for general versions of HPSG. A prerequisite for such an upper bound is that a complete description of HPSG is available. Alas no complete description of HPSG exists. So we have to settle for the practical situation, and consider fragments of HPSG. Let us first consider the fragment of HPSG presented in Section 5.1. Then we will consider polynomial extensions of this fragment, and argue that these extensions suffice to describe full HPSG-grammars.

In Section 5.1 we showed that this fragment of HPSG is based on the standard feature theory from Section 3.2. This standard feature theory has a nice computational property: the unification problem is tractable. We indicated in Section 3.2.2 that minor changes to Smolka's (1992) constraint solving algorithm yield a polynomial time unification algorithm for AVMs. To be more precise, the unification of two AVMs $[F]$ and $[H]$ is computable in polynomial time, with respect to the sizes of $[F]$ and $[H]$. Moreover, the amount of space used by the unification is linear with respect to the sizes of $[F]$ and $[H]$.

Now we will prove an *NP* upper bound on the complexity of the universal recognition problem of this restricted fragment of HPSG. The instances of the universal recognition problem are an HPSG-grammar G and a string w (see Definition 3.2.2).

In Section 5.2 we represented principles by the form of the AVMs that they describe. We showed that the combined action of principles and schemas can also be expressed by the form of the AVMs that they describe. In Section 5.2 this combined action was expressed by the form, $[F_P]$. In general this combined action may result in a bounded set of such forms for AVMs, G_P. These forms of AVMs from the set G_P will be used to express the effect of the principles and schemas. Obviously, the forms in G_P may only describe the forms of AVMs from the standard feature theory. So given the sort hierarchy, the HPSG-grammar G can be described by a set of lexical AVMs, G_L, a set of AVMs that encodes the combined action of principles and schemas, G_P, and a distinguished AVM.

Now we will call an AVM from the set G_P a *rule* iff the AVM contains the attribute DTRS. We call an AVM a k-ary rule iff the AVM contains the attribute DTRS, and the value of this attribute describes exactly k daughter constituents, e.g., one head-daughter and $k-1$ complement-daughters. A *unary* (1-ary) rule contains the attribute DTRS, and the value of this attribute describes exactly one daughter constituent. We say that a derivation contains a *detour* if the derivation contains a sequence of unary rules, and some rule occurs twice in this sequence. Probably, if detours are allowed in derivations, the recognition problem of HPSG will become undecidable. In order to guarantee a decidable fragment of HPSG we forbid detours in derivations.

The proof of the *NP* upper bound is based on the following two observations. First, a derivation for a string w and an HPSG-grammar G consists of a polynomial amount of steps. This first observation is proven in Lemma 5.4.4. Second, the reverse of a derivation step is computable in polynomial time. This second observation is proven in Lemma 5.4.5.

5.4.4. LEMMA. *A derivation for a string w and a grammar G has polynomial size, with respect to the sizes of w and G.*

Proof. Because we postulated a ban on detours in derivations, every derivation must combine two AVMs after at most an amount of steps that is linear with respect to the size of G_P. Hence the total derivation for a string w consists of at most a linear amount of steps with respect to the size of the string and the size of the grammar. □

5.4.5. LEMMA. *Given a k-ary rule, $[G']$, from G_P and k AVMs, $[F_1] \ldots [F_k]$. The corresponding reversed derivation step is computable in polynomial time, with respect to the sizes of $[F_1] \ldots [F_k]$ and $[G']$.*

Proof. Given an AVM $[G']$ from G_P of which the attribute DTRS describes k daughter constituents. The reversed derivation step consists of the unification of the k AVMs $[F_i]$ with the daughter constituents in $[G']$. This takes k unification steps. Hence this reversed derivation step takes an amount of time that is polynomial in the size of the rule and the sizes of the daughter AVMs. □

Now given a string w and an HPSG-grammar G, we guess a linear amount of entries from the lexicon, and a polynomial sized sequence of rules that encode the derivation for w. The previous two lemmas then show that both guesses can be checked in polynomial time. Hence the following lemma holds.

5.4.6. LEMMA. *The universal recognition problem of HPSG for string w and HPSG-grammar G is computable in nondeterministic polynomial time.*

Proof. A derivation for the string w is described by a sequence of lexicon entries and a sequence of applied rules. So we guess a sequence of entries from the lexicon of G, for the string w. These entries have a total size that is linear in the size of the string and the grammar. Next we guess a sequence of rules that complete the description of the derivation for w. This sequence consists of a polynomial amount of rules, by Lemma 5.4.4.

Now in order to check whether the guesses result in a derivation for the string w, we compute, in reversed order, the derivation from these guesses. Lemma 5.4.5 states that this computation takes polynomial time, with respect to the size of guessed entries and rules, which is polynomial in the size of string w and HPSG-grammar G.

All that remains is to check that the distinguished AVM starts the derivation and the string w results from the derivation. The only point worth noticing is that we assume that the function *order-constituents*, which orders the constituents, is computable in polynomial time. This assumption is legitimate in the practical cases, i.e., the cases where *order-constituents* yields a permutation of the values of the attributes PHON of the daughter constituents (Manandhar 1995). □

Let us now consider extensions of the fragment of HPSG that we presented in Section 5.1. We consider "polynomial" extensions in which the reversed application of derivation steps remain computable in polynomial time, and in which the derivations remain of polynomial size. Clearly, the recognition problems of these extensions are also computable in nondeterministic polynomial time. We claim that these polynomial extensions may contain

1. all universal principles;
2. all language specific principles that occur in actual practice;
3. the use of the structured values set;
4. the disjunctive values and AVMs.

All actual principles of HPSG are stated as simple relations between the mothers and the daughters of constituents. The theory of HPSG presents itself as a theory in which unification is the only operation needed. So the effect of a principle is computable by unification with the AVMs on which the principle is applied. Therefore, we think that it is legitimate to demand that full HPSG-grammars are polynomial extensions of the fragment presented in Section 5.1. Hence by Theorems 5.4.2 and 5.4.3:

5.4.7. THEOREM.
(*i*) *The fixed recognition problem of HPSG is NP-complete.*
(*ii*) *The universal recognition problem of HPSG is NP-complete.*

5.4.2 Weak Generative Capacity

We will now determine a lower bound and an upper bound on the weak generative capacity of HPSG. The lower bound results from the simulation. The upper bound results from the upper bound of the complexity of the fixed recognition problem. These two bounds on the weak generative capacity are represented in Theorem 5.4.8.

A lower bound on the weak generative capacity. Theorem 5.3.9 proves that for every CUG-grammar there exists an HPSG-grammar that recognizes the same language. Hence the set of languages recognized by a CUG-grammar, the *CUG-languages*, is a subset of the set of languages recognized by an HPSG-grammar, the *HPSG-languages*. So if we can prove the location of the CUG-languages in the Chomsky hierarchy, we know that the HPSG-languages take the same or a higher location in the Chomsky hierarchy. For instance, it is known that the set of languages that the classical categorial grammars generate is the set of context-free languages. Therefore the CUG-languages generate all context-free languages. Hence the HPSG-languages generate all context-free languages.

An upper bound on the weak generative capacity. Once again, we are unable to provide an upper bound for general versions of HPSG. A prerequisite for such an upper bound is a complete description of HPSG, which is not available. So once again we settle for the fragment of HPSG presented in Section 5.1.

In Chapter 2 we connect the complexity of the recognition problem and weak generative capacity of restricted attribute-value grammars. These restricted attribute-value grammars respect a liberal variation on the so-called *off-line parsability constraint*. This constraint (Johnson 1988) is a well-known generalization of the "Definition of a Valid Derivation" from Lexical Functional Grammar (Kaplan and Bresnan 1982). The off-line parsability constraint relates the amount of "work" done by the grammar to produce a string linearly to the number of terminal symbols produced. It is therefore a sort of honesty constraint that is common in complexity theory.

In Chapter 2 a variation on the off-line parsability constraint is introduced: the *honest parsability constraint*. This honest parsability constraint is a more liberal constraint than the off-line parsability constraint. The honest parsability constraint relates the amount of "work" done by the grammar to produce a string polynomially to the number of terminal symbols produced. For convenience, we state the definition of the honest parsability constraint, Definition 2.5.1, below.

> *A grammar G satisfies the honest parsability constraint iff there exists a polynomial p such that for each string w in the language generated by G there exists a derivation with at most $p(|w|)$ steps.*

A nice property of the restricted attribute-value grammars that respect the honest parsability constraint is that they generate exactly all languages in the complexity class *NP*. Or, as we can restate Theorem 2.5.3:

> *Let L be a language that has an NP recognition algorithm. Then there exists a restricted attribute-value grammar G, that respects the honest parsability constraint, such that G generates the language L.*

By Theorem 5.4.7 the restricted fragment of HPSG has an *NP* recognition problem. Hence any restricted HPSG-grammar can be simulated by a restricted attribute-value grammar. Thus the languages generated by restricted attribute-value grammars that respect the honest parsability constraint provide an upper bound on the weak generative capacity of restricted HPSG-grammars. Hence the following theorem holds.

5.4.8. THEOREM. *The weak generative capacity of the restricted fragment of HPSG presented in this chapter is enclosed between the context-free languages and the collection of languages generated by restricted attribute-value grammars that respect the honest parsability constraint.*

Moreover, we conjecture a stronger upper bound for the weak generative capacity of HPSG. A close look at the proof of Lemma 5.4.6 shows that the fixed recognition problem of HPSG is computable in nondeterministic linear space. We conjecture that the polynomial extensions of the fragment presented in Section 5.1 are also computable in nondeterministic linear space. This implies that the HPSG-languages are recognized by linear bounded automata (LBAs). Due to the equivalence of LBAs and context-sensitive grammars, HPSG generates only context-sensitive languages (CSLs). Hence we conjecture that the collection of CSLs that are recognized in nondeterministic polynomial time form an upper bound for the weak generative capacity of HPSG (cf., Book 1978). Or stated differently:

5.4.9. CONJECTURE. *The collection of context-sensitive languages that the restricted attribute-value grammars that respect the honest parsability constraint generate form an upper bound for the weak generative capacity of HPSG.*

Chapter 6

Lexical Functional Grammar

In this chapter we will present a formulation of Lexical Functional Grammar (LFG). The chapter is based on the original formulation of Kaplan and Bresnan (1982) and the extended survey of Sells (1985). The main purpose of this chapter is to show that LFG-grammars can simulate any arbitrary CUG-grammar that was presented in Section 3.3. This simulation provides a way of comparing the different grammatical formalisms. For instance, the simulation shows that the generative capacity of LFG is at least as high as the generative capacity of CUG. Moreover, because the simulation is computable in polynomial time, it is a polynomial time many-one reduction. Therefore the recognition problem of LFG is at least as hard as the *NP*-complete recognition problem of CUG.

Section 6.1 contains an informal introduction in LFG and discusses which part of LFG are needed for a simulation of CUG. In Section 6.2 we present the simulation of CUG. In Section 6.3 we prove the correctness of the simulation. We conclude with Section 6.4, in which we discuss the weak generative capacity of LFG and the complexity of its recognition problem. We compare our results with the results of Barton, Berwick and Ristad (1987), Nakanishi et al. (1992), and Seki et al. (1993).

6.1 Introduction in LFG

This section contains an informal introduction in Lexical Functional Grammar (LFG). LFG distinguishes two components for the syntactic description of sentences: the c-structure and the f-structure. The c-structure is a conventional constituent structure tree. A c-structure is defined in terms of nonterminals, terminal strings, and dominance and precedence relationships. The f-structures are the descriptions that we have called attribute-value matrices (AVMs) in this thesis. An f-structure, henceforth AVM, provides a characterization of such notions like "subject," "object," "complement," and "adjunct."

LFG contains combinatory rules to combine the c-structure and the f-structure components. These rules states how constituents are formed, and how AVMs are related to these constituents. The rules are annotated, extended context-free rewrite rules. These annotations are statements that express typically how feature information is passed from mother nonterminal to daughter nonterminal.

Thus an LFG-grammar generates both a constituent structure tree and a collection of statements for a string. Such a collection of statements, which is called an *f-description*, specifies various properties of the AVM of the string. The f-description of the string is an intermediate between the constituent structure tree and the AVM of the string. The statements of the f-description can be used in a constructive and in a declarative manner. That is, the statements can be used to construct an AVM that satisfies the properties required by the grammar, and can be used to decide whether or not a given AVM has all the properties required by the grammar.

The further introduction of LFG is split in two parts. First, we compare the f-descriptions and AVMs from LFG with the standard feature theory from Section 3.2. We will also indicate the parts that are needed for the simulation of CUG. Second, we compare the production processes of LFG and CUG. This comparison will reveal some notions of LFG that are important for the simulation.

6.1.1 Comparison with the Standard Feature Theory

The f-descriptions and attribute-value matrices of Lexical Functional Grammar differ in four aspects from the standard feature theory.

- The combinatory rules to LFG govern the combination and unification of the f-descriptions and AVMs. These rules are annotated, extended context-free rewrite rules. The rules extend context-free rules in expressing that a nonterminal occurs optionally, or an arbitrary number of times. The annotations are equations that describe a partial AVM. See 1 in Example 6.1.1 for a simplified combinatory rule. For the simulation it suffices to consider annotated, ordinary context-free rewrite rules.
- Besides the atomic and compound values also structured and special values are permitted in AVMs and f-descriptions. The structured values are sets, which are used to treat adjuncts (e.g., Kaplan and Bresnan 1982, 215 ff.). The special values, which are restricted to the attribute PRED, are the semantic forms. These semantic forms specify both the semantic interpretation and the grammatical functions of an AVM. In the simulation, however, these structured and special values are not needed.
- The most salient difference between AVMs in LFG and the standard feature theory is the way reentrance is expressed. In the standard feature theory reentrance in an AVM is denoted by path-equations or box-labels. In LFG however, reentrance in an AVM is denoted by lines. These lines link the endpoint of the paths that share their values. In the simulation we will use box-labels instead of these lines. The similarity between lines and box-labels is illustrated by 2 in Example 6.1.1.
- In LFG also some issues are introduced which seem to be not more than notational sugar. On these issues we take a different stand than the common theory. A first shortcut is found in the f-descriptions. For some feature information,

e.g., the feature information that denotes case, a distinction is made between the marked and the unmarked situation. When an f-description does not specify this feature information explicitly, the unmarked situation is assumed (cf., Sells 1985, p. 145). In the simulation we ignore the distinction between marked and unmarked situations. We demand that all feature information is specified explicitly.
A second shortcut is also found in the f-descriptions. This shortcut seems to be introduced because nonterminals may occur an arbitrary number of times in a combinatory rules. The shortcut allows one to state that the atomic value of some attribute in an AVM also exists as an attribute in an AVM (cf., Kaplan and Bresnan 1982, p. 197). Because this shortcut seems to be restricted to atomic values, this shortcut corresponds to finitely many situations. These shortcuts are not needed in the simulation.

6.1.1. EXAMPLE.

1. The following simplified rule serves to illustrate the combinatory rules. All nonterminals in this rule are obligatory. The annotation of the rule consists of equations, indicated by $\doteq$, that are linked to some nonterminal on the right-hand side. The equations contain metavariables in the shape of arrows. An upward arrow refers to the AVM of the mother nonterminal, the *VP*. A downward arrow refers to the AVM of the nonterminal to which the equation is linked, the *V* and *NP*'s. Intuitively, the equations state that the AVM of the *V* and the *VP* are the same. Furthermore, this AVM contains attributes OBJ and OBJ2 with as value the AVMs of the first and second *NP*, respectively.

$$\begin{array}{llll} VP \rightarrow & V & NP & NP \\ & \uparrow \doteq \downarrow & (\uparrow \textsc{obj}) \doteq \downarrow & (\uparrow \textsc{obj}2) \doteq \downarrow \end{array}$$

   Formally the equations state the following. Let f_0 be a variable that corresponds to the AVM of the *VP*, f_1 correspond to the AVM of the *V*, and f_2 and f_3 correspond to the AVM of the first and second *NP*, respectively. Then this rule yield the following three f-description statements: $f_0 \doteq f_1$, $(f_0\,\textsc{obj}) \doteq f_2$, $(f_0\,\textsc{obj}2) \doteq f_3$.
   Fully worked-out examples how an f-description for a string is produced and how the f-description is used to construct an AVM are found in (Kaplan and Bresnan 1982, 183 ff.)
2. Consider the following simplified entry from the lexicon for the verb "persuade." The entry consists of a string, a nonterminal, and an annotation.

$$\text{persuade} : V, (\uparrow \textsc{vcomp subj}) \doteq (\uparrow \textsc{obj}).$$

   This entry would give rise to an AVM in which the paths $\langle$VCOMP SUBJ$\rangle$ and $\langle$OBJ$\rangle$ share their values. In the LFG-literature, this AVM would contain a line that connects the values of the two paths.

$$\begin{bmatrix} \textsc{obj} & [\;] \\ \textsc{vcomp} & \begin{bmatrix} \textsc{subj} & [\;] \end{bmatrix} \end{bmatrix}$$

In the simulation we will express this reentrance by box-labels. Box-labels with equal indices correspond to the endpoint of the same line.

$$\begin{bmatrix} \text{OBJ} & \boxed{1} \\ \text{VCOMP} & \begin{bmatrix} \text{SUBJ} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

6.1.2 Comparison with CUG

Let us now compare the production processes of LFG and CUG. Three differences will be mentioned.

1. The finiteness of constituent structure trees for strings is achieved differently in CUG and LFG. CUG one the one hand is a lexicon based formalism. That is, the entries in the lexicon determine how words are combined, whereas the rules of CUG-grammars are fixed. Consequently, a CUG-grammar assigns binary branching constituent structure trees to strings. This binary branching ensures that constituent structure trees for strings are finite. LFG on the other hand is a rule based formalism. That is, the lexicon is largely fixed, whereas the rules of an LFG-grammar have an arbitrary form. Thus in principle arbitrary constituent structure trees may be assigned to strings. A separate constraint, expressed in the "Definition of a Valid Derivation" (Kaplan and Bresnan 1982, p. 266), is required to force finiteness of constituent structure trees. This constraint states that a constituent structure tree is valid only if no nonterminal appears twice in a non-branching dominance chain. We will see later that the simulation obeys this constraint. Including this constraint distinguishes our formulation of LFG from formulations of Nakanishi et al. (1992), and Seki et al. (1993).
2. There are two views on derivations in LFG. One is sometimes called *off-line*, the other *on-line* (cf., Pereira and Warren 1983). In the off-line view, a derivation consists of two phases. In the first phase, a total derivation-tree for the constituent structure tree is constructed and f-description statements are collected into an f-description E. In the second phase, the AVMs described by the f-description E are computed. In the on-line view, there is only one phase in the derivation. The AVMs described by the f-description statements are computed by unification while the derivation-tree for the constituent structure tree is constructed. The AVMs that result from both views are the same because a derivation only adds f-description statements and unification is associative and commutative. We will use both the off-line and the on-line view in the simulation, whichever seems to be the most convenient at a particular moment. In order to switch easily from the off-line view to the on-line view, and vice versa, we introduce the notion of an inconsistent f-description. For a precise definition of an inconsistent f-description we refer to Kaplan and Bresnan (1982). Informally, we say that an f-description is *inconsistent* if from its equations we can conclude that a path has two distinct atomic values, or a path has

an atomic and a non-atomic value, or a path has some part of itself as value. Just like the unification of AVMs fails when it is inconsistent, the construction of an f-description fails when it is inconsistent.

3. The combinatory rules of CUG and LFG differ remarkably. On the one hand, there are the daughters of a constituent in CUG. These daughters are related, viz., they are unifiable. To be more precise, a rule is applicable only if the argument category of one daughter unifies with the other daughter. On the other hand, there are the daughters of a constituent in LFG. These daughters are not directly related with one another. These daughters are related at most to their mother. Obviously, two daughters that are related in the same way to their joint mother, are also mutually related, albeit indirectly.

Given a set of attributes L and a set of atomic values A, we can describe of an LFG-grammar by its lexicon, its set of combinatory rules, its distinguished nonterminal. The rules are annotated context-free rewrite rules. The annotations are restricted to the nonterminals on the right-hand side of the rule. Let p, q be paths from L^*, a be a atomic value from A, and M, M' be metavariables in the shape of arrows. Then an annotation is a set of equations of the following form

$$M_1\, p \doteq M_2\, q, \text{ or } M_1\, p \doteq a.$$

Moreover, we demand that the LFG-grammars conform to the Definition of a Valid Derivation.

Consider the following example of the production process of a simplified LFG-grammar. The example will clarify the off-line view on derivations. The example will also show how an annotation is turned into a set of f-description statements.

6.1.2. Example. Consider the following simplified LFG-grammar that contains three lexicon entries and two combinatory rules. The nonterminal S is distinguished as the nonterminal that generates sentences.

1. The lexicon consists of three entries. Each entry consists of a string, a nonterminal, and an annotation.
   (a) John: NP, $\{(\uparrow \text{PERS}) \doteq \textit{3rd}, (\uparrow \text{NUM}) \doteq \textit{sing}\}$
   (b) girls: NP, $\{(\uparrow \text{PERS}) \doteq \textit{3rd}, (\uparrow \text{NUM}) \doteq \textit{plur}\}$
   (c) kisses: V, $\{(\uparrow \text{SUBJ PERS}) \doteq \textit{3rd}, (\uparrow \text{SUBJ NUM}) \doteq \textit{sing}\}$
2. The LFG-grammar contains two rules. The first rule states that a sentence consists of a noun phrase, which is the subject, followed by a verb phrase. The second rule states that a verb phrase is a verb followed by a noun phrase, which is the object.
   (a) $\begin{array}{lcc} \text{S} \rightarrow & \text{NP} & \text{VP} \\ & (\uparrow \text{SUBJ}) \doteq\downarrow & \uparrow\doteq\downarrow \end{array}$
   (b) $\begin{array}{lcc} \text{VP} \rightarrow & \text{V} & \text{NP} \\ & \uparrow\doteq\downarrow & (\uparrow \text{OBJ}) \doteq\downarrow \end{array}$

Given the lexicon and the rules above, let us consider a derivation according to the off-line view. The distinguished nonterminal S and an f-description without

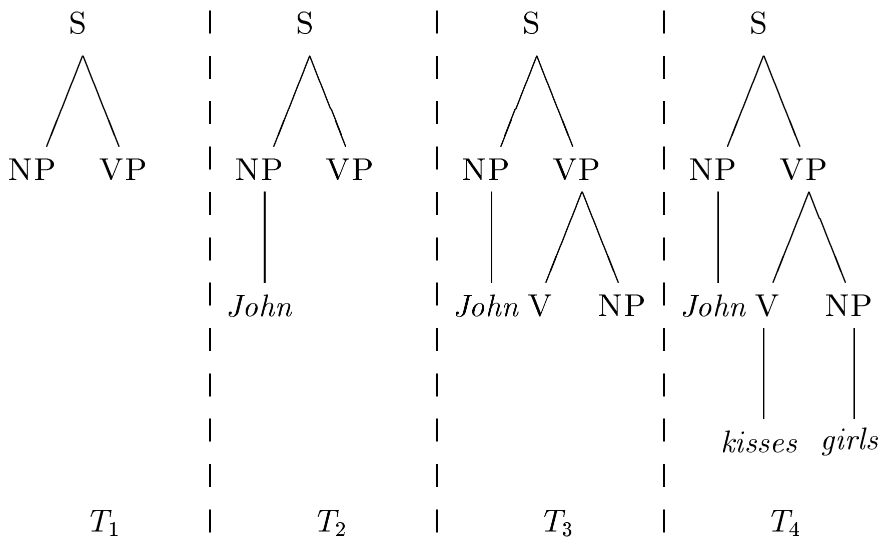


Figure 6.1: Four constituent structure trees.

any statements starts the derivation. Let us call the AVM that belongs to the distinguished nonterminal f_0.

Let us assume that the rule 2a is applied. This application yields the constituent structure tree T_1 given in Figure 6.1. Let us call the AVM that belongs to the NP nonterminal f_1 and the AVM of the VP nonterminal f_2. Then the annotation of the NP, $(\uparrow \text{SUBJ}) \doteq \downarrow$, is turned into an f-description statement by substituting the $\uparrow$ by f_0 and $\downarrow$ by f_1. Likewise the annotation of the VP is turned into an f-description statement by substituting the $\uparrow$ by f_0 and $\downarrow$ by f_2. The resulting f-description statements are then added to the, empty, f-description yielding:

$$\{(f_0 \text{SUBJ}) \doteq f_1, f_0 \doteq f_2\}.$$

Let us assume that the lexicon entry 1a is applied to the NP in constituent structure tree T_1. This application yields the constituent structure tree T_2 given in Figure 6.1. The AVM that belongs to the NP nonterminal is f_1. Hence the annotation of the lexicon entry for "John" is turned into a set of f-description statements by substituting the $\uparrow$ by f_1. These resulting f-description statements are then added to the f-description yielding:

$$\{(f_0 \text{SUBJ}) \doteq f_1, f_0 \doteq f_2, (f_1 \text{PERS}) \doteq \mathit{3rd}, (f_1 \text{NUM}) \doteq \mathit{sing}\}.$$

Now let the rule 2b be applied to the VP nonterminal in the constituent structure tree T_2. This application yields the constituent structure tree T_3 in Figure 6.1. Let the AVM belonging to the V nonterminal be f_3 and the AVM belonging to the NP nonterminal f_4. Now the annotations are turned into an f-description statements by substituting the $\uparrow$ by f_2 and $\downarrow$ by f_3 or f_4. Hence the following f-description statements are then added to the f-description:

$$f_2 \doteq f_3, (f_2\text{OBJ}) \doteq f_4.$$

Now let the lexicon entries 1c and 1b be applied to the V and NP in constituent structure tree T_3. This application yields the constituent structure tree T_4 in Figure 6.1. Now the annotation of the lexicon entries are turned into f-description statements by substituting the $\uparrow$ by f_3 or f_4. Hence the following f-description statements are then added to the f-description:

$$(f_3\text{SUBJ PERS}) \doteq \mathit{3rd}, (f_3\text{SUBJ NUM}) \doteq \mathit{sing}, (f_4\text{PERS}) \doteq \mathit{3rd}, (f_4\text{NUM}) \doteq \mathit{plur}.$$

The derivation has resulted in a constituent structure tree and an f-description for the string "John kisses girls." This f-description consists of ten statements, which describe five AVMs. The AVMs have a large overlap, the AVMs f_1 and f_4 are substructures of the AVMs f_0, f_2, and f_3. The AVMs of the nonterminals NP, f_1 and f_4, are both described by two f-description statements, resulting in:

$$\begin{bmatrix} \text{PERS} & \mathit{3rd} \\ \text{NUM} & \mathit{sing} \end{bmatrix} \text{ and } \begin{bmatrix} \text{PERS} & \mathit{3rd} \\ \text{NUM} & \mathit{plur} \end{bmatrix}.$$

The AVMs f_0, f_2, and f_3 are the same, as expressed by the f-description statements $f_0 \doteq f_2$ and $f_2 \doteq f_3$. According to the f-description statement $(f_2\text{OBJ}) \doteq f_4$ the AVM f_4 is a substructure of f_2. The f-description statements $(f_0\text{SUBJ}) \doteq f_1, (f_3\text{SUBJ PERS}) \doteq \mathit{3rd}$ and $(f_3\text{SUBJ NUM}) \doteq \mathit{sing}$ express that f_1 is a substructure of f_0. Meanwhile the equations check the agreement in person and number between the subject and the verb. Hence the f-description statements describe that the AVM of the nonterminals S, VP and V is

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{PERS} & \mathit{3rd} \\ \text{NUM} & \mathit{sing} \end{bmatrix} \\ \text{OBJ} & \begin{bmatrix} \text{PERS} & \mathit{3rd} \\ \text{NUM} & \mathit{plur} \end{bmatrix} \end{bmatrix}.$$

6.2 The Simulation

In this section we will present a simulation f of CUG by LFG. In Section 6.3 we will prove the correctness of f. That is, CUG-grammar G generates string w iff LFG-grammar $f(G)$ generates w. In Section 6.4 we will show that f is a polynomial time many-one reduction. Hence the recognition problem of LFG is NP-hard.

The simulation f resembles strongly the simulation of CUG by FUG. We recall that a CUG-grammar is described by its lexicon, its distinguished primitive category, and its combinatory rules to combine categories. Below we will first explain how the simulation maps a CUG-category onto a set of statements. Then we will show how the lexicon of CUG is mapped onto a lexicon in LFG. Finally we will show how the distinguished primitive category is mapped onto the distinguished nonterminal of the LFG-grammar, and how the combinatory rules of the CUG-grammar are mapped onto the combinatory rules of the LFG-grammar.

Categories. The simulation f will map a CUG-category onto a set of statements, which forms the annotation of a lexicon entry. This mapping consists of three steps. The first step, by auxiliary function h_1, resembles the mapping from CUG-categories onto AVMs in the simulation of FUG (Section 4.2). The second step, by auxiliary function h_2, turns the AVM constructed in the first step into a set of equations. The third and final step, by auxiliary function h_3, turns the set of equations into an annotation.

The first step, which maps CUG-categories onto AVMs is defined as follows. The set of CUG-categories is defined inductively. Therefore, the auxiliary function h_1 is defined inductively on the categories. Let the values x, y be either atomic or AVMs with attributes VAL, DIR and ARG.

1. If $\mathrm{A}\,[F]$ is a primitive CUG-category then the function h_1 maps this CUG-category onto the following AVM
$$\begin{bmatrix} \textsc{cat} & \mathrm{A} \\ \textsc{feat} & [F] \end{bmatrix}.$$

2. Let the function h_1 map CUG-category $\mathrm{X}\,[F]$ and CUG-category $\mathrm{Y}\,[H]$ onto the following two AVMs, respectively,
$$\begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix} \text{ and } \begin{bmatrix} \textsc{cat} & y \\ \textsc{feat} & [H] \end{bmatrix}.$$
Then h_1 maps the CUG-categories $\mathrm{X}[F] \underset{[H]}{\backslash} \mathrm{Y}$ and $\mathrm{Y} \underset{[H]}{/} \mathrm{X}[F]$ onto the following two AVMs,
$$\begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} & \begin{bmatrix} \textsc{cat} & y \end{bmatrix} \\ \textsc{dir} & \textit{left} \\ \textsc{arg} & \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix} \end{bmatrix} \\ \textsc{feat} & [H] \end{bmatrix} \text{ and } \begin{bmatrix} \textsc{cat} & \begin{bmatrix} \textsc{val} & \begin{bmatrix} \textsc{cat} & y \end{bmatrix} \\ \textsc{dir} & \textit{right} \\ \textsc{arg} & \begin{bmatrix} \textsc{cat} & x \\ \textsc{feat} & [F] \end{bmatrix} \end{bmatrix} \\ \textsc{feat} & [H] \end{bmatrix}.$$

The following fact is proven by induction on the number of slashes in a CUG-category because the atomic values *left* and *right* only unify with themselves.

6.2.1. FACT. The unification of the CUG-categories $\mathrm{X}\,[F]$ and $\mathrm{Y}\,[H]$ yields CUG-category $\mathrm{U}[F']$ iff the unification of the AVMs $h_1(\mathrm{X}\,[F])$ and $h_1(\mathrm{Y}\,[H])$ yields the AVM $h_1(\mathrm{U}[F'])$.

The function h_1 expresses the important ideas of the simulation. The next two functions, h_2 and h_3, merely serve to express the information contained in the AVMs that result from h_1 in a form that LFG-grammars can manipulate.

The following function h_2 maps AVMs onto sets of equations in an efficient and rather obvious way. Only the treatment of the box-labels deserves some attention. Mainly for efficiency reasons, the box-labels in the AVM are encoded by some new attributes, viz., for each box-label $\boxed{i}$ an attribute $\textsc{box}_i$ in introduced.

The auxiliary function h_2 is defined as follows, where ATTR, $\textsc{attr}_j$, $\textsc{box}_i$ are attributes, v, v_j are values, a is an atomic value, p is a path consisting of zero or more attributes, and $\boxed{i}$ is a box-label.

1. If $[F]$ has the form $\left[\begin{array}{ll} p & \left[\begin{array}{ll} \text{ATTR}_1 & w_1 \\ \vdots & \vdots \\ \text{ATTR}_n & w_n \end{array}\right] \end{array}\right]$, where $w_j = \boxed{i}\, v_j$, or $w_j = v_j$, then

$$h_2([F]) := \bigcup_{1 \leq j \leq n} h_2(\left[\ p\,[\text{ATTR}_j \quad w_j\,]\ \right])$$

2. If $[F]$ has the form $\left[\ p\,[\text{ATTR} \quad \boxed{i}\,v]\ \right]$ then

$$h_2([F]) := h_2(\left[\ p\,[\text{ATTR} \quad \boxed{i}\,]\ \right]) \cup h_2(\left[\ p\,[\text{ATTR} \quad v]\ \right])$$

3. If $[F]$ has the form $\left[\ p\,[\text{ATTR} \quad \boxed{i}\,]\ \right]$ then

$$h_2([F]) := \{\langle p\, \text{ATTR}\rangle \doteq \langle \text{BOX}_i\rangle\}$$

4. If $[F]$ has the form $\left[\ p\,[\text{ATTR} \quad a]\ \right]$ then

$$h_2([F]) := \{\langle p\, \text{ATTR}\rangle \doteq a\}$$

Let us assume that the function h_2, is applied to some AVM $[F]$ with box-labels $\boxed{1} \ldots \boxed{n}$. Then the set of equations $h_2([F])$ can be partitioned into equations whose right-hand side is an atomic value, E, and whose right-hand side is path $\langle \text{BOX}_i\rangle$, E_i. Thus $E = \{p \doteq a \mid p \doteq a \in h_2([F])\}$, and $E_i = \{p \doteq \langle \text{BOX}_i\rangle \mid p \doteq \langle \text{BOX}_i\rangle \in h_2([F])\}$.

The final function h_3 maps these set of equations $E, E_1, \ldots, E_n$ onto a set of statements, which as we will see later form an annotation for an entry in the lexicon.

1. If $h_2([F]) = \bigcup_i E_i \cup E$, then $h_3(h_2([F])) := \bigcup_i h_3(E_i) \cup h_3(E)$.
2. $h_3(E) := \{(\uparrow\ p) \doteq a \mid p \doteq a \in E\}$
3. For some equation $p \doteq \langle \text{BOX}_i\rangle$ in the set E_i, we define $h_3(E_i) := h_3(E_i, p)$.
4. $h_3(E_i, p) := \{(\uparrow\ p) \doteq (\uparrow\ q) \mid q \doteq \text{BOX}_i \in E_i\}$

6.2.2. FACT. The AVM denoted by the metavariable $\uparrow$ in the annotation $h_3(h_2([F]))$ is the AVM $[F]$.

By the previous fact and Fact 6.2.1 the following fact holds.

6.2.3. FACT. The unification of the CUG-categories X $[F]$ and Y $[H]$ yields CUG-category U$[F']$ iff the AVM denoted by the metavariable $\uparrow$ in the annotation $f(\text{U}\,[F'])$ $= h_3(h_2(h_1(\text{U}\,[F'])))$ equals the AVM denoted by the metavariable $\uparrow$ in $f(\text{X}\,[F]) \cup f(\text{Y}\,[H])$.

6.2.4. EXAMPLE. Consider how the simulation maps CUG-category $\underset{\boxed{1}[\text{DEF}\ +]}{\text{N} \quad / \quad \text{N}\boxed{1}}$, onto an annotation.

First h_1 maps the CUG-categories N$\boxed{1}$[DEF $+$] and N$\boxed{1}$ onto the following two AVMs:

$$\left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \boxed{1}[\text{DEF}\ +] \end{array}\right] \text{ and } \left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \boxed{1} \end{array}\right].$$

Thus h_1 maps $\mathrm{N} \underset{\boxed{1}[\textsc{def}\ +]}{/} \mathrm{N}\boxed{1}$ onto the AVM $[F]$

$$[F] = \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{val} & [\textsc{cat}\ N] \\ \textsc{dir} & \mathit{right} \\ \textsc{arg} & \left[\begin{array}{ll} \textsc{cat} & N \\ \textsc{feat} & \boxed{1} \end{array}\right] \end{array}\right] \\ \textsc{feat} & \boxed{1}[\textsc{def}\ +] \end{array}\right]$$

Next h_2 maps the AVM $[F]$ in the following way onto a set of equations.

$$\begin{array}{rcl} h_2([F]) & = & h_2\left(\left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{val} & [\textsc{cat}\ N] \\ \textsc{dir} & \mathit{right} \\ \textsc{arg} & \left[\begin{array}{ll} \textsc{cat} & N \\ \textsc{feat} & \boxed{1} \end{array}\right] \end{array}\right] \end{array}\right]\right) \cup h_2(\left[\ \textsc{feat}\ \ \boxed{1}[\textsc{def}\ +]\ \right]) \\ & = & h_2(\left[\ \textsc{cat} \mid \textsc{val}\,[\textsc{cat}\quad N]\ \right]) \cup h_2(\left[\ \textsc{cat}\,[\textsc{dir}\quad \mathit{right}]\ \right]) \\ & \cup & h_2(\left[\ \textsc{cat} \mid \textsc{arg}\,[\textsc{cat}\quad N]\ \right]) \cup h_2(\left[\ \textsc{cat} \mid \textsc{arg}\,[\textsc{feat}\quad \boxed{1}]\ \right]) \\ & \cup & h_2(\left[\ \textsc{feat}\quad \boxed{1}\ \right]) \cup h_2(\left[\ \textsc{feat}\,[\textsc{def}\quad +]\ \right]) \\ & = & \{\langle \textsc{cat val cat}\rangle \doteq N, \langle \textsc{cat dir}\rangle \doteq \mathit{right}, \langle \textsc{cat arg cat}\rangle \doteq N, \\ & & \langle \textsc{cat arg feat}\rangle \doteq \langle \textsc{box}_1\rangle, \langle \textsc{feat}\rangle \doteq \langle \textsc{box}_1\rangle, \langle \textsc{feat def}\rangle \doteq +\} \end{array}$$

Finally h_3 maps this set of equations on the following annotation.

$$\begin{array}{rl} \{ \qquad\qquad (\uparrow \textsc{cat val cat}) \doteq N, & (\uparrow \textsc{cat dir}) \doteq \mathit{right}, \\ (\uparrow \textsc{cat arg cat}) \doteq N, & (\uparrow \textsc{feat def}) \doteq +, \\ (\uparrow \textsc{cat arg feat}) \doteq (\uparrow \textsc{cat arg feat}), & (\uparrow \textsc{cat arg feat}) \doteq (\uparrow \textsc{feat})\} \end{array}$$

Similarly, the CUG-category $\mathrm{N}\boxed{1}\left[\begin{array}{ll} \textsc{pers} & \mathit{3rd} \\ \textsc{num} & \mathit{sing} \end{array}\right] \underset{\boxed{1}}{\backslash}\ \mathrm{S}$ is successively mapped onto the following AVM, set of equations and annotation:

$$\left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{val} & [\textsc{cat}\ S] \\ \textsc{dir} & \mathit{left} \\ \textsc{arg} & \left[\begin{array}{ll} \textsc{cat} & N \\ \textsc{feat} & \boxed{1}\left[\begin{array}{ll} \textsc{pers} & \mathit{3rd} \\ \textsc{num} & \mathit{sing} \end{array}\right] \end{array}\right] \end{array}\right] \\ \textsc{feat} & \boxed{1} \end{array}\right]$$

$$\begin{array}{rl} \{ \qquad \langle \textsc{cat val cat}\rangle \doteq S, & \langle \textsc{cat dir}\rangle \doteq \mathit{left}, \\ \langle \textsc{cat arg cat}\rangle \doteq N, & \langle \textsc{cat arg feat}\rangle \doteq \langle \textsc{box}_1\rangle, \\ \langle \textsc{cat arg feat pers}\rangle \doteq \mathit{3rd}, & \langle \textsc{cat arg feat num}\rangle \doteq \mathit{sing}, \\ \langle \textsc{feat}\rangle \doteq \langle \textsc{box}_1\rangle & \qquad\qquad\qquad\qquad \} \end{array}$$

$$\{ \quad (\uparrow \text{ CAT VAL CAT}) \doteq S,\ (\uparrow \text{ CAT DIR}) \doteq \textit{left},\ (\uparrow \text{ CAT ARG CAT}) \doteq N,\ (\uparrow \text{ CAT ARG FEAT PERS}) \doteq \textit{3rd},\ (\uparrow \text{ CAT ARG FEAT NUM}) \doteq \textit{sing},\ (\uparrow \text{ CAT ARG FEAT}) \doteq (\uparrow \text{ CAT ARG FEAT}),\ (\uparrow \text{ FEAT}) \doteq (\uparrow \text{ CAT ARG FEAT}) \quad \}$$

The lexicon. Now let us show how the lexicon of the CUG-grammar is mapped onto the lexicon of the LFG-grammar. Given the auxiliary functions h_1, h_2, and h_3 above, the simulation of the lexicon is rather straightforward. The nonterminals in the entries of the lexicon of the LFG-grammar do not contribute much to the intension of the entries. The simulation encodes the category information of the simulated CUG-categories in the AVMs of the nonterminals. All entries of the CUG-lexicon are mapped onto entries in the lexicon of LFG, and the lexicon of the LFG-grammar contains no other entries. This mapping is defined in the following way.

Given an entry of the CUG-lexicon, w : $\mathrm{X}[F]$. The CUG-category $\mathrm{X}\,[F]$ is mapped onto the set of statements $f(\mathrm{X}\,[F]) \ = \ h_3(h_2(h_1(\mathrm{X}\,[F])))$. Now the simulation f maps this entry of the CUG-lexicon onto the following *entry* in the *lexicon* of the LFG-grammar:

$$(w, \mathrm{A}, f(\mathrm{X}\,[F])).$$

Distinguished nonterminal. The distinguished category in CUG identifies derivations that produce sentences. The sentences that an LFG-grammar produces are identified by the distinguished nonterminal. The simulation maps the distinguished primitive category S onto the distinguished nonterminal S, and one special combinatory rule, called R_s, which is discussed below.

Combinatory rules. The LFG-grammar contains the four *combinatory rules* from Table 6.1. The reader can easily check that the derivations that result from the rules obey the "Definition of a Valid Derivation." The first combinatory rule, which is called R_s, starts the derivations. This rule accomplishes that sentences are produced only by nonterminals whose AVM contains the category information of the distinguished category S. The two rules R_l and R_r correspond to the left and right application rules of CUG. A simple trick is used to express the constraint in CUG that the daughters of a constituent must unify. To be precise, the argument category of the functor daughter, and the other daughter are passed on to a special attribute, TST, of the mother. The attribute TST is mainly used to check whether the argument category of the functor daughter and the other daughter, the argument daughter, are unifiable. The exact value of the attribute TST, however, is irrelevant for the mother nonterminal. All the relevant information in the AVM of the mother nonterminal is reachable from the attributes CAT and FEAT. The fourth rule, R_b, is used to extract only the relevant information from the AVM of a nonterminal B.

$$(R_s) \quad \mathrm{S} \rightarrow \begin{array}{c} \mathrm{A} \\ (\downarrow \textsc{cat}) \doteq \mathrm{S} \end{array}$$

$$(R_l) \quad \mathrm{B} \rightarrow \begin{array}{c} \mathrm{A} \\ (\uparrow \textsc{tst}) \doteq \downarrow \end{array} \quad \begin{array}{r} \mathrm{A} \\ (\uparrow \textsc{tst}) \doteq (\downarrow \textsc{cat arg}) \\ (\uparrow \textsc{cat}) \doteq (\downarrow \textsc{cat val cat}) \\ (\uparrow \textsc{feat}) \doteq (\downarrow \textsc{feat}) \\ (\downarrow \textsc{cat dir}) \doteq \mathit{left} \end{array}$$

$$(R_r) \quad \mathrm{B} \rightarrow \begin{array}{r} \mathrm{A} \\ (\uparrow \textsc{tst}) \doteq (\downarrow \textsc{cat arg}) \\ (\uparrow \textsc{cat}) \doteq (\downarrow \textsc{cat val cat}) \\ (\uparrow \textsc{feat}) \doteq (\downarrow \textsc{feat}) \\ (\downarrow \textsc{cat dir}) \doteq \mathit{right} \end{array} \quad \begin{array}{c} \mathrm{A} \\ (\uparrow \textsc{tst}) \doteq \downarrow \end{array}$$

$$(R_b) \quad \mathrm{A} \rightarrow \begin{array}{c} \mathrm{B} \\ (\uparrow \textsc{cat}) \doteq (\downarrow \textsc{cat}) \\ (\uparrow \textsc{feat}) \doteq (\downarrow \textsc{feat}) \end{array}$$

Table 6.1: Combinatory rules of LFG-grammar $f(G)$, provided that S is the distinguished CUG-category.

6.2.5. DEFINITION. Given a CUG-grammar G, the *simulation* f maps this grammar onto the LFG-grammar $f(G)$, where $f(G)$ is a set of combinatory rules, a lexicon and distinguished nonterminal.

- The *lexicon* is defined as the set of entries

$$\{(w, A, f(\mathrm{X}\,[F])) \mid w : \mathrm{X}\,[F] \text{ an entry from the lexicon of } G\}.$$

- The set of *combinatory rules* is defined as the set of rules given in Table 6.1, provided that S is the distinguished category of G.
- The *distinguished nonterminal* of the LFG-grammar is defined as S, if S is the distinguished category of G.

Let us now consider a simple example of the simulation. We will present some examples of entries in the lexicon of the simulating LFG-grammar. We conclude with some examples of constituents that may be constructed by the simulating LFG-grammar.

6.2.6. EXAMPLE. Let there be a CUG-grammar with the ordinary combinatory rules, the distinguished category S and the following lexicon with five entries:

the: $N_{\boxed{1}[\textsc{def}\ +]} \ / \ N\boxed{1}$ a: $N_{\boxed{1}[\textsc{def}\ -]} \ / \ N\boxed{1}[\textsc{num}\ \textit{sing}]$ kisses: $(N\boxed{1}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \backslash_{\boxed{1}} S)/N$

boy: $N\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix}$ girl: $N\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix}$

The lexicon of the CUG-grammar is mapped onto the lexicon of the LFG-grammar. Now the simulation maps the entry from the lexicon of CUG-grammar for the string "a," $N_{\boxed{1}[\textsc{def}\ -]} \ / \ N\boxed{1}[\textsc{num}\ \textit{sing}]$, onto the entry (a, A, T) from the lexicon of the LFG-grammar, where T is the following annotation.

$$
\begin{array}{rll}
\{ & (\uparrow \textsc{cat val cat}) \doteq N, & (\uparrow \textsc{cat dir}) \doteq \textit{right}, \\
 & (\uparrow \textsc{cat arg cat}) \doteq N, & (\uparrow \textsc{feat def}) \doteq -, \\
 & (\uparrow \textsc{cat arg feat num}) \doteq \textit{sing} & (\uparrow \textsc{cat arg feat}) \doteq (\uparrow \textsc{cat arg feat}), \\
 & (\uparrow \textsc{cat arg feat}) \doteq (\uparrow \textsc{feat}) & \qquad \}
\end{array}
$$

Similarly, the entry for the string "kisses" is $(N\boxed{1}\begin{bmatrix}\textsc{pers} & \textit{3rd}\\ \textsc{num} & \textit{sing}\end{bmatrix} \backslash_{\boxed{1}} S)/N$ is mapped onto the entry $(\textit{kisses}, \mathrm{A}, T)$ from the lexicon of the LFG-grammar, where T is the following set.

$$
\begin{array}{rll}
\{ & (\uparrow \textsc{cat val cat val cat}) \doteq S, & (\uparrow \textsc{cat val cat dir}) \doteq \textit{left}, \\
 & (\uparrow \textsc{cat val cat arg cat}) \doteq N, & (\uparrow \textsc{cat arg cat}) \doteq N, \\
 & (\uparrow \textsc{cat val cat arg feat pers}) \doteq \textit{3rd} & (\uparrow \textsc{cat dir}) \doteq \textit{right}, \\
 & (\uparrow \textsc{cat val cat arg feat num}) \doteq \textit{sing} & (\uparrow \textsc{feat}) \doteq (\uparrow \textsc{feat}) \\
 & (\uparrow \textsc{feat}) \doteq (\uparrow \textsc{cat val cat arg feat}) & \qquad \}
\end{array}
$$

Given the combinatory rules and the entries in the lexicon of the LFG-grammar, we can construct the strings "the boy" and "a girl." We will consider the construction of the string "the boy" in detail. We confine to a sketch of the construction of the string "a girl" because it is rather similar to the construction of "the boy". The LFG-grammar generates a constituent structure tree and an f-description for each of the two strings. The rules R_b and R_r, and the lexicon entries for "the" and "boy" construct the left constituent structure tree in Figure 6.2.

Assume that the AVM for the upper nonterminal A in the constituent structure tree of "the boy" is f, the AVM for the nonterminal B is f_0, f_1 is the AVM for the leftmost daughter of nonterminal B, and f_2 is the AVM for its rightmost daughter. The f-description for the constituent structure of "the boy" in Figure 6.2, then is as given in Table 6.2.

The f-description given in Table 6.2 describes four AVMs, viz., f, f_0, f_1 and f_2. These four AVMs have a large overlap. We have added indices as superscripts in the AVMs below to emphasize which parts of the AVMs are the same.

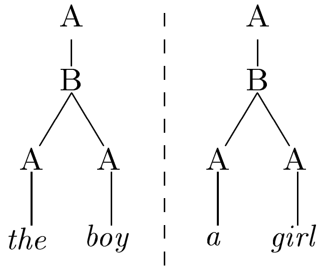


Figure 6.2: Constituent structure trees for “the boy” and “a girl.”

$$f = \left[\begin{array}{ll} \textsc{cat}^1 & N \\ \textsc{feat}^2 & \left[\begin{array}{ll} \textsc{def} & + \\ \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{array}\right] \end{array}\right]$$

$$f_0 = \left[\begin{array}{ll} \textsc{cat}^1 & N \\ \textsc{feat}^2 & \boxed{1}\left[\begin{array}{ll} \textsc{def} & + \\ \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{array}\right] \\ \textsc{tst}^3 & \left[\begin{array}{ll} \textsc{cat} & N \\ \textsc{feat} & \boxed{1} \end{array}\right] \end{array}\right]$$

$$f_1 = \left[\begin{array}{ll} \textsc{cat} & \left[\begin{array}{ll} \textsc{val} & [\textsc{cat}^1\ N] \\ \textsc{dir} & \textit{right} \\ \textsc{arg}^3 & \left[\begin{array}{ll} \textsc{cat} & N \\ \textsc{feat} & \boxed{2} \end{array}\right] \end{array}\right] \\ \textsc{feat}^2 & \boxed{2}\left[\begin{array}{ll} \textsc{def} & + \\ \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{array}\right] \end{array}\right]$$

$$f_2 = \left[\begin{array}{ll} \textsc{cat} & N \\ \textsc{feat} & \left[\begin{array}{ll} \textsc{def} & + \\ \textsc{pers} & \textit{3rd} \\ \textsc{num} & \textit{sing} \end{array}\right] \end{array}\right]^3$$

We will now give a sketch of the construction of the string “a girl.” The LFG-grammar generates a constituent structure tree for “a girl” similar to the constituent structure tree for “the boy,” which is given in Figure 6.2. The only difference is that the leaves are “a” and “girl” instead of “the” and “boy.”

Assume that the AVM for the upper nonterminal A in structure tree for “a girl” is f', for the nonterminal B f_3, for the leftmost daughter f_4, and the rightmost daughter f_5. The f-description for “a girl” is also similar to the f-description given in Table 6.2, when the variables f, f_0, f_1 and f_2 are replaced by the variables f', f_3, f_4, and f_5, respectively. The only real differences lies in the f-description statements that come from the entry “a”, given at the begin of this example:

1. Statement $(f_4 \langle \textsc{feat def} \rangle) \doteq -$ replaces statement $(f_1 \langle \textsc{feat def} \rangle) \doteq +$.

From the lexicon entry "the"		
$(f_1\,\langle\textsc{cat val cat}\rangle)$	$\doteq$	N,
$(f_1\,\langle\textsc{cat dir}\rangle)$	$\doteq$	*right*,
$(f_1\,\langle\textsc{cat arg cat}\rangle)$	$\doteq$	N,
$(f_1\,\langle\textsc{feat def}\rangle)$	$\doteq$	$+$,
$(f_1\,\langle\textsc{cat arg feat}\rangle)$	$\doteq$	$(f_1\,\langle\textsc{cat arg feat}\rangle)$,
$(f_1\,\langle\textsc{cat arg feat}\rangle)$	$\doteq$	$(f_1\,\langle\textsc{feat}\rangle)$
From the lexicon entry "boy"		
$(f_2\,\langle\textsc{cat}\rangle)$	$\doteq$	N,
$(f_2\,\langle\textsc{feat pers}\rangle)$	$\doteq$	*3rd*,
$(f_2\,\langle\textsc{feat num}\rangle)$	$\doteq$	*sing*,
From the combinatory rule R_r		
$(f_0\,\langle\textsc{tst}\rangle)$	$\doteq$	$(f_1\,\langle\textsc{cat arg}\rangle)$
$(f_0\,\langle\textsc{cat}\rangle)$	$\doteq$	$(f_1\,\langle\textsc{cat val cat}\rangle)$
$(f_0\,\langle\textsc{feat}\rangle)$	$\doteq$	$(f_1\,\langle\textsc{feat}\rangle)$
$(f_1\,\langle\textsc{cat dir}\rangle)$	$\doteq$	*right*
$(f_0\,\langle\textsc{tst}\rangle)$	$\doteq$	f_2
From the combinatory rule R_b		
$(f\,\langle\textsc{cat}\rangle)$	$\doteq$	$(f_0\,\langle\textsc{cat}\rangle)$
$(f\,\langle\textsc{feat}\rangle)$	$\doteq$	$(f_0\,\langle\textsc{feat}\rangle)$

Table 6.2: An f-description for "the boy."

2. The statement $(f_4\,\langle\textsc{cat arg feat num}\rangle) \doteq$ *sing* is added.

It is evident that the f-description for "a girl" describes the three AVMs f', f_3, f_4, and f_5, which differ only in the value of the attribute DEF from the AVMs f, f_0, f_1 and f_2.

Let us now consider simplified constituent structure trees and AVMs that belong to the strings "kisses a girl" and "the boy kisses a girl." The first string is constructed from the strings "kisses" and "a girl" by the rules R_b and R_r. The second string is constructed from the strings "the boy" and "kisses a girl" by the rules R_s, R_b, and R_l. The constituent structure trees for both strings are given in Figure 6.3.

Schematic representations of the AVMs for the nonterminals B in the constituent structure trees in Figure 6.3 are given below, as f_6 and f_7, respectively. The value of attribute TST in f_6 is the AVM for the string "a girl," f_3. The value of attribute TST in f_7 is the AVM for the string "the boy," f_0. The AVMs for the upper nonterminals A of these constituent structure trees are obtained from f_6 and f_7, by removing the attribute TST and its value.

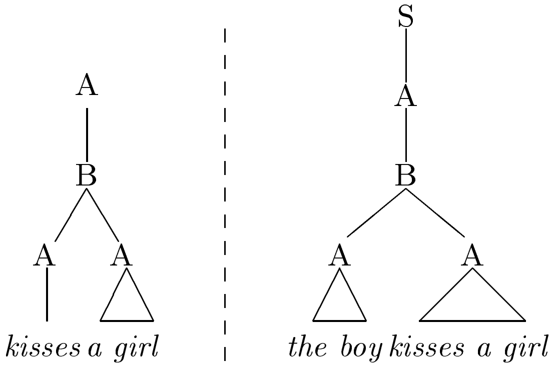


Figure 6.3: Constituent structure trees for “kisses a girl” and “the boy kisses a girl.”

$$f_6 = \left[\begin{array}{ll}
\text{CAT} & \left[\begin{array}{ll} \text{VAL} & [\text{CAT}\ S] \\ \text{DIR} & \textit{left} \\ \text{ARG} & \boxed{1}\left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \left[\begin{array}{ll} \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{array}\right] \end{array}\right] \end{array}\right] \\
\text{FEAT} & \boxed{1} \\
\text{TST} & \left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \boxed{2}\left[\begin{array}{ll} \text{DEF} & - \\ \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{array}\right] \\ \text{TST} & \left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \boxed{2} \end{array}\right] \end{array}\right]
\end{array}\right]$$

$$f_7 = \left[\begin{array}{ll}
\text{CAT} & S \\
\text{FEAT} & \boxed{1}\left[\begin{array}{ll} \text{DEF} & + \\ \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{array}\right] \\
\text{TST} & \left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \boxed{1}\left[\begin{array}{ll} \text{DEF} & + \\ \text{PERS} & \textit{3rd} \\ \text{NUM} & \textit{sing} \end{array}\right] \\ \text{TST} & \left[\begin{array}{ll} \text{CAT} & N \\ \text{FEAT} & \boxed{1} \end{array}\right] \end{array}\right]
\end{array}\right]$$

The reader should notice that the top nonterminal of the constituent structure tree for “the boy kisses a girl” is S. Hence the string “the boy kisses a girl” is a sentence.

6.3 Correctness of the Simulation

In the previous section we have presented the simulation. The main purpose of this section is to prove the correctness of the simulation. That is, CUG-grammar G and LFG-grammar $f(G)$ generate the same language. Before we will prove the correctness of the simulation we will present some definitions and facts that will be convenient in the lemmas to come.

6.3.1 Notation and Definitions

Let us start with the introduction of a notation that connects annotations and set of f-description statements.

6.3.1. DEFINITION. Let f_0 and f_1 be variables for AVMs from LFG, and T and T' be annotations. Suppose that T contains both metavariables $\uparrow$ and $\downarrow$ and T' contains only metavariable $\uparrow$.

1. The *f-description* $T|_{\{\uparrow/f_0\,\downarrow/f_0\}}$ is the set of equations from T in which each occurrence of metavariable $\uparrow$ is instantiated by f_0, and each occurrence of metavariable $\downarrow$ is instantiated by f_1.
2. Likewise, the *f-description* $T'|_{\{\uparrow/f_0\}}$ is the set of equations T' in which each occurrence of metavariable $\uparrow$ is instantiated by f_0.

For the benefit of the simulation, we present our own definitions of the notions "derives," "produces" and "generates" for LFG. The definition of the notions "derives" and "produces" are rather similar to Definition 3.3.11 and 3.3.12.

6.3.2. DEFINITION. A nonterminal A_0 with AVM f_0 and f-description E *derives* a sequence of nonterminals $\mathrm{A}_1 \ldots \mathrm{A}_n$ with AVMs $f_1 \ldots f_n$, and f-descriptions $E_1 \ldots E_n$, respectively, iff

1. the grammar contains a rule

$$\begin{array}{llll} \mathrm{A}_0 \rightarrow & \mathrm{A}_1 & \ldots & \mathrm{A}_n \\ & T_1 & & T_n \end{array}$$

   and T_i are annotations, and $E_0 = \bigcup_{1\leq i\leq n} T_i|_{\{\uparrow/f_0,\downarrow/f_i\}} \cup E_i$, or
2. the grammar contains a rule

$$\begin{array}{llll} \mathrm{A}_0 \rightarrow & \mathrm{B}_1 & \ldots & \mathrm{B}_m \\ & T_1 & & T_m \end{array}$$

   and T_i are annotations, and $E = \bigcup_{1\leq i\leq m} E_i \cup T_i|_{\{\uparrow/f_0,\downarrow/g_i\}}$, and B_i with AVM g_i and f-description E_i derives the sequence of nonterminals $\mathrm{A}_{b_i} \ldots \mathrm{A}_{e_i}$ with AVMs $f_{b_i} \ldots f_{e_i}$, and f-descriptions $E_{b_i} \ldots E_{e_i}$, respectively, and $\mathrm{A}_1 \ldots \mathrm{A}_n = \mathrm{A}_{b_1} \ldots \mathrm{A}_{e_j} \mathrm{A}_{b_{j+1}} \ldots \mathrm{A}_{e_m}$, $(1 \leq j < m)$.

6.3.3. DEFINITION. A nonterminal A with AVM f_0 and f-description E *produces* a string $w = w_1 \dots w_n$ iff

1. the LFG-lexicon contains an entry (w, A, T) and $T|_{\{\uparrow/f_0\}} \subseteq E$, or
2. nonterminal A with AVM f_0 and f-description $E' \subseteq E$ derives a sequence of nonterminals $\mathrm{A}_1 \dots \mathrm{A}_n$ with AVMs $f_1 \dots f_n$, respectively, and A_i with AVM f_i and f-description E produces w_i.

6.3.4. DEFINITION. LFG-grammar G *generates* string w iff the distinguished nonterminal S, with some AVM f_0 and some f-description E, produces w. The *language* that grammar G generates ($L(G)$) is defined as the set of all strings that S produces.

6.3.2 Proof of Correctness

Given the definitions above, we can prove the correctness of the simulation in Theorem 6.3.8. Lemmas 6.3.5 and 6.3.6 show that one single application of a combinatory rule for a CUG-grammar G can be simulated by the LFG-grammar $f(G)$. Lemma 6.3.7 shows that a CUG-category and the corresponding (nonterminal, f-description)-pair produce the same strings.

The following lemma states roughly that a CUG-constituent with mother $\mathrm{Y}\,[H]$, left daughter $\mathrm{X}\,[F]$ and right daughter $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ can be constructed iff an LFG-constituent can be constructed, whose top nonterminal is A with AVM f_0 and whose bottom nonterminals are A and A with AVM f_1 with AVM f_2, respectively

6.3.5. LEMMA. *Let the simulation map* $\mathrm{X}\,[F]$ *onto the annotation* T_1*,* $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ *onto the annotation* T_2*, and* $\mathrm{Y}\,[H]$ *onto* T_0*, and let there be a nonterminal* A *with AVM* f_1 *and f-description* $T_1|_{\{\uparrow/f_1\}}$*, and another nonterminal* A *with AVM* f_2 *and f-description* $T_2|_{\{\uparrow/f_2\}}$*.*

The left application of the CUG-categories $\mathrm{X}\,[F]$ *and* $(\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ *yields* $\mathrm{Y}\,[H]$ *iff nonterminal* A *with AVM* f_0 *and f-description* E *derives, by rules* R_b *and* R_l*, the sequence of nonterminals* A A *with* f_1*,* $T_1|_{\{\uparrow/f_1\}}$ *and* f_2*,* $T_2|_{\{\uparrow/f_2\}}$*, respectively, and the AVM* f_0 *in* E *equals the AVM denoted in* T_0*.*

Proof. The simulation maps $\mathrm{X}\,[F]$ onto T_1, $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ onto T_2, and $\mathrm{Y}\,[H]$ onto T_0, where

$$T_1 = \; h_3(h_2([F_1])), \quad [F_1] = \begin{bmatrix} \text{CAT} & x \\ \text{FEAT} & [F] \end{bmatrix}$$

$$T_2 = \; h_3(h_2([F_2])), \quad [F_2] = \begin{bmatrix} \text{CAT} & \begin{bmatrix} \text{VAL} & [\text{CAT } v] \\ \text{DIR} & \textit{left} \\ \text{ARG} & \begin{bmatrix} \text{CAT} & u \\ \text{FEAT} & [F'] \end{bmatrix} \end{bmatrix} \\ \text{FEAT} & [H'] \end{bmatrix}$$

$$T_0 = \; h_3(h_2([F_0])), \;\; [F_3] = \begin{bmatrix} \text{CAT} & y \\ \text{FEAT} & [H] \end{bmatrix}$$

Only if: Assume that the left application of $\mathrm{X}\,[F]$ and $\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V}$ yields $\mathrm{Y}\,[H]$. Then both $\mathrm{X}\,[F]$ and $\mathrm{U}[F']$, and $\mathrm{V}\,[H']$ and $\mathrm{Y}\,[H]$ are unifiable.

Now let the nonterminal B with AVM f' and f-description E' derive, by rule R_l, the sequence of nonterminals A A. Then the f-description E' equals

$$T_1|_{\{\uparrow/f_1\}} \cup T_2|_{\{\uparrow/f_2\}} \cup \{(f'\,\langle \text{TST}\rangle) \doteq f_1,\; (f'\,\langle \text{TST}\rangle) \doteq (f_2\,\langle \text{CAT ARG}\rangle),$$
$$(f'\,\langle \text{CAT}\rangle) \doteq (f_2\,\langle \text{CAT VAL CAT}\rangle), (f'\,\langle \text{FEAT}\rangle) \doteq (f_2\,\langle \text{FEAT}\rangle), (f_2\,\langle \text{CAT DIR}\rangle) \doteq \mathit{left}\}.$$

The f-description E' is consistent iff the AVM f_1 unifies with the value of path $\langle$CAT ARG$\rangle$ in AVM f_2, and the value of path $\langle$CAT DIR$\rangle$ in AVM f_2 is *left*. By Fact 6.2.2 the AVMs f_1 and f_2 equal the AVMs $[F_1]$ and $[F_2]$ above. Hence the value of path $\langle$CAT DIR$\rangle$ in AVM f_2 is indeed *left*. By the assumption that the CUG-categories $\mathrm{X}\,[F]$ and $\mathrm{U}[F']$ unify and Fact 6.2.3 the unification of AVM f_1 with the value of path $\langle$CAT ARG$\rangle$ in AVM f_2 succeeds. Hence E' is a consistent f-description. Thus the AVM f' that is described in E' consists of an attribute TST whose value is the unification of f_1 with the value of $\langle$CAT ARG$\rangle$ in f_2. Furthermore, the value of attribute CAT in f' is equal to the value of path $\langle$CAT VAL CAT$\rangle$ in f_2 and the value of attribute FEAT is f' is equal to the value of attribute FEAT is f_2. The assumption states that these values are changed into y and $[F]$ due to the unification of f_1 with the value of $\langle$CAT ARG$\rangle$ in f_2.

Now let the nonterminal A with AVM f_0 and f-description E derive B, by rule R_b. Then the f-description E equals $E' \cup \{(f_0\,\langle \text{CAT}\rangle) \doteq (f'\,\langle \text{CAT}\rangle),\; (f_0\,\langle \text{FEAT}\rangle) \doteq (f'\,\langle \text{FEAT}\rangle)\}$. Hence the AVM f_0 in E equals the AVM denoted in T_0: $[F_0]$.

If: Assume that nonterminal A with AVM f_0 and f-description E derives, by rules R_b and R_l, the sequence of nonterminals A A with f_1, $T_1|_{\{\uparrow/f_1\}}$ and f_2, $T_2|_{\{\uparrow/f_2\}}$, respectively, and the AVM f_0 in E equals the AVM denoted in T_0. Then nonterminal A derives, by rule R_b, nonterminal B with AVM f' and f-description E'. This nonterminal B derives the sequence of nonterminals A A by rule R_l. Hence f-description $E' = T_1|_{\{\uparrow/f_1\}} \cup T_2|_{\{\uparrow/f_2\}} \cup \{(f'\,\langle \text{TST}\rangle) \doteq f_1,\; (f'\,\langle \text{TST}\rangle) \doteq (f_2\,\langle \text{CAT ARG}\rangle),$ $(f'\,\langle \text{CAT}\rangle) \doteq (f_2\,\langle \text{CAT VAL CAT}\rangle),\; (f'\,\langle \text{FEAT}\rangle) \doteq (f_2\,\langle \text{FEAT}\rangle),\; (f_2\,\langle \text{CAT DIR}\rangle) \doteq \mathit{left}\}$.

Now the left application of the CUG-categories $\mathrm{X}\,[F]$ and $(\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ yields $\mathrm{Y}\,[H]$ iff $\mathrm{X}\,[F]$ unifies with $\mathrm{U}\,[F']$ and the CUG-category $(\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ contains box-labels such that the unification of $\mathrm{X}\,[F]$ with $\mathrm{U}\,[F']$ turns the CUG-category $\mathrm{V}\,[H']$ into $\mathrm{Y}\,[H]$.

From the assumption it follows that the AVM f_1 in $T_1|_{\{\uparrow/f_1\}}$ equals $[F_1]$ and f_2 in $T_2|_{\{\uparrow/f_2\}}$ equals $[F_2]$. The statements $(f'\,\langle \text{TST}\rangle) \doteq f_1$ and $(f'\,\langle \text{TST}\rangle) \doteq (f_2\,\langle \text{CAT ARG}\rangle)$ in E' show that the AVMs $\begin{bmatrix} \text{CAT} & x \\ \text{FEAT} & [F] \end{bmatrix}$ and $\begin{bmatrix} \text{CAT} & u \\ \text{FEAT} & [F'] \end{bmatrix}$ unify. Hence the CUG-categories $\mathrm{X}\,[F]$ and $\mathrm{U}[F']$ unify.

According to the AVM denoted by the metavariable $\uparrow$ in the annotation T_2, the value of path $\langle$CAT VAL CAT$\rangle$ is v, and the value of attribute FEAT is $[H']$. However, the value of the attribute CAT in the f-description E is y. The value of this attribute equals the value of path $\langle$CAT VAL CAT$\rangle$ in $T_2|_{\{\uparrow/f_2\}}$ after the unification at the attribute TST has taken place. Similarly, after the unification at the attribute TST has taken place, the value of attribute FEAT in the f-description E is $[F]$. This shows that the unification of $\mathrm{X}\,[F]$ and $\mathrm{U}[F']$ turns the CUG-category $\mathrm{V}\,[H']$ into $\mathrm{Y}\,[H]$. Hence the left application of the CUG-categories $\mathrm{X}\,[F]$ and $(\mathrm{U}[F'] \underset{[H']}{\backslash} \mathrm{V})$ yields $\mathrm{Y}\,[H]$. □

The following lemma states roughly that a CUG-constituent with mother $\mathrm{Y}\,[H]$, left daughter $\mathrm{V} \underset{[H']}{/} \mathrm{U}[F']$ and right daughter $\mathrm{X}\,[F]$ can be constructed iff an LFG-constituent can be constructed, whose top nonterminal is A with AVM f_0 and whose bottom nonterminals are A and A with AVM f_1 with AVM f_2, respectively

6.3.6. LEMMA. *Let the simulation map* $\mathrm{V} \underset{[H']}{/} \mathrm{U}[F']$ *onto the annotation* T_1, $\mathrm{X}\,[F]$ *onto the annotation* T_2, *and* $\mathrm{Y}\,[H]$ *onto* T_0, *and let there be a nonterminal* A *with AVM* f_1 *and f-description* $T_1|_{\{\uparrow/f_1\}}$, *and another nonterminal* A *with AVM* f_2 *and f-description* $T_2|_{\{\uparrow/f_2\}}$.

The right application of the CUG-categories $\mathrm{V} \underset{[H']}{/} \mathrm{U}[F']$ *and* $\mathrm{X}\,[F]$ *yields* $\mathrm{Y}\,[H]$ *iff nonterminal* A *with AVM* f_0 *and f-description* E *derives, by rules* R_b *and* R_r, *the sequence of nonterminals* A A *with* f_1, $T_1|_{\{\uparrow/f_1\}}$ *and* f_2, $T_2|_{\{\uparrow/f_2\}}$, *respectively, and the AVM* f_0 *in* E *equals the AVM denoted in* T_0.

Proof. Similar to the proof of Lemma 6.3.5. □

6.3.7. LEMMA. *Given a CUG-grammar* G. *Let* w *be a string and* $\mathrm{Y}\,[H]$ *be a CUG-category. Let the simulation map* $\mathrm{Y}\,[H]$ *onto* T_0. *CUG-category* $\mathrm{Y}\,[H]$ *produces string* w *in* G *iff the nonterminal* A *with AVM* f_0 *and f-description* $T_0|_{\{\uparrow/f_0\}}$ *produces string* w *in LFG-grammar* $f(G)$.

Proof. By induction on the length of the string w.

String w is in the lexicon. The CUG-lexicon contains entry $w : \mathrm{Y}\,[H]$ iff the LFG-lexicon contains entry (w, A, T_0). Hence CUG-category $\mathrm{Y}\,[H]$ produces string w iff nonterminal A with AVM f_0 and f-description $T_0|_{\{\uparrow/f_0\}}$ produces w.

String w is not in the lexicon. Because string w is not in the lexicon, $w = w_1 w_2$, and CUG-category $\mathrm{Y}\,[H]$ produces string w iff $\mathrm{Y}\,[H]$ derives sequence $\mathrm{U}\,[F']\ \mathrm{V}\,[H']$ such that $\mathrm{U}\,[F']$ produces w_1 and $\mathrm{V}\,[H']$ produces w_2. Let the simulation map $\mathrm{Y}\,[H]$ onto T_0, $\mathrm{U}\,[F']$ onto T_1, and $\mathrm{V}\,[H']$ onto T_2. By induction the nonterminal A with AVM f_1 and f-description $T_1|_{\{\uparrow/f_1\}}$ produces string w_1 and the nonterminal A with

AVM f_2 and f-description $T_2|_{\{\uparrow/f_2\}}$ produces string w_2 iff U $[F']$ produces w_1 and V $[H']$ produces w_2. By either Lemma 6.3.5 or Lemma 6.3.6 the nonterminal A with f-description T_0 derives A A with AVM f_1 and f-description $T_1|_{\{\uparrow/f_1\}}$, and AVM f_2 and f-description $T_2|_{\{\uparrow/f_2\}}$. Hence nonterminal A with f-description T_0 and AVM f_0 produces $w = w_1 w_2$. iff CUG-category Y $[H]$ produces string $w = w_1 w_2$. □

6.3.8. THEOREM. *Given a CUG-grammar G and string w. Let the simulation map G onto LFG-grammar f(G). CUG-grammar G generates string w iff LFG-grammar f(G) generates string w.*

Proof. The LFG-grammar generates string w iff the distinguished nonterminal S produces w. According to rule R_s S derives A with AVM f_0 and f-description $(f_0\langle \textsc{cat}\rangle) \doteq \mathrm{S}$.

The CUG-grammar G generates string w iff the distinguished category S produces w. The category S is mapped onto the annotation $(\uparrow \textsc{cat}) \doteq \mathrm{S}$. According to Lemma 6.3.7 CUG-category S produces string w iff nonterminal A with AVM f_0 and f-description $(f_0\langle \textsc{cat}\rangle) \doteq \mathrm{S}$ produces w.

Hence the CUG-grammar G and the LFG-grammar $f(G)$ generate the same strings. □

6.4 Formal Properties

In this section we consider two formal properties. First, we handle the complexity of the recognition problems of LFG. Second, we handle the weak generative capacity of LFG.

6.4.1 Complexity of the Recognition Problems

A lower bound on the complexity of the universal recognition problem of LFG is known for some time. Barton, Berwick and Ristad (1987) proved an *NP*-hard lower bound. We will now prove the stronger result that the recognition problem for a fixed LFG-grammar G is *NP*-hard. Furthermore, we will also prove an *NP* upper bound on the complexity of the universal recognition problem. These two proofs together determine the complexity of the recognition problems exactly.

A lower bound on the complexity. A lower bound on the complexity of the recognition problems results from the simulation by the following argumentation. Given any CUG-grammar, the simulation f provides us with an LFG-grammar that recognizes the same language as the CUG-grammar. Thus the mapping from a CUG-grammar G onto an LFG-grammar $f(G)$ is a many-one reduction. Moreover, Lemma 6.4.2 proves that the mapping is a polynomial time, many-one reduction.

In Lemma 6.4.2 we consider the costs of computing the grammar LFG-grammar $f(G)$ that simulates the CUG-grammar G. Before we will consider the costs of the total simulation, let us consider the cost of the auxiliary functions h_1, h_2, and h_3.

6.4.1. LEMMA. *The auxiliary functions h_1, h_2, and h_3 compute a set of statements from a given CUG-category in quadratic time, with respect to the size of this CUG-category.*

Proof. The function h_1 maps a given CUG-category onto an AVM. Next the function h_2 maps this AVM onto a set of equations. Finally the function h_3 maps this set of equations onto a set of statements.

The function h_1 decomposes a complex CUG-category into two smaller CUG-categories. After an linear amount of steps the CUG-category is decomposed in primitive CUG-categories. Each primitive CUG-category is mapped onto an AVM, and these AVMs are composed in linear time into the AVM that corresponds to the original CUG-category. The total computation of h_1 takes linear time, with respect to the size of the CUG-category.

The function h_2 regards the AVM as a tree instead of a graph. Multiple occurrences of the same box-label are considered to be different nodes. So there is a unique path from the root of the tree to a node. These paths are at most as long as the size of the AVM: the length of each path is linear. The number of nodes in the tree is smaller than the size of the AVM. Hence the set of equations is computable in quadratic time, and has quadratic size with respect to the size of the AVM, i.e., with respect to the size of the CUG-category.

The function h_3 partitions a set of equations into subsets. Then the function adds a metavariable to each equations, and conjoint some equations. The total computation of h_3 takes linear time in the size of the set of equations. Hence the composition of the auxiliary functions h_1, h_2, and h_3 compute a set of statements from a given CUG-category in quadratic time, with respect to the size of this CUG-category. □

6.4.2. LEMMA. *The LFG-grammar $f(G)$ that simulates a CUG-grammar G is computed in quadratic time, with respect to the size of the CUG-grammar G.*

Proof. The mapping from the lexicon of the CUG-grammar onto the lexicon of the LFG-grammar has the largest impact on the cost of the computation of the LFG-grammar. This mapping depends mainly on the mapping from CUG-categories onto set of statements. This latter mapping costs quadratic time, with respect to the size of the CUG-category. So the mapping from the CUG-lexicon costs quadratic time with respect to the size of the CUG-grammar. Hence the LFG-grammar is computed in quadratic time, with respect to the size of the CUG-grammar G. □

So Theorem 6.3.8 and Lemma 6.4.2 show that the simulation of CUG by LFG that we presented in Section 4.2 is in fact a polynomial time, many-one reduction. By Theorem 3.3.22, the recognition problem of CUG for a fixed grammar is NP-hard. Hence the recognition problem of LFG for a fixed grammar is NP-hard, as indicated by the following theorem.

6.4.3. THEOREM. *The recognition problem of LFG for a fixed grammar is NP-hard.*

Proof. According to Theorem 6.3.8, the LFG-grammar $f(G)$ recognizes the same language as the CUG-grammar G presented in Section 3.3.2. Lemma 6.4.2 shows that the LFG-grammar $f(G)$ is computable in polynomial time, with respect to the size of G. Hence the recognition problem of LFG for a fixed grammar is as hard as the recognition problem of CUG for a fixed grammar. Theorem 3.3.22 proves that the fixed recognition problem of CUG is NP-hard. Hence the recognition problem of LFG for a fixed grammar is NP-hard. □

Because the universal recognition problem is at least as hard as the fixed recognition problem, as an immediate consequence, we obtain the result proven by Barton, Berwick and Ristad (1987): the universal recognition problem of LFG is NP-hard.

6.4.4. THEOREM. **(Barton, Berwick and Ristad 1987)** *The universal recognition problem of LFG is NP-hard.*

An upper bound on the complexity. Theorems 6.4.3 and 6.4.4 present lower bounds on the complexity of the recognition problems of LFG. Obviously, we would also like to have an upper bound on the complexity of the recognition problems. In the ideal situation we would provide an upper bound for general versions of LFG. A prerequisite for such an upper bound is that a complete description of LFG is available. Alas no such complete description of LFG exists. So we have to settle for the practical situation, and consider fragments of LFG. Let us first consider the fragment of LFG presented in Section 6.1. Then we will consider polynomial extensions of this fragment, and argue that these extensions suffice to describe full LFG-grammars.

In Section 6.1 we showed that this fragment of LFG is based on the standard feature theory from the introduction. This standard feature theory has a nice computational property: the unification problem is tractable. We indicated in Section 3.2.2 that minor changes to Smolka's (1992) constraint solving algorithm yield a polynomial time unification algorithm for AVMs. To be more precise, the unification of two AVMs $[F]$ and $[H]$ is computable in polynomial time, with respect to the sizes of $[F]$ and $[H]$. Moreover, the amount of space used by the unification is linear with respect to the sizes of $[F]$ and $[H]$.

We can easily prove an NP upper bound on the complexity of the universal recognition problem of this restricted fragment of LFG. The instances of the universal recognition problem are an LFG-grammar G and a string w (see Definition 3.2.2). The LFG-grammar G consists of a lexicon, a set of combinatory rules, and a distinguished nonterminal.

The proof of the NP upper bound is based on the following two observations. First, a derivation for a string w and an LFG-grammar G consists of a polynomial amount of steps. This first observation is proven in Lemma 6.4.5. Second, the reversed application of a derivation step is computable in polynomial time. We can prove this second observation by means of the off-line view on derivations (see Lemma 6.4.6). In the off-line view, a derivation consists of two phases. In the first phase, a total derivation-tree for the constituent structure tree is constructed and f-description statements are collected into an f-description E. In the second phase, the AVMs described by the f-description E are computed. Both phases are computable in polynomial time.

6.4.5. LEMMA. *A derivation for a string w and a grammar G has polynomial size, with respect to the sizes of w and G.*

Proof. The derivations of an LFG-grammar are short by the "Definition of a Valid Derivation." This definition states that a constituent structure is valid, only if no category appears twice in a non-branching dominance chain (Kaplan and Bresnan 1982, p.266). It follows that the number of steps in a derivation for a string w is at most linear with respect to the size of the string and the size of the grammar. $\square$

Now given a string w and an LFG-grammar G, we guess a linear amount of entries from the lexicon, and a polynomial sized sequence of combinatory rules that encode the derivation for w. The previous lemma shows that both guesses can be checked in polynomial time. Hence the following lemma holds.

6.4.6. LEMMA. *The universal recognition problem of LFG for string w and LFG-grammar G is computable in nondeterministic polynomial time.*

Proof. A derivation for the string w is described by a sequence of lexicon entries and a sequence of applied combinatory rules. We will guess both sequences and check in polynomial time that the described derivation indeed yields w.

We can guess a sequence of entries from the lexicon for the string w. These entries have a total size that is linear in the size of the string and the grammar. So the check that the entries form the string w takes linear time.

We can guess a sequence of combinatory rules that complete the description of the derivation. The size of this sequence of rules is polynomial in the size of the string w and the size of the grammar G. Given both sequences of polynomial size, we can compute the derivation for w in polynomial time, in the following way.

Given a sequence of rules and lexicon entries, the constituent structure tree is constructed in linear time, with respect to the size of the sequence. The number of f-description statements is also linear in the size of the sequence. So given the sequence of rules and lexicon entries, the first phase is computable in polynomial time.

Given the f-description E, the second phase is computable in polynomial time as indicated in Section 3.2. Hence given a sequence of rules and lexicon entries, the derivation steps are computable in polynomial time. The final check that the distinguished nonterminal produces w takes linear time. $\square$

Let us now consider extensions of the fragment of LFG that we presented in Section 6.1. We consider "polynomial" extensions in which the reversed application of derivation steps remain computable in polynomial time, and in which the derivations remain of polynomial size. Clearly, the recognition problems of these extensions are also computable in nondeterministic polynomial time. We claim that these polynomial extensions may contain the following devices (Shieber 1986):

1. Lexicon entries for the empty string, which are needed to treat long-distance dependencies. The Definition of a Valid Derivation (Kaplan and Bresnan 1982, p. 266) bounds the number of empty strings in a derivation for a string w linearly to the size of w.
2. Context-free rules that are extended with regular expressions as described in Section 6.1.1.
3. Set values, which allow attributes to take sets of AVMs as their values.
4. Constraint equations, which are used to guarantee a value for an attribute without specifying that value. These equations play a major role in the use of attribute PRED (Section 6.1.1).
5. Disjunctions of values and AVMs.
6. Long-distance metavariables that are used in the analysis of unbounded dependencies.

The devices 1, 2, 5, and 6 only require a polynomial increase of the number of guesses and some extra bookkeeping. The use of device 3 is limited to tests for membership and simple operations, which require polynomial time, like union. The device 4 seems to require only a simple test on the final result of a derivation.

The above seems to indicate that the devices used by full LFG-grammars increase neither the size of a derivation, nor the time to compute derivation steps, in a substantial way. Therefore we think that it is justified to state that in general (i) the derivations of LFG-grammars have polynomial size; (ii) the reversed application of derivation steps is computable in polynomial time. Thus, we claim that the universal recognition problem of LFG-grammars is computable in nondeterministic polynomial time. Hence by Theorems 6.4.3 and 6.4.4:

6.4.7. THEOREM.
(i) *The fixed recognition problem of LFG is NP-complete.*
(ii) *The universal recognition problem of LFG is NP-complete.*

6.4.2 Weak Generative Capacity

We will now determine a lower bound and an upper bound on the weak generative capacity of LFG. The lower bound results from the simulation. The upper bound results from the upper bound of the complexity of the fixed recognition problem. These two bounds on the weak generative capacity are represented in Theorem 6.4.8.

A lower bound on the weak generative capacity. Theorem 6.3.8 proves that for every CUG-grammar there exists an LFG-grammar that recognizes the same language. Hence the set of languages recognized by a CUG-grammar, the *CUG-languages*, is a subset of the set of languages recognized by an LFG-grammar, the *LFG-languages*. So if we can prove the location of the CUG-languages in the Chomsky hierarchy, we know that the LFG-languages take the same or a higher location in the Chomsky hierarchy. For instance, it is known that the set of languages that the classical categorial grammars generate is the set of context-free languages. Therefore the CUG-languages generate all context-free languages. Hence the LFG-languages generate all context-free languages.

An upper bound on the weak generative capacity. Once again, we are unable to provide an upper bound for general versions of LFG. A prerequisite for such an upper bound is a complete description of LFG, which is not available. So we settle for the polynomial extensions of the fragment presented in Section 6.1.

In Chapter 2 we connect the complexity of the recognition problem and weak generative capacity of restricted attribute-value grammars. These restricted attribute-value grammars respect a liberal variation on the so-called *off-line parsability constraint*. The off-line parsability constraint (OLP) (Johnson 1988) is a generalization of the "Definition of a Valid Derivation." The OLP relates the amount of "work" done by the grammar to produce a string linearly to the number of terminal symbols produced. It is therefore a sort of honesty constraint that is common in complexity theory.

In Chapter 2 a variation on the off-line parsability constraint is introduced: the *honest parsability constraint*. This honest parsability constraint is a more liberal constraint than the off-line parsability constraint. The honest parsability constraint relates the amount of "work" done by the grammar to produce a string polynomially to the number of terminal symbols produced. For convenience, we state the definition of the honest parsability constraint, Definition 2.5.1, below.

> *A grammar G satisfies the honest parsability constraint iff there exists a polynomial p such that for each string w in the language generated by G there exists a derivation with at most $p(|w|)$ steps.*

A nice property of the restricted attribute-value grammars that respect the honest parsability constraint is that they generate exactly all languages in the complexity class *NP*. Or, as we can restate Theorem 2.5.3:

> *Let L be a language that has an NP recognition algorithm. Then there exists a restricted attribute-value grammar G, that respects the honest parsability constraint, such that G generates the language L.*

By Theorem 6.4.7 the restricted fragment of LFG has an *NP* recognition problem. Hence any restricted LFG-grammar can be simulated by a restricted attribute-value grammar. Thus the languages generated by restricted attribute-value grammars that respect the honest parsability constraint provide an upper bound on the weak generative capacity of restricted LFG-grammars. Hence the following theorem holds.

6.4.8. THEOREM. *The weak generative capacity of the restricted fragment of LFG presented in this chapter is enclosed between the context-free languages and the collection of languages generated by restricted attribute-value grammars that respect the honest parsability constraint.*

Moreover, we conjecture a stronger upper bound for the weak generative capacity of LFG. A close look at the proof of Lemma 6.4.6 shows that the fixed recognition problem of LFG is computable in nondeterministic linear space. We conjecture that the polynomial extensions of the fragment presented in Section 6.1 are also computable in nondeterministic linear space. This implies that the LFG-languages are recognized by linear bounded automata (LBAs). Due to the equivalence of LBAs and context-sensitive grammars, LFG generates only context-sensitive languages (CSLs). Hence we conjecture that the collection of CSLs that are recognized in nondeterministic polynomial time form an upper bound for the weak generative capacity of LFG (cf., Book 1978). Or stated differently:

6.4.9. CONJECTURE. *The collection of context-sensitive languages that the restricted attribute-value grammars that respect the honest parsability constraint generate form an upper bound for the weak generative capacity of LFG.*

Seki et al. (1993) and Nakanishi et al. (1992) also study the complexity and generative capacity of LFG. They take a generalization of Kaplan and Bresnan's (1982) original formulation as a starting point. Seki et al. (1993) introduce two polynomial time recognizable generalizations of context-free grammars: parallel multiple context-free grammars and multiple context-free grammars. Then they introduce three restricted versions of their generalization of LFG: nondeterministic copying LFGs, deterministic copying LFGs, and finite copying LFGs. Finally they prove that the following generalizations generate the same set of languages: (*i*) the multiple context-free grammars and the finite copying LFGs, and (*ii*) the parallel multiple context-free grammars and the deterministic copying LFGs. Furthermore, they show that the nondeterministic copying LFGs generate an *NP*-complete language.

Seki et al.'s (1993) approach to LFG and our approach to LFG depart from different principles. Seki et al. (1993) and Nakanishi et al. (1992) restrict the AVMs, but not the constituent structures. For instance, the admissible annotations of rules in nondeterministic copying LFGs are only "$\uparrow$ ATTR $\doteq\downarrow$" and "$\uparrow$ ATTR $\doteq$ *value*." Hence the AVM of a mother nonterminal is always larger than the AVMs of its daughters. However, Seki et al. (1993) and Nakanishi et al. (1992) neglect the Definition of a Valid Derivation. Contrariwise, we follow the standard literature and restrict the constituent structures. We include the Definition of a Valid Derivation, which states that no nonterminal appears twice in a non-branching dominance chain. Further research should indicate to what extent these two approaches may complement each other.

Chapter 7

Complexity of Functional Grammar

7.1 Introduction

7.1.1 Functional Grammar

Dik (1978) developed the theory of Functional Grammar (FG) in the late 70's. The version of FG that is mostly used nowadays is Dik's (1989) major revision. This chapter is also mainly based on that version of FG.

The theory of Functional Grammar stems from predicate logic, and can be seen as a reaction to the Transformational Grammar of Chomsky. This reaction manifests itself by forbidding transformations, like deletions or movements. For instance, the sentence "I consider John unfit for the job" is not derived from the sentence "I consider John to be unfit for the job" by deleting the phrase "to be." Also, the sentence "I believe John to be unfit for the job" is not derived from the sentence "I believe that John is unfit for the job" by moving John from subject position in the subordinate sentence to the object position in the main sentence.

An FG-grammar consists of several hierarchically organized processes, each with its own task (see Figure 7.1). At the lowest level there is the *lexicon*, which consists of a finite set of basic predicates and basic terms. The process called fund formation constructs derived predicates and derived terms from the basic predicates and terms in the lexicon. These derived predicates and terms form the *fund*. A further process, called clause structure formation, combines predicates and terms from the fund with operators and functions, which are taken from some fixed set, into a *clause structure*. These clause structures serve as the underlying representations of sentences. Two further processes are needed to come from an underlying representation to a *sentence*. First the lexical form of the words in the sentence are specified by the form specification process. Next the order of these words within a sentence is determined by the order specification process.

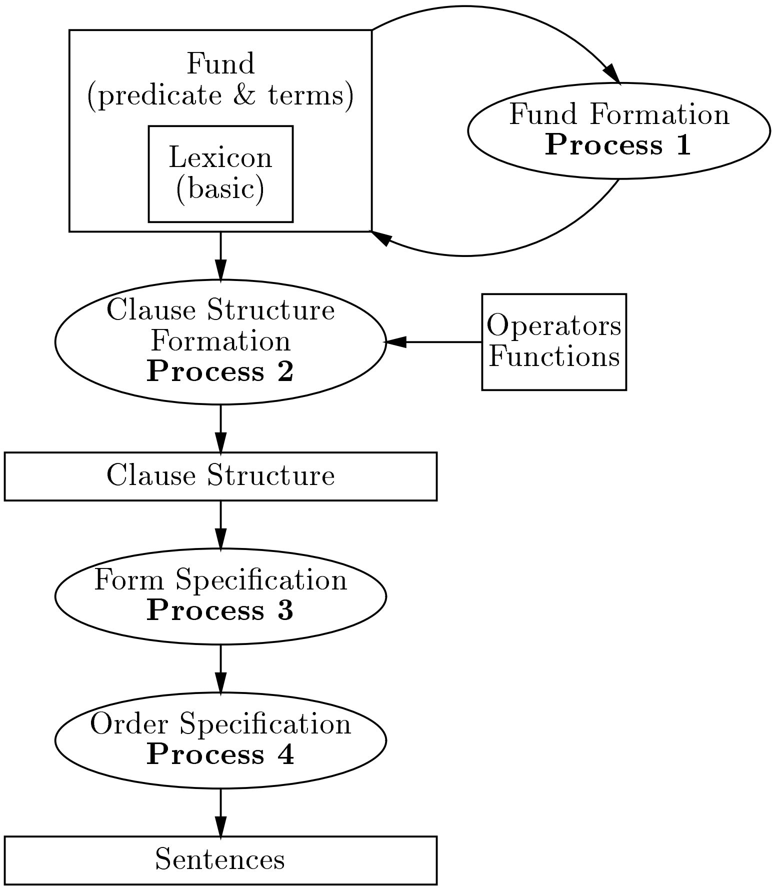


Figure 7.1: The generation process of Functional Grammar.

7.1.2 Complexity Theory

In complexity theory one tries to determine the complexity of problems. The complexity is measured by the amount of time and space needed to solve a problem. Usually, one considers *decision problems*, i.e., problems that are answered "Yes" or "No." For instance, the problem whether a given grammar generates a given string is a decision problem. Not all problems can in fact be solved. A well-known problem from formal language theory for which no algorithm exists that solves the problem is the *Halting Problem* for a Turing machine M and string w. Such problems are called *undecidable* problems. In complexity theory we are mostly interested in problems that can be solved. These problems are called *decidable* problems. (See any textbook on formal language theory for further definitions (e.g., Hopcroft and Ullman 1979, Partee, ter Meulen and Wall 1990)).

Some complexity classes. Often we are interested in the distinction between tractable and intractable problems. A problem is *tractable* if its solution requires an amount of steps that is polynomial in the size of the input: we say that the problem requires *polynomial time*. Likewise, we speak of linear time, quadratic time, etcetera. The class of all tractable problems is called P. Occasionally we will refer to the tractable problems as *P problems*. The intractable problems are called *NP-hard problems*. The easiest intractable problems are the *NP-complete problems*. The *NP*-complete problems are contained in the complexity class *NP*. Solutions for problems in *NP* can be guessed and checked in polynomial time. It is strongly believed that the complexity class P and the complexity class *NP* are different, although this has not been proven. In fact this is the major open problem in complexity theory.

Honest functions & oracles. Honest functions and oracles are familiar notions from complexity theory. A function is *honest* if the size of the input is polynomial in the size of the output. The input of an honest function that is also computable in polynomial time is neither much larger, nor much smaller than the output. The honest functions appear in several places in this chapter. They are important because honest functions are a flexible and abstract formalization for operations that manipulate structures without deleting a substantial amount of material. So the honest functions correspond nicely with the FG principles, which forbid deletions.

An *oracle* can be seen as a machine that solves questions of a particular problem. The user of an oracle does not know (and need not know) how the oracle performs its computation. The user only needs to know in which form questions to the oracle have to be asked. The oracle takes the question as input and outputs, magically, the correct answers to the question. The magic is that the whole computation of the question takes only one step. So, consulting an oracle costs only one step more than the amount of steps that are needed to write down the question. The oracle is a nice way to abstract from some complexity in a problem that is, in fact, irrelevant for complexity of the actual problem.

The precise definitions, as well as some nice characteristics, of honest functions and oracles may be found in (Balcázar, Díaz and Gabarró 1988).

Many-one reductions & 3-Satisfiability. There is a direct manner to determine the upper bound complexity of a problem, if there is an algorithm that solves the problem: determine the complexity of that algorithm. An indirect way to determine the lower bound complexity of a problem is the *reduction*. A reduction from some problem A to some problem B maps instances of problem A onto instances of problem B. The intuition behind reductions is that a solution for problem B helps to solve problem A. Therefore, if the reduction is simple, problem B is at least as hard as problem A.

The reductions that we will consider are known as *polynomial time, many-one reductions*. These many-one reductions are subject to two conditions: (*i*) the reductions are easy to compute, and (*ii*) the reductions preserve the answers. A reduction from A to B is *easy* to compute, if the mapping takes polynomial time. A reduction *preserves answers* if the answer to the instance of A is the same as the answer to the instance of B. That is, the answer to the instance of A is “Yes” if and only if the answer to the instance of B is also “Yes.”

A reduction is an elegant way to classify a problem as intractable. Suppose problem B is a problem with unknown complexity. Let there be a reduction f from an *NP*-hard problem A to problem B. Furthermore, let f conform to the two conditions above. By an indirect proof, it follows from this reduction that B is at least as hard as A. Hence B is also an *NP*-hard problem. If we also prove that we can guess a solution for B and check that guessed solution in polynomial time, then B is an *NP*-complete problem.

A well-known *NP*-complete problem is 3-SATISFIABILITY (3SAT):

7.1.1. DEFINITION. 3-SATISFIABILITY
INSTANCE: A formula φ, from propositional logic, in 3-conjunctive normal form, where each disjunction consists of exactly three literals.
QUESTION: Is there an assignment of truth-values to the propositional variables of φ, such that φ is true?

The instances of 3-SATISFIABILITY are formulas in *3-conjunctive normal form*, i.e., the formulas are conjunctions of *clauses*. The clauses are disjunctions of exactly three literals, and the *literals* are positive and negative occurrences of propositional variables. We call formula φ a *satisfiable* formula if an assignment exists that makes formula φ true.

An assignment assigns either the value true or false to a propositional variable. Given such an assignment, we can determine the truth-value of a formula. The formula $\varphi = (\gamma_1 \wedge \ldots \wedge \gamma_m)$ is true iff each clause, γ_i, is true. A clause $\gamma = (l_1 \vee l_2 \vee l_3)$ is true iff at least one literal, l_i, is true. A positive literal, $l_i = p_j$, is true iff the variable p_j is assigned the value true. A negative literal, $l_i = \overline{p_j}$, is true iff the variable p_j is assigned the value false.

Universal version versus fixed version. One of the most significant works on complexity theory and natural language is (Barton, Berwick and Ristad 1987). They recognize that there are two versions of problems for grammatical formalisms. In the general case, a problem concerns an arbitrary string and an arbitrary grammar. This version of a problem is called *universal*. A more specific version of a problem results when one considers a fixed grammar instead of an arbitrary one. This version is often called *fixed*.

A universal version of a problem is always at least as difficult as its fixed version. The universal version may be more difficult. An algorithm that solves the universal version efficiently also solves the fixed version efficiently. On the other hand, an efficient algorithm for the fixed version may take an amount of time that is exponential in the size of the grammar. Clearly, such an algorithm does not solve the universal version efficiently.

7.1.3 Outline of this chapter

A computationally attractive property of FG is its aversion to transformations. That is, no changes are allowed within structures that are composed at an earlier stage.

In this chapter we investigate the computational complexity of parsing in FG. The complexity of FG has not been examined before. Therefore the search for a global impression of its complexity is a well-founded goal for this investigation. The *universal version* of a problem provides such a global impression. Therefore we will consider the complexity of the universal version of the various problems in FG.

A prerequisite for complexity analyses is a formalization of the problem. Because the formalization of FG is still very sketchy, we often have to formalize FG ourselves. As a consequence, a large part of this chapter will be devoted to the formalization of FG. The starting point of our formalization of FG is (Dik 1989). The ideas from Bakker (1994) and Kwee (1994) also have a large impact on the formalization. This latter work shows that our formalization of FG might have difficulties with complex clauses. We have not taken these complex clauses into consideration, because Dik (1989) postpones the discussion of complex clauses until the follow-up, which is called TFG2. We conjecture that if the treatment of complex clauses in TFG2 fits within the theory laid down by Dik (1989), then our formalization will also be able to treat complex clauses. Future research will have to prove or disprove this conjecture.

The structure of this chapter follows the structure of an FG-grammar. In the next four sections (Sections 7.2–7.5) we will consider the four processes (see Figure 7.1) one after the other. Each of these sections is divided into three parts.

- The first part consists of examples that should clarify the process. These examples serve as a starting point for the formalization in the second part.
- The second part starts with a formalization of the rules and the structures that play a role in the process. Then we compare the formalization and the examples from the first part. Our main attention goes out to the differences in capacity. To be more precise, the descriptions that are used in the examples, but cannot be expressed by the formalization, and the descriptions that can be expressed by the formalization, but were not found in the examples. This part ends with a formulation of the process as a decision problem.
- In the third part, we prove the complexity bounds of the process under this formalization, and we propose some restrictions that lower the complexity.

7.2 Fund Formation Process

A sketch of the situation. An FG-grammar contains a finite collection of terms and predicates, which is called the *lexicon*. The lexicon contains all and only all basic terms and basic predicates. An unbounded supply of derived terms and predicates can be made by the *fund formation rules* (FFRs). The combination of all basic predicates and terms, and all derived predicates and terms form the *fund*.

7.2.1 An Informal Introduction

The lexicon. The lexicon consists of basic terms and basic predicates. The basic terms are pronouns and proper names. The pronoun "I" is represented as:

$$(\mathtt{Sp1}\, x).$$

This basic term consists of two components: the two *term operators* Sp and 1, and the *variable* x. The term operator Sp states that entity x is the speaker, rather than the addressee. The term operator 1 states that entity x is singular, as opposed to plural. Thus the basic term $(\mathtt{Sp1}\, x)$ denotes the singular speaker: "I."

The other sort of basic terms are proper names. As an example of a proper name consider the representation of "John":

$$(\mathtt{d1}x : \mathrm{john}(x)).$$

In this basic term two components are worth mentioning: the *body* and the *term operators*. The body of this basic term is "john," indicating that the entity x is a "john." The basic term contains two term operators, denoted by d1. The operator d stands for "definite," the operator 1 for "singular." So the entity x is definite and singular. Thus the basic term denotes the definite, singular entity x, such that x is "john."

The basic predicates come in more variety than the terms. There are, for instance, basic adjectival, nominal and verbal predicates. As an example consider the representation of the basic verbal predicate for "to kiss."

$$\begin{gathered}\mathrm{kiss}_V(x_1 : \langle\mathrm{animate}\rangle(x_1))_{\mathtt{Ag}}(x_2 : \langle\mathrm{animate}\rangle(x_2))_{\mathtt{Go}} \\ \overset{mp}{\rightarrow} \\ \mathrm{touch}_V(x_1 : \langle\mathrm{animate}\rangle(x_1))_{\mathtt{Ag}}(x_2 : \langle\mathrm{animate}\rangle(x_2))_{\mathtt{Go}}(\mathtt{dm}x_3 : \mathrm{lip}_N(x_3))_{\mathtt{Instr}}\end{gathered}$$

We distinguish four components in the predicate: the *category*, the *body*, the *argument positions*, and the *meaning postulate*. The last two components are complex. We will explain them in a moment, but we start with explaining the two simple components.

- The predicate is a verbal predicate, which is indicated by its category V.
- The body of this basic predicate is "kiss," i.e., this predicate denotes a kiss event.
- The predicate takes two arguments, indicated by the two variables x_1 and x_2 in the argument positions $(x_1 : \langle\mathrm{animate}\rangle(x_1))_{\mathtt{Ag}}$ and $(x_2 : \langle\mathrm{animate}\rangle(x_2))_{\mathtt{Go}}$. The first one bears the *semantic function* Ag, which identifies the agent. The second one bears the semantic function Go, which identifies the goal. The argument positions also indicate that the predicate cannot take all kinds of arguments. Both positions are restricted to animate beings, which is expressed by the *selection restriction* $\langle$animate$\rangle$. The number of arguments that a predicate takes is called its *arity*.

- The representations of the meaning postulate and the rest of this basic predicate are separated by the symbol $\overset{mp}{\rightarrow}$. We confine ourselves to a sloppy representation of the meaning postulate in which the semantic function `Instr` identifies the "Instrument." The meaning postulate is intended to state that if x_1 kisses x_2 then x_1 touches x_2 with the lips.

The fund. The lexicon is the source of the fund. All predicates and terms of the lexicon are contained in the fund. The fund formation rules extend the fund with derived terms and predicates (cf., Dik 1980). Some typical examples of FFRs that create derived terms are the following.

- Turn a nominal predicate into a term, e.g., turn the nominal predicate "man" ($\text{man}_N(x)$) into the term "the man":

$$(\mathtt{d1}x: \ \text{man}_N(x)).$$

- Add an operator to a term. For instance, add the diminutive operator `Dim` to the term "the book"

$$(\mathtt{d1}x: \ \text{book}_N(x))$$

  in order to obtain the term "the booklet":

$$(\mathtt{Dim}\,\mathtt{d1}x: \ \text{book}_N(x)).$$

- Place a restriction on a term, e.g., the term "the man" ($\mathtt{d1}x: \ \text{man}_N(x)$) may be restricted by an adjectival predicate "old" with one free variable ($\text{old}_A(x)$), turning it into "the old man":

$$(\mathtt{d1}x: \ \text{man}_N(x): \ \text{old}_A(x)).$$

More complicated cases of fund formation appear in the creation of derived predicates. Some typical examples of FFRs that create derived predicates are the following.

- Combine a nominal and a verbal predicate into a verbal predicate if the nominal predicate conforms to the selection restriction imposed by the argument position that the nominal predicate will fill. For instance, the verb "to watch" and the noun "bird"

$$\text{watch}_V(x_1: \langle\text{animate}\rangle(x_1))(x_2: \langle\text{animate}\rangle(x_2)) \ \text{bird}_N(y_1)$$

  can be combined into a verb that means "bird watching"

$$\{\text{bird}_N - \text{watch}_V\}_V(x_1: \langle\text{animate}\rangle(x_1)),$$

  because the noun conforms to the selection restriction of the second argument position.

- Remove the argument position that is identified as the semantic goal of a verbal predicate that indicates an action with a clear endpoint. For instance, from the transitive form of the verb "eat"

$$\text{eat}_V(x_1 : \langle\text{animate}\rangle(x_1))_{\texttt{Ag}}(x_2)_{\texttt{Go}}$$

  the intransitive form can be obtained (cf., Bakker 1994).

$$\text{eat}_V(x_1 : \langle\text{animate}\rangle(x_1))_{\texttt{Ag}}.$$

- Change the semantic function of an argument position of a verbal predicate if the predicate indicates an action by which something is applied to some surface in such a way that the surface gets covered with something as a result of the action. For instance, the predicate for "x_1 smears x_2 on x_3"

$$\text{smear}_V(x_1)_{\texttt{Ag}}(x_2)_{\texttt{Go}}(x_3)_{\texttt{Loc}}$$

  may be turned into a predicate for "x_1 smears x_3 with x_2"

$$\text{smear}_V(x_1)_{\texttt{Ag}}(x_3)_{\texttt{Go}}(x_2)_{\texttt{Instr}}.$$

- Change a verbal predicate whose argument position is identified as the semantic agent into a nominal predicate. For instance, the verb "walk"

$$\text{walk}_V(x_1)_{\texttt{Ag}}$$

  may be turned into the noun "walker"

$$\{\text{walk}_N\text{-er}\}_N(x_1).$$

- Turn a term into a predicate by extending the term with an argument position. For instance, the possessive "John's" ($(\{\texttt{d1}x_1 : \text{john}(x_1)\}_{\texttt{Poss}}(x_2))$) is a predicate that is derived from a term, in this case "John" $(\texttt{d1}x_1 : \text{john}(x_1))$.

The first and last examples show that combinations of FFRs can be applied recursively. The examples also show that the inputs of FFRs are one or more terms or predicates. The FFRs may change the body, the category, the arguments, and the meaning postulate. We also presented transformation-like examples that reduce the number of arguments. These transformation-like examples show that the fund formation rules are exceptional in FG. On the one hand, the principles of FG forbid changes within structures that are composed at an earlier stage. On the other hand, the fund formation rules seem to perform deletions. However, it is claimed that the FFRs do not violate the principles of FG because the FFRs describe and do not compose the fund, which simply exists. So in this sense, the FFRs do not change structures that are composed at an earlier stage, and hence do not violate the principles of FG. Thus the only reason that the FFRs are exceptional is because they describe, instead of construct, structures.

7.2.2 The Formalization

Before we will formalize the fund formation process, we make a brief remark about the *elementary entities* that are distinguished in an FG-grammar. These entities will appear every now and then in the formalizations to come. In the sequel of this chapter we will presuppose that these entities are specified by the particular FG-grammar. We assume a finite set of *strings* Σ; a finite set of *categories* C; a finite set of *functions* F, in which FG distinguishes three disjunct subsets: syntactic, semantic, and pragmatic functions; a finite set of *operators* Ω, in which FG distinguishes three distinct subsets: term, clause-structure, and auxiliary operators; and a countably infinite set of *variables* V that refer to either individuals or events.

In the previous part of this section we presented some examples that make clear what the typical fund formation rules can do. Now we will formalize the fund formation process. First we will formalize the notions of terms and predicates. Then we formalize the fund formation rules. Next we compare the formalization and the examples. We end with a formulation of the fund formation problem.

The lexicon is a finite set that contains all basic terms and all basic predicates. The basic terms are pronouns and proper names. We assume that in addition to the components stated in the examples above a meaning postulate is assigned to these basic terms.

7.2.1. DEFINITION. *Basic Terms*
In the structure of *pronouns* three components are distinguished:

- A set of term operators from Ω.
- A variable that refers to an individual from V.
- A meaning postulate, which is a partial description by semantically more primitive elements of the fund. We defer a further discussion of meaning postulates until after Definition 7.2.3.

In the structure of *proper names* four components are distinguished:

- A set of term operators from Ω.
- A variable that refers to an individual for V.
- A body, which is a unary predicate of some special category from C.
- A meaning postulate, which is a partial description by semantically more primitive elements of the fund. We defer a further discussion of meaning postulates until after Definition 7.2.3.

Though the basic terms are specified vaguely in FG, the derived terms are specified rather precisely. The structure of derived terms is similar to the structure of proper names.

7.2.2. DEFINITION. *Derived Terms*
In the structure of *derived terms* four components are distinguished:

- A set of term operators from Ω.
- A variable from V that refers to an individual.

- A meaning postulate.
- A body, which is a list of 1-open predications; a *1-open predication* is an n-ary predicate of which $(n-1)$ argument positions are filled with terms.

The predicates are also specified rather precisely in FG. The basic and derived predicates have the same structure.

7.2.3. DEFINITION. *Predicates & Argument Positions*
In the structure of predicates four components are distinguished. We distinguish the following components in a *predicate* of arity n (n-ary predicate):

- A body, which is a string from Σ for basic predicates.
- A category from C.
- A list of n argument positions, which are complex structures that we will formalize below.
- A meaning postulate, which is a partial description by semantically more primitive elements of the fund. We defer a further discussion of meaning postulates until after this definition.

The *argument positions* are complex structures. In each argument position three components are distinguished:

- A function from F.
- A variable from V.
- A selection restriction, which we assume to be a combination of zero or more unary predicates from the lexicon. The further discussion of selection restrictions is deferred until after this definition.

The *meaning postulates* are difficult to formalize because examples of meaning postulates in the FG-literature are rare. According to Dik (1989) the meaning postulate of a term or predicate is a partial description by semantically more primitive elements of the fund. The intention is the following: "if B is a meaning postulate of A, then B holds if A holds." We will call B a *meaning definition* when the meaning postulate B holds if and only if A holds. Since this chapter is about complexity theory, we should remark that the problem of determining a minimal set of the semantic primitive predicates and terms from a given lexicon is an *NP*-complete problem (Dailey 1986). Apart from what is stated in this section, the meaning postulates will be ignored.

A *selection restriction* is defined as a combination of zero or more basic unary predicates. In this chapter we treat the combinations as *conjunctions*. This suffices for most cases where selection restrictions occur. As a consequence, the selection restrictions can be seen as subsets of the lexicon. Therefore we can define the selection restrictions of a *term* as the union of the selection restrictions of the predicates in its body.

The selection restrictions are used to restrain the ways in which predicates and terms can be combined. For example, if an adjectival restriction is added to a term, the selection restriction of the adjective must be as least as specific as the selection restriction of the term. These restraints are formulated by a partial relation on selection restrictions that denotes *at least as specific*. This partial relation is a partial order, which the particular grammar defines. In most cases, and the only cases that we consider, the partial order is imposed on the basic predicates. The relation then follows for arbitrary selection restrictions in the following way. A selection restriction $\langle \mathrm{S} \rangle$ is at least as specific as another selection restriction $\langle \mathrm{R} \rangle$ iff for every basic predicate in the selection restriction $\langle \mathrm{R} \rangle$, the selection restriction $\langle \mathrm{S} \rangle$ contains a basic predicate that is at least as specific. Thus we can for instance say that a predicate or term conforms to the selection restrictions of an argument position if the selection restriction of the predicate or term is at least as specific as the selection restriction of the argument position.

The definitions above make clear what the predicates and terms are. With these definitions in mind we will now formalize the FFRs. A distinction is made between the FFRs that create predicates and FFRs that create terms. Recall that a function is honest if the size of the input is polynomial in the size of the output. A polynomial time computable honest function ensures that the output of the fund formation rules is not totally different from the input: the output is neither much larger, nor much smaller than the input.

7.2.4. DEFINITION. *Fund Formation Rules*
We define the *fund formation rules* for predicates by the following six conditions.

- The FFRs take predicates and terms as input.
- If a predicate serves as input of an FFR, then the FFR specifies the category, the arity, and the functions of the argument positions of the predicate in advance.
  If a term serves as input of an FFR, then the FFR may specify that the term contains or does not contain some term operators.
- The FFRs do not refer to the body, the variables of the argument positions, or the meaning postulate.
- The FFRs may place conditions on the meaning postulate. The FFRs may also place conditions on the selection restrictions of the input. These conditions are computable in polynomial time.
- The category, the arity, and the functions of the argument positions of the predicate that is the output of the FFR are fixed. That is, the category, the arity, and the functions of the argument positions do not depend on the input.
- The input and output are related in the following way.
  - The body of the output is the result of some polynomial time computable, honest function on the bodies and the categories of the input.
  - The argument positions of the output are the result of some polynomial time computable, honest function on the argument positions of the input.
  - The meaning postulate of the output is the result of some polynomial time computable, honest function on the meaning postulate of the input.

We define the *fund formation rules* for terms by the following five conditions.

- The FFRs take predicates and terms as input.
- If a predicate serves as input of an FFR, then the FFR specifies the category, the arity, and the functions of the argument positions of the predicate in advance.
  If a term serves as input of an FFR, then the FFR may specify that the term contains or does not contain some term operators.
- The FFRs do not refer to the body, the variables of the argument positions, or the meaning postulate.
- The FFRs may place conditions on the meaning postulate. The FFRs may also place conditions on the selection restrictions of the input. These conditions are computable in polynomial time.
- The input and output are related in the following way.
  - The body of the output is a 1-open predication, which is the result of some polynomial time computable honest function on the body and categories of the input.
  - The set of term operators of the output is the result of some polynomial time computable honest function on the sets of term operators of the input.
  - The meaning postulate of the output is the result of some polynomial time computable honest function on the meaning postulate of the input.

Let us now compare the formalization given above and the examples presented in Section 7.2.1. After two minor remarks about the formalization of terms and predicates, we consider the formalization of the FFRs.

1. The basic terms in the formalization contain a meaning postulate that was not present in the examples. The main reason to introduce the meaning postulates is to arrive at a more or less uniform representation of basic and derived terms.
2. In the examples the selection restrictions consists of one predicate. In the literature, however, examples are given where the selection restrictions are combinations of unary predicates. These combinations can often be treated as conjunctions. We will not consider cases where selection restrictions cannot be treated as conjunctions, or contain predicates that are not unary.
3. In the first example of an FFR for derived predicates, the functions of the argument positions are not specified. However, from a computational point of view it is more convenient if the functions are specified. Moreover, we conjecture that from a linguistic point of view the FFR is more adequate if the functions are specified.
4. The conditions that are stated in the examples of formation rules seem to be computable by a polynomial time function. Of course, if the conditions are not computable by a polynomial function, then this process is not efficiently computable.
5. The examples of FFRs that are found in the literature all seem to relate the output in a *natural* way to the input. We claim that natural FFRs are nat-

ural to compute, i.e., computable by a polynomial time honest function. It seems that all examples of FFRs described in the literature are computable by polynomial time honest functions.

We will now formulate the fund formation problem for a lexicon L and a set of fund formation rules F.

7.2.5. DEFINITION. *The Fund Formation Problem* (FFP(L, F))
Given a lexicon L and a set of fund formation rules F as defined above.
INSTANCE: A predicate p or a term t.
QUESTION: Is p (or t) derivable from L by F?

7.2.3 The Fund Formation Problem is Undecidable

Although our definition of the fund formation rules may seem to be restrictive, we will show in this section that the FFP is still undecidable. In general, a problem is often undecidable when

1. in the computation of the problem some structure of arbitrary size can be constructed,
2. the input and output of the problem do not have to account for all structures that are constructed in the computation of the problem.

A problem in which these two conditions are fulfilled is often undecidable because during the computation of the problem a structure of arbitrary size, which does not show up in the input or output, can be constructed. Such a structure of arbitrary size can be used, for instance, to simulate the tape of a Turing machine. A further step towards the simulation of the Halting Problem of Turing machines is then often straightforward, and thus the undecidability follows.

In the fund formation process there are two candidate places where structures of arbitrary size can be constructed. The first place is the body of the predicate. The second place is the meaning postulate of the predicate.

We expect that the body is not a good place to construct structures of arbitrary size, for the following reasons. Let us consider the question whether p is a derived predicate. This predicate p is a derived predicate iff p stems from basic predicates and terms in L. Therefore in order to construct a structure of arbitrary size, the fund formation rules need to have rules to build these structures from the basic predicates and terms, and rules to decompose these structures to arrive at predicate p. The decomposition step in the fund formation process seems unusual. Especially, the decomposition of the body of a predicate seems counterintuitive. Therefore the body is probably not a good place to create structures of arbitrary size. However, the decomposition is less peculiar for meaning postulates because we may reduce the arity of a predicate. Hence we expect that the meaning postulate is a good place to construct structures of arbitrary size.

The following theorem proves what we demonstrated above. The theorem shows that no general algorithm exists that for any set of fund formation rules, F, lexicon,

L, and predicate p, or term t, determines whether the predicate, or term, is contained in the fund. that F and L together describe. whether a predicate is derivable from a lexicon and a set of fund formation rules.

7.2.6. THEOREM. *The universal version of the fund formation problem for lexicon L and set of fund formation rules F is undecidable.*

Proof. (Sketch of the proof.)
We simulate the computation of a Turing machine M, which has some nice properties, on input $\$\#$. The properties, however, do not effect the computational capacity of the Turing machine. It is clear from general formal language theory, that there is a reduction from the question whether an arbitrary Turing machine halts on the empty string to the question whether a Turing machine with the nice properties accepts the string $\$\#$.

Now let M be a Turing machine with the following nice properties

- M has one accepting state q_f,
- M accepts only if the tape is empty,
- M has a two-way infinite tape,
- M has begin-tape marker $\$$ and end-tape marker $\#$ for the portion of the tape scanned,
- M has a binary tape alphabet, and
- M moves its head at each computation step.

We simulate the computation of M on input $\$\#$ by choosing an appropriate set of fund formation rules and lexicon. The reader should check that the fund formation rules satisfy the five conditions in the definition of the fund formation rules for predicates.

The set of fund formation rules F that is appropriate is constructed as follows.

- For each instruction $((q, a), (q', b, R))$ in the program of M the set of fund formation rules contains the following rule.

  Input: $\mathrm{pred}_q(x_1)(x_2)_{\mathtt{L}}(x_3)_{\mathtt{R}}$
  Output: $\mathrm{pred}_{q'}(x_1)(x_2)_{\mathtt{L}}(x_3)_{\mathtt{R}}$
  Condition: The meaning postulate of the input describes a state where the `L` argument indicates a stack $a\gamma$ with top a and the `R` argument indicates a stack $c\gamma'$ with top c. The meaning postulate of the output describes a state where the `L` argument indicates a stack $cb\gamma$ with top c and the `R` argument indicates a stack γ'.

- For each instruction $((q, a), (q', b, L))$ in the program of M the set of fund formation rules contains a similar rule.
- For the accepting state q_f the set of fund formation rules contains the following rule.

Input: $\text{pred}_{q_f}(x_1)(x_2)_{\mathtt{L}}(x_3)_{\mathtt{R}}$
Output: $\text{pred}_{q_f}(x_1)$
Condition: The meaning postulate of the input describes a state where the `L` argument indicates an empty stack, and the `R` argument indicates an empty stack.

- For the starting state q_0 the set of fund formation rules contains a similar rule.

Input: $\text{pred}_{q_0}(x_1)$
Output: $\text{pred}_{q_0}(x_1)(x_2)_{\mathtt{L}}(x_3)_{\mathtt{R}}$
Condition: The meaning postulate of the output describes a state where the `L` argument indicates an empty stack, and the `R` argument indicates an empty stack.

The lexicon L contains only one predicate $\text{id}_{q_0}(x_1)$. This predicate corresponds to the immediate description of the Turing machine for the starting state. Let a similar predicate $\text{id}_{q_f}(x_1)$ correspond to the immediate description of the Turing machine for the accepting state.

We now claim that given this set of fund formation rules F and lexicon L, the predicate $\text{id}_{q_f}(x_1)$ is in the fund described by F and L iff the Turing machine M halts on input \$#. □

Restrictions that lead towards decidability. We see two restrictions that would tackle the undecidability of the fund formation problem. Both restrictions are based on the idea to guide a parsing algorithm by means of an abstract measure. This idea originates from Landsbergen (1981) and is later used in the machine translation system Rosetta (Rosetta 1994). The notion of an abstract measure was first introduced in FG by Janssen (1989), where it is called "complexity."

Decidability of the FFP follows if we demand that the FFRs compose predicates and terms from smaller predicates and terms only. The intuition of "smaller" is formalized by the requirement that FFRs should be measure-increasing. To be more precise, the fund formation process is accompagnied with a measure function μ that assigns a measure to every predicate and term. Now the FFRs are measure-increasing when an FFR with input $\phi_1, \ldots, \phi_n$ yields output ψ only if $\mu(\phi_i) < \mu(\psi)$ for all $1 \leq i \leq n$. In other words, the measure of the output must be larger than the measure of each part of the input.

The requirement that the FFRs are measure-increasing implies that the fund formation problem is decidable, for the following reasons. Given a finite lexicon, a finite set of FFRs and a predicte p, whose measure is $\mu(p)$. Because the number of FFRs is finite and the FFRs are measure-increasing, there are only finitely many predicates with measure less or equal to $\mu(p)$ in the fund. Now the following algorithm, which terminates after a finite number of steps, determines whether p is in the fund: "Compute the finite collection of all predicates in the fund with measure less or equal to $\mu(p)$; predicate p is in the fund iff p is in this collection."

FFRs increase the information in meaning postulates. The first restriction that we propose to in order to tackle the undecidability, involves the treatment of meaning postulates. In an informal sense, it seems that the output of a fund formation rule is always more specific than the input. That is, the output contains more information than the input. This information is typically denoted in the meaning postulates of predicates and terms. Therefore, we think that it is admissible to require that the function that computes the meaning postulate of the output is a monotonically increasing function. With this requirement on FFRs, the amount of information expressed by meaning postulates is a measure that ensures decidability. So the first restriction on the FFRs that we propose is that the FFRs must increase the amount of information that a meaning postulate denotes. A disadvantage of the previous restriction is that the notion of meaning postulates has sofar remained unclear in FG. A better restriction is given next.

FFRs never decrease the size of the body. The second restriction that we will present below, is in fact a group of three restrictions. The measure function μ for this second restriction is a little complicated. The measure of a predicate will involve its body and its argument positions. Before we introduce the measure of a predicate, let us examine the sizes of the body and the argument positions more closely.

- The body of a predicate is composed of a number of elementary entities. These elementary entities are strings given in the lexicon, affixes, categories, term operators, and functions. The total number of these elementary entities in an FG-grammar is finite. So the number of the various elementary entities that occur in the body of a predicate seems to indicate the size of the body.
- The size of an argument position depends on the size of its function and its selection restrictions. Because the number of functions is finite, the size of the function is constant. The size of the selection restrictions depends on the number of predicates in the selection restrictions. However, because the selection restrictions are defined as combinations of basic predicates, the number of different selection restrictions is finite. Thus the size of an argument position is constant.

From the observations above, it follows that the size of a predicate mainly depends on the number of elementary entities that occur in its body and the number of argument positions.

Given the group of three restrictions on FFRs to be describe below, the following measure ensures decidability. This measure consists of two parts. The first part is the number of elementary entities in the body of a predicate. This part contributes to the measure unconditionally and positively. That is, an increase in this part always implies an increase in the measure. The second part of the measure is the number of argument positions. This part contributes to the measure in a conditional and negative way. That is, if the first part of the measure does not change and the second part is decreased then the measure is increased.

Now we propose the following group of three restrictions on the FFRs, which yield a decidable fund formation problem:

1. FFRs never decrease the size of the body of a predicate.
2. if an FFR increases the number of argument positions, then the size of the body of the result must also be increased.
3. If the FFRs create two predicates with the same body, the same category, and the same argument positions, then the two predicates are the same predicate.

We will now show that the number of FFRs that can be applied to a predicate without increasing the measure of the predicate is bounded with respect to the size of the FG-grammar and the predicate. The FFRs that do not increase the measure of a predicate are FFRs that change the category of the predicate or the function or selection restrictions of the predicate's argument positions. Now because the number of functions and different selection restrictions are all bounded with respect to the size of the FG-grammar, the number of different forms of an argument position is bounded, with respect to the size of the FG-grammar. Because the set of categories in an FG-grammar is finite, the number of categories is bounded with respect to the size of the FG-grammar. Obviously, the number of argument positions of a predicate is bounded with respect to the size of the predicate. And therefore the number of different argument positions of a predicate is bounded with respect to the sizes of the FG-grammar and the predicate. Hence the number of FFRs that can be applied to a predicate without increasing the measure of the predicate is bounded with respect to the size of the FG-grammar and the predicate. Thus for any predicate we know that after a bounded sequence of FFRs the measure of the predicate must be increased. So given a predicate p with measure $\mu(p)$, we can compute the bounded collection of all predicates in the fund with measure less or equal to $\mu(p)$ and check whether the predicate p is in this collection, thus in the fund.

Although the measure is slightly more complicated than the first measure we introduced, the restriction on the FFRs seems less severe. Given the measure, all the examples of FFRs that we gave in Section 7.2.1 conform to the three restrictions.

7.3 Clause Structure Formation Process

A sketch of the situation. We presuppose a fund that contains all predicates and terms. In this process, predicates and terms from this fund, and operators and functions are combined into clause structures. These clause structures serve as the underlying representation of sentences.

7.3.1 An Informal Introduction

In the clause structure formation process, a predicate from the fund is turned into an underlying representation for a sentence. Given that certain conditions, examples of which are described below, are satisfied, a clause structure is formed by

- filling in the argument positions of the predicate,

- extending the predicate with satellites,
- adding operators to the predicate, and
- adding functions to the arguments of the predicate.

The argument positions are filled with predicates and terms from the fund. The satellites are optional arguments of the predicate, like prepositions. The operators typically express tense (past, present, future) and illocution (declarative, interrogative, imperative). In this chapter we have little to say about the syntactic and pragmatic functions that are added in this process.

As examples we consider first three correct sentences, and then four incorrect sentences. The first correct sentence, "John kisses Mary," may be represented by the clause structure

$$\mathtt{Decl}\,\mathtt{Pres}\,\mathrm{kiss}_V(\mathtt{d1}x_1 : \mathrm{john}(x_1))_{\mathtt{AgSubj}}(\mathtt{d1}x_2 : \mathrm{mary}(x_2))_{\mathtt{GoObj}}. \tag{7.1}$$

Both the argument positions are filled with basic terms. No use is made of the possibility to extend the predicate with a satellite. The predicate is extended with the operators `Decl` for declarative, and `Pres` for present tense. The syntactic functions subject (`Subj`) and object (`Obj`) are added to the arguments.

A simplified clause structure for the second correct sentence, "John kisses Mary in the garden," is an extension, of the previous clause structure. The clause structure that is given in (7.1). is extended by the satellite $(\mathtt{d1}x_3 : \mathrm{garden}(x_3))_{\mathtt{Loc}}$, which denotes a location:

$$\mathtt{Decl}\,\mathtt{Pres}\,\mathrm{kiss}_V(\mathrm{john})_{\mathtt{AgSubj}}(\mathrm{mary})_{\mathtt{GoObj}}(\mathtt{d1}x_3 : \mathrm{garden}(x_3))_{\mathtt{Loc}}.$$

A simplified clause structure for the third correct sentence, "John will kiss Mary because John loves Mary," may be represented by the following clause structure. The operator `Fut` stands for future tense, the function `Rsn` expresses that the satellite denotes a reason.

$$\mathtt{Decl}\,\mathtt{Fut}\,\mathrm{kiss}_V(\mathrm{john})_{\mathtt{AgSubj}}(\mathrm{mary})_{\mathtt{GoObj}}\ (\mathtt{Pres}\,\mathrm{love}_V(\mathrm{john})_{\mathtt{Subj}}(\mathrm{mary})_{\mathtt{Obj}})_{\mathtt{Rsn}}$$

Now let us consider the following four incorrect sentences, and explain their ungrammaticality.

1. * The book kisses the sky.
2. * John is old in the garden.
3. * John kissed Mary tomorrow.
4. * John kisses Mary, so that Mary loved John.

These incorrect sentences show that satellites, operators and arguments may only be added under certain conditions. For instance we mentioned in Section 7.2.1 that arguments may fill the argument positions only if the selection restrictions of the argument positions are satisfied by the arguments. The first ungrammatical sentence shows two arguments, "The book" and "the sky," that violate the selection restrictions imposed by the verb "to kiss." The second sentence is ungrammatical if we assume that the satellite "in the garden" can only be added to a predicate that indicates an action. The third sentence is ungrammatical, because the satellite "tomorrow" requires future tense. The fourth ungrammatical sentence contains a satellite that denotes a result. Such a satellite must describe an event that occurs later in time than the event that caused the result. In this sentence, however, the satellite "so that Mary loved John" denotes an event that occurs prior to the event denoted by "John kisses Mary."

7.3.2 The Formalization

Above we presented some examples that make clear what the clause structure formation process does. Now we will formalize the clause structure formation process. First we will discuss the input and output of the clause structure formation process and formalize the notion of a *clause structure*. Then we formalize the *clause structure formation rules*. Next we compare the formalization and the examples. We end with a formulation of the *clause structure formation problem*.

Given some set of clause structure rules, the clause structure formation process constructs a clause structure. The next process, the form specification process, must express any clause structure resulting from this process as some string. So in a certain sense, the clause structure formation process determines the grammaticality of strings.

The input of the clause structure formation process consists of *terms* and *predicates* from the fund, *clause-structure operators*, and *semantic functions*. In Section 7.2 we have formalized the notions of terms and predicates. The other entities that are used in the clause structure formation are the clause-structure operators and semantic functions. Both kinds of entities are taken from fixed finite sets, which are specified by the particular FG-grammar.

The output of the clause structure formation process is a clause structure. We define the clause structures as follows.

7.3.1. DEFINITION. *Clause Structures*
In the structure of *clause structures* three components can be distinguished.

- A set of *clause-structure operators* from Ω.
- A set of *satellites*, which are pairs that consist of a *semantic* function from F, and either a term or a clause structure.
- A *predication*, which is a predicate whose argument positions are filled with terms and clause structures. Optionally, *syntactic* functions from F are assigned to argument positions.

Four successive steps may be distinguished in the clause structure formation. See Bakker (1994, §8.2) and Kwee (1994, §3.5.2) for some worked-out examples of the clause structure formation.

1. In the first step of the clause structure formation terms and clause structures are inserted in the argument positions of a predicate.

2. When all the argument positions of the predicate are filled, syntactic and pragmatic functions are assigned to some of the argument positions. The resulting structure is a *predication.*
3. The third step starts with extending the predication with an empty set of operators and an empty set of satellites. Successively, this initial clause structure is extended with satellites and operators. The clause structure that results from this third step is called a *candidate clause structure.*
4. The fourth and final step of the clause structure formation checks whether the candidate clause structure conforms to the imposed *clause structure constraints.* Obviously, the clause structure formation process outputs only candidate clause structures that conform to the clause structure constraints.

Our main attention goes to the clause structure formation rules that are used in the first and fourth step. In the first step two kinds of restrictions play a role. The one kind of restrictions we have seen before. In Section 7.2.1, we mentioned that the selection restrictions given in the argument positions of a predicate restrict the kind of arguments that the predicate takes. Although this conception of the selection restrictions is a slight misrepresentation of the actual situation in FG, we hold on to this idea because no real alternative is provided. According to the theory of FG (Dik 1989), a violation of the selection restrictions does not block the further generation of the clause structure, but provokes a special interpretation of the clause structure. However, FG does not dilate upon this special interpretation whatsoever. The other kind of restriction on the arguments is the sort of variable in the argument position. In Section 7.2.2 we introduced two types of variables: variables that refer to individuals and variables that refer to events. Obviously an argument position that is specified for a variable that refers to one type of variable may not be filled with an argument of the other type.

Now we claim that the clause structure formation rules that play a role in the first step of the clause structure formation process hold for all FG-grammars. Thus, the first step is defined in the following way.

7.3.2. Definition. *Clause Structures Insertion Rules*
In a predicate p it is allowed to *insert*

- a clause structure C for a variable e_i in p iff the variable e_i refers to an event, or
- a term t for a variable x_i in p iff the variable x_i refers to an individual and the selection restrictions of the term t are at least as specific as the selection restrictions of the i-th argument position of p.

In contrast to the previous rules, the rules that play a role in the fourth step depend on the particular FG-grammar that is considered. The rules in the fourth step are constraints on the well-formedness of a clause structure. As for these clause structure constraints, no formalism whatsoever has been put forward by FG. We propose that clause structure constraints are stated in terms of the properties of a clause structure. To be more precise, we define

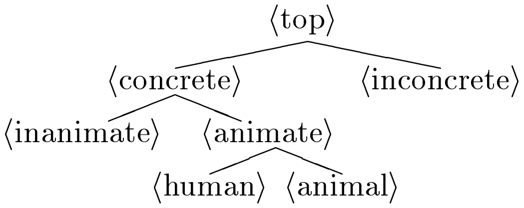


Figure 7.2: A possible partial order on selection restrictions.

7.3.3. DEFINITION. *Clause Structure Constraints*
A *clause structure constraint* is a boolean combination of properties of clause structures, and properties of satellites.

Now we define the properties of a satellite, and a clause structure as follows.

- The only *properties of a satellite* are the semantic function of the satellite, and either the term operators of the satellite's term, or the clause-structure operators of the satellite's clause structure.
- The only *properties of a clause structure* are its clause-structure operators, the functions of its satellites, and the semantic functions of the arguments of its predication.

Let us now compare the formalization given above and the examples presented in Section 7.3.1. We will first consider the four incorrect sentences and their ungrammaticality. Then we consider briefly the three correct sentences.

- In the first example of an incorrect sentence, "The book kisses the sky," the arguments do not satisfy the selection restrictions of their argument positions. We may safely assume that the selection restrictions of "The book" and "the sky" are ⟨inanimate⟩ and ⟨inconcrete⟩, respectively. The selection restrictions of the argument positions of the verb "to kiss" are ⟨animate⟩, see Section 7.2.1. Let the partial order on selection restrictions be as given in Figure 7.2. The least specific selection restriction is ⟨top⟩. In this (incomplete) hierarchy neither ⟨inanimate⟩, nor ⟨inconcrete⟩ is more specific than ⟨animate⟩.
- In the second example of an incorrect sentence the adjective "old" is used as a predicate. A candidate clause structure for this sentence might contain the following set of operators, predication, and set of satellites:

  $$\{\texttt{Decl}, \texttt{Pres}\};\ \mathrm{old}_A(\texttt{d1}x : \mathrm{john}(x))_{\texttt{Zero}};\ \{(\texttt{d1}y : \mathrm{garden}(y))_{\texttt{Loc}}\}.$$

  The semantic function `Zero` denotes that the argument of the predication does not have a semantic function.

  We stated that the second example of an incorrect sentence is ungrammatical because the satellite "in the garden" can only be added to a predicate that indicates an action. A predicate indicates an action if it contains an argument position that bears the semantic function `Ag`. Thus the clause structure constraint can be stated as: "If a clause structure has the property `Loc`, then the clause structure must also have the property `Ag`." Clearly, the candidate clause structure for this sentence does not have this property `Ag`. So the candidate clause structure does not satisfy the constraint.

- The third example of an incorrect sentence contains the satellite "tomorrow," which denotes time. In the literature this satellite is treated in different ways. On the one hand, Dik (1992, p. 50) treats "tomorrow" as a kind of basic satellite. On the other hand, Dik (1989, p. 181) treats entities similar to "tomorrow" as terms. The observation that we want to make however, does not depend on the actual representation of "tomorrow."
  We stated that this sentence is ungrammatical because the satellite "tomorrow" may only be added if the predication describes an event in the future. However, it does not seem to be in the spirit of FG to have a constraint stating that the specific satellite "tomorrow" can only extend clause structures that contain clause-structure operator `Fut`. Therefore we assume that the satellite "tomorrow" contains some property "future." The ungrammaticality of this sentence would then be described by a violation of the following general constraint: "If a clause structure contains a satellite with semantic function `Time` and this satellite contains property "future," then the clause structure contains clause-structure operator `Fut`."
- The fourth example of an incorrect sentence contains the satellite of reason "so that Mary loved John," which is a clause structure. A schematic candidate clause structure for the sentence contains the following set of operators, predication, and set of satellites:

$$\{\texttt{Decl},\texttt{Pres}\};\ \text{kiss}_V(\text{john})(\text{mary});\ \{(\{\texttt{Past}\};\ \text{love}_V(\text{mary})(\text{john});\ \emptyset)_{\texttt{Rsn}}\}.$$

  We stated that the fourth example of an incorrect sentence is ungrammatical because a satellite that denotes a result must describe an event that occurs later in time than the event that caused the result. This clause structure constraint can be stated as: "If a clause structure has the properties `Rsn` and `Pres`, then the satellite with property `Rsn` has property `Pres`, or `Fut`." Clearly, this constraint is violated by the candidate clause structure because the satellite has property `Past`.
- In each of the three examples of correct sentences the argument positions of the predicate kiss_V are filled with the arguments "john" and "mary." We presuppose that the selection restrictions of both arguments are $\langle$human$\rangle$. Thus given the partial order in Figure 7.2, the selection restrictions of both arguments are more specific than the selection restrictions of the argument positions of the verb "to kiss." So the first step of the clause structure formation process is possible for each of the three correct sentences. Hence the first example, "John kisses Mary," satisfies the clause structure formation rules.
- The second example, "John kisses Mary in the garden," shows a clause structure with a satellite that denotes a location. The set of the properties of

the satellite is $\{\mathtt{d}, \mathtt{1}, \mathtt{Loc}\}$. The set of the properties of the clause structure is $\{\mathtt{Decl}, \mathtt{Pres}, \mathtt{AgSubj}, \mathtt{GoObj}, \mathtt{Loc}\}$. Clearly, the constraint that if a clause structure has the property `Loc`, then the clause structure must also have the property `Ag`, is satisfied.

- In the third example, "John will kiss Mary because John loves Mary," the satellite, which consists of a clause structure, denotes a reason. The set of the properties of the satellite is $\{\mathtt{Pres}, \mathtt{Rsn}\}$. The set of the properties of the main clause structure is $\{\mathtt{Decl}, \mathtt{Fut}, \mathtt{AgSubj}, \mathtt{GoObj}, \mathtt{Rsn}\}$. Clearly, the constraint that if a clause structure has the properties `Rsn` and `Pres`, then the satellite with property `Rsn` has property `Pres` or `Fut`, is satisfied.

We have illustrated above that the formalizations of clause structures and clause structure rules seems acceptable. Now we have come to the point where we will formulate the clause structure formation process as a decision problem: the clause structure formation problem.

In the clause structure formation process predicates and term from the fund are turned into clause structures. So the clause structure formation problem can be seen as the following problem: "Given some predicates and terms and a set of clause structure rules; is some clause structure C formed by combining these predicates and terms according to these rules?"

A small complication in this formulation of the clause structure formation problem is that the problem has to solve the fund formation problem from Definition 7.2.5. As we saw in the previous section it is not at all trivial to determine whether predicates or terms are in the fund. So if we try to analyze the complexity of the clause structure formation problem, we should be careful not to include the complexity of the fund formation problem in our analyses. We avoid this inclusion when we assume that there is a easy way to determine whether terms and predicates are in the fund. A common technique to solve difficult questions in an easy way is to introduce an oracle that answers the question. In this case we say that the oracle A answers the fund formation problem from Definition 7.2.5.

We pointed out in Section 7.2 that the notion of meaning postulates is not worked-out in FG. So we ignore the existence of meaning postulates completely in the clause structure formation problem. Thus the oracle A answers the questions for the fund formation problem regardless of meaning postulates. Now we define the clause structure formation problem as follows.

7.3.4. DEFINITION. *The Clause Structure Formation Problem* $(\mathrm{CSP}(L, F, R))$
Given a lexicon L, a set of fund formation rules F, an oracle A for "$p,\ t \in \mathrm{FFP}(L, F)$?," and a set of clause structure formation rules R.
INSTANCE: A clause structure C.
QUESTION: Is C formed by R from predicates and terms in $\mathrm{FFP}(L, F)$?

7.3.3 The Complexity of the Clause Structure Formation Problem

In this section we consider the complexity of the clause structure formation problem. First we will show that given Definition 7.3.4 CSP is an *NP*-problem. We will prove that CSP is in *NP* by means of an *NP*-algorithm. This algorithm will show that the nondeterministic character of CSP is in fact caused by the limited definition of the notion selection restriction. After this *NP*-algorithm we will propose two alterations which both lead to a tractable CSP.

CSP(L, F, R) is in *NP*. We will prove that CSP(L, F, R) is in *NP*, by the following sketch of an nondeterministic polynomial time algorithm. Given a lexicon L, a set of fund formation rules F, an oracle A for "p, $t \in$ FFP(L, F)?," a set of clause structure formation rules R, and a clause structure C.

7.3.5. LEMMA. *Given a clause structure C, the elementary components of C and their properties are computable in polynomial time, with respect to the size of C.*

Proof. We mean by elementary components predicates, terms, clause-structure operators, predications, satellites, etcetera. By definition, the clause-structure operators, predication and set of satellites in a clause structure is easily distinguished. We can also directly determine the functions, predicate, terms and clause structures from which the predication and satellites are built. Thus we can determine which predicates, terms, clause structures, functions, and clause-structure operators build up the clause structure C. Given all the elementary components of C, we can determine their properties. From Definition 7.3.3 it follows that all properties are locally specified. Hence the elementary components and their properties can be computed in polynomial time, with respect to the size of the clause structure C. □

One of the problems that CSP has to solve is whether a clause structure satisfies a set of clause structure constraints.

7.3.6. LEMMA. *Given a clause structure C, and a set of clause structure constraints R'. It can be computed in polynomial time, with respect to the size of R' and the size of C, whether C satisfies all clause structure constraints.*

Proof. Clause structure constraints are boolean combinations of properties of satellites and clause structures. Lemma 7.3.5 shows that these properties are computed in polynomial time from a given clause structure. Given these properties the conditions are checked by evaluating them. It takes on amount of steps that is linear in the size of the boolean formula in order to evaluate a boolean formula for some given values. Hence we can check the clause structure constraints in an amount of time that is linear in the size of the set of clause structure constraints. □

All that remains is to determine whether the predicates and terms that built the clause structure come from the fund and whether these predicates and terms satisfy the clause structure insertion rules. Because the clause structure contains no clues to determine the selection restrictions of terms and predicates, we compute these selection restrictions nondeterministically.

7.3.7. LEMMA. *Given a clause structure C and oracle A for "$p, t \in FFP(L, F)$." We can check in nondeterministic polynomial time with respect to the size of C and L whether the predicates and terms in C are in the fund and whether the arguments conform to the selection restrictions of the argument positions.*

Proof. Lemma 7.3.5 shows that, deterministically, in polynomial time the elementary components of C and their properties are established. Because selection restrictions are subsets of the lexicon L, we can, nondeterministically, compute each selection restriction in polynomial time, with respect to the size of L. A linear amount of questions, with respect to the size of C, to oracle A suffices to determine whether all predicates and terms are in the fund. Given the selection restrictions of each term and predicate, the partial relation "at least as specific" provides the answer whether arguments conform to the selection restrictions of the argument positions. □

7.3.8. THEOREM. *The universal version of the clause structure formation problem is computable in nondeterministic polynomial time.*

Proof. By Lemmas 7.3.6 and 7.3.7. □

Towards a tractable clause structure formation problem. Let us reconsider the clause structure formation problem. Lemma 7.3.7 shows that the most difficult part of the clause structure formation problem is the first step in the clause structure formation. In the first step, terms and clause structures are inserted in argument positions. The insertion of these arguments in argument positions is partially managed by selection restrictions. This step is difficult because the meaning of a clause structure is involved, whereas FG has not taken meaning into account, yet.

Now we will explain why the insertion-step is difficult for the clause structure formation problem. The reader should realize that the insertion-step boils down to the following problem in the clause structure formation problem: "Given some predication, determine whether the arguments conform to the selection restrictions of the argument positions." The difficulty now lies in the fact that selection restrictions are not easily available. The only place in which the selection restrictions are found is the fund. Furthermore, information from the fund can only be obtained in a constraint way. That is, we can only ask the oracle A whether some predicate or term is in the fund. Moreover, each component of the questioned predicate or term, except for the meaning postulate, has to be specified.

This constraint way implies that in the worst case, e.g., when the questioned predicate or term is not in the fund, only after exponentially many questions to the oracle A the selection restriction of a predicate or term is known. Clearly, this is not an efficient way to compute the selection restriction for a given predicate or term. What is more, this inefficiency is actually due to FFP and not, in essence, part of CSP.

We will now propose two alterations that lead to a tractable clause structure formation problem. Further research should indicate whether the proposals are acceptable.

Include selection restrictions in grammar descriptions. In Section 7.2.2 we defined the form and use of selection restrictions. We confined ourselves to a general description because it is not apparent from the literature what the precise nature is of selection restrictions. We conjecture that a thorough discussion of the selection restrictions and their use will demand that the selection restrictions are included in the description of FG-grammars.

Our first proposal now is to add the set of all possible selection restrictions that may play a role in an FG-grammar to the description of that FG-grammar. This proposal yields that the number of selection restrictions is linear with respect to the size of the FG-grammar. Hence Lemma 7.3.7 need not compute the right selection restrictions of predicates and terms nondeterministically. Instead Lemma 7.3.7 can simply try all possible selection restrictions and thus compute whether predicates and terms are in the fund and whether arguments conform to the selection restrictions of the argument positions. Because the number of selection restrictions is bounded with respect to the size of the grammar, this computation is bounded polynomially with respect to the sizes of the clause structure and the grammar.

Thus we claim that this first proposal not only facilitates a thorough discussion of selection restrictions, but also yields a tractable clause structure formation problem.

Link selection restrictions to semantic functions. In this second proposal two extra assumptions make the CSP tractable. First, we assume that there is an oracle A', which is less stringent than the oracle A. Second, we assume a connection between selection restrictions and semantic functions. Further research should indicate whether this connection is acceptable on linguistic grounds.

First, consider an oracle A' that is similar to A, but less stringent. The oracle A' answers "yes" iff the fund contains a predicate or term that is an equally or more specific version of the questioned predicate or term. We mean by an equally or more specific version that the two predicates only differ in selection restrictions, where the selection restrictions of the predicate in the fund are at least as specific as the selection restrictions of the questioned predicate.

For instance, suppose that the fund contains the predicate "$\mathrm{john}_N(x : \langle\mathrm{male}\rangle(x))$" and the selection restriction $\langle$male$\rangle$ is more specific than the selection restriction $\langle$human$\rangle$. The oracle A will answer "No" to the question whether the predicate "$\mathrm{john}_N(x : \langle\mathrm{human}\rangle(x))$" is in the fund. However, the less stringent oracle A' will answer "Yes" to the same question.

Second, Bakker (1994) argues that the semantic function of an argument position determines to some extent the selection restriction of the argument-position. For instance, the semantic function that indicates an agent is restricted to animate beings. We propose to enforce Bakker's (1994) argumentation further and assume that each semantic function corresponds to at most one selection restriction. Hence we restrict the FG-grammars by demanding that a very strong dependence hold between semantic functions and selection restrictions. Further research should indicate whether this demand is acceptable on linguistic grounds.

Now we will show that the insertion-step is easy to compute if the restriction on FG-grammars and the assumption of a less stringent oracle A' hold. We show how the insertion-step is computed in case of a simple predication. In the case of arbitrary predications a generalization suffices.

Given a predication that consists of a predicate with n terms. The semantic functions of the argument-position are given by the predication. From these semantic functions we can compute the selection restrictions of the argument-positions of the predicate. Now we consult the oracle A' to determine whether the predicate with the appropriate selection restrictions is in the fund. The oracle answers "yes" iff the predicate with exactly these n selection restrictions appears in the fund. (If the fund would contain a more specific version of the predicate then some semantic function would correspond to more than one selection restriction.) Now we consult the oracle whether the terms with the appropriate selection restrictions are in the fund. An affirmative answer of the oracle then proves that the questioned term conforms to the selection restriction imposed by the argument-position.

7.4 Form Specification Process

A sketch of the situation. The form specification process determines the *form* that the predicates and terms of a clause structure will take in the sentence. The words "inherit" the properties of the predicates and terms, for instance, category and operators. These properties are necessary for the left-to-right ordering of the words in the order specification process (see Section 7.5).

7.4.1 An Informal Introduction

In the form specification process terms and predicates are expressed as words. Form specification rules determine the lexical form of terms and predicates, based on the presence of, for instance, operators and functions in the clause structure in which these terms and predicates appear. As examples, consider the following typical form specification rules.

- Let the following clause structure be the underlying representation of the sentence "John kisses the girl"

$$\mathtt{Decl}\ \mathtt{Pres}\, \mathrm{kiss}_V(\mathtt{d1}x : \mathrm{john}(x))_{\mathtt{AgSubj}}(\mathtt{d1}y : \mathrm{girl}_N(y))_{\mathtt{GoObj}}$$

  The basic term for the proper name "John" is simply expressed as the string in the term's body. The derived term $(\mathtt{d1}y : \mathrm{girl}_N(y))$ is expressed as the two words "the" and "girl." The verbal predicate kiss_V is expressed as the word "kisses," because the subject is a third person singular agent and the clause structure represents present tense.

- The verbal predicate kiss_V within the simplified clause structure for the sentence "John is kissed by Mary"

$$\texttt{Decl Pres}\,\text{kiss}_V(\text{john})_{\texttt{AgObj}}(\text{mary})_{\texttt{GoSubj}}$$

  is expressed as the two words "is" and "kissed," because the subject is a third person singular goal and the clause structure represents present tense.
- The anaphoric terms ($\texttt{A}x_1$) and ($\texttt{A}x_2$) in the next simplified clause structure for the sentence "John believes that Mary told the police that she will kiss him" are expresses as "him" and "she," respectively,

$$\texttt{Decl Pres}\ \text{believe}_V(\texttt{d1}x_1 : \text{john}(x_1))(\texttt{Past}\ \text{tell}_V(\texttt{d1}x_2 : \text{mary}(x_2))(\text{the police}) \\ (\texttt{Fut}\ \text{kiss}_V(\texttt{A}x_2)_{\texttt{Subj}}(\texttt{A}x_1)_{\texttt{Obj}})\ )$$

  because the subject x_2 is female, and the object x_1 is male.

7.4.2 The Formalization

Above we presented some examples that make clear what the typical form specification rules can do. Now we will formalize the form specification process. The input of the form specification process are clause structures, which are defined in Section 7.3.2. So we start this section with a thorough discussion of the *output* of the form specification process. We then, gradually, move on to a formalization of the *form specification rules.* Next we compare the formalizations and the examples. We end with a formulation of the form specification problem.

According to Dik (1989), the form and order specification processes are distinct processes and the order is determined after the form. However, the precise nature of the form and order specification process is unclear. Because the form and the order specification processes both seem to refer to properties of predicates and terms in a clause structure, it is argued that the form and order specification processes should be combined (cf., Bakker 1994, Kwee 1994). This later idea also does justice to the conception that the clause structure is the final underlying representation of a sentence. Although we believe that this latter approach is interesting, we hold on to Dik's (1989) original idea that separates the form and order specification processes.

The form specification process determines the surface forms that the various elements in a clause structure will take in the represented sentence. The form specification rules express *predicates* and *terms* as set of words. *Operators* and *functions* are optionally expressed as separate words or affixes. A *clause structure* is expressed when all the predicates and term in the clause structure are expressed. Thus in a simplified view a clause structure would be expressed as a set of strings. In a more accurate view, however, we would see that a clause structure is expressed as some recursive structure. The recursive elements in such a structure correspond roughly to what are called constituents.

The predicate and arguments of a *predication*, and the *satellites* are each expressed as a set within the set that is the expression of the clause structure in which they occur.

The description that we gave of the output of the form specification process up to now, is clearly insufficient. The next process, the order specification process, can never purely on the basis of the form of words determine in which way these words must be ordered. The information that is needed in the forms specification process in order to settle the surface form of a predicate or term, is also needed in the order specification process to order these words linearly. For instance, a word "can" is either a verb or a noun, a word "John" may be a subject or equally well an object, and from a set of words declarative and interrogative sentences can be formed.

In order to meet the requirements of the order specification process, we assume that the output of the forms specification process consists of recursive structures of which the basic elements are pairs of a word and its properties. This output is called a *template-filler*," and defined as follows.

7.4.1. DEFINITION. *Template-fillers*
Presuppose a collection of properties that play a role in the form and order specification processes.

1. If w is a word and Q is a set of properties, then the pair (w, Q) is a template-filler. These template-fillers are the basic template-fillers.
2. If $\tau_1, \ldots, \tau_n$ are template-fillers, then the set $\{\tau_1, \ldots, \tau_n\}$ is a template-filler.
3. Nothing else is a template-filler.

The properties that play a role in both the form specification process and the order specification process stem from the predicates and terms in a clause structure. We define these properties as follows

- The only *properties of a predicate* in a clause structure are
  - the category of the predicate,
  - the function of the predicate as a satellite or argument,
  - the functions of the arguments of the predicate,
  - the clause-structure operators that are applied to the predicate.
- The only *properties of a term* in a clause structure are
  - the variable of the term,
  - the function that the term has as a satellite or argument,
  - the term operators that are applied to the term.

Now that the template-fillers and properties are defined, we will formalize the form specification rules. Three types of form specification rules are distinguished: *ordinary*, *contextual* and *auxiliary* form specification rules. The first two types are very similar; the third type is rather exceptional.

The ordinary form specification rule maps a predicate (or term) and its set of properties onto a template-filler which consists only of one or more basic template-fillers. The set of properties of each basic template-filler is assumed to be equal to the set of properties of the input predicate (or term).

The contextual form specification rule also maps a predicate (or term) and its set of properties onto a template-filler which consists only of, one or more, basic template-fillers. The set of properties of each basic template-filler is assumed to be equal to the set of properties of the input predicate (or term). In addition to the ordinary form specification rule, the contextual form specification rule presupposes that the clause structure also contains some other predicates and sets of properties.

The auxiliary form specification rule is exceptional, because it does not map a predicate (or term) and its set of properties onto a template-filler. Instead, a predicate (or term) and its set of properties is mapped onto a number of predicates and terms and their sets of properties. We require that the output of the auxiliary form specification rule extends the input. We say that the output extends the input if the input occurs in the output and either the set of properties of the input is increased, or at least one new predicate (or term) and its properties occurs in the output. An ample discussion by Dik (1989) is devoted to the kinds of auxiliary properties that an auxiliary rule introduces. This results in the observation that auxiliary rules introduce operators that do not appear in the clause structure. These operators are only used to trigger other expression rules. We will call these auxiliary operators *trigger operators*. These trigger operators are auxiliary properties that only exist during the form specification process. We will assume that only the input predicate or term of an auxiliary form specification rule may be extended with trigger operators. Eventually every predicate or term in the output of an auxiliary form specification rule must be expressed by an ordinary or contextual form specification rule. Furthermore, the auxiliary form specification rules are sensitive to the fact whether properties have served to satisfy the conditions of other auxiliary form specification rules. In this way we avoid that an auxiliary form specification rule is applied twice to a predicate, and that mutually exclusive auxiliary form specification rules are applied to the same predicate one after the other.

Now let us define these three types of form specification rules.

7.4.2. DEFINITION. *Ordinary Form Specification Rules*
We define an *ordinary form specification rule* for a predicate as follows.

- The form specification rule specifies a set of properties Q' that the input must contain, if the rule is applicable.
- The form specification rule specifies a polynomial time computable, honest function f, which computes from the body of the input the set of words as which the predicate is expressed.

So on input predicate P with set of properties Q, the ordinary form specification rule computes the template-filler $\{(w_1, Q), \ldots, (w_n, Q)\}$, provided that $Q \supseteq Q'$ and $f(b) = \{w_1, \ldots, w_n\}$ where b is the body of P.

The contextual form specification rules determine the string of the predicate based on the predicate's properties and the properties of other predicates in the clause structure.

7.4.3. DEFINITION. *Contextual Form Specification Rules*
We define a *contextual form specification rule* for a predicate as follows.

- The form specification rule specifies a set of properties Q'_0 that the input must contain, if the rule is applicable.
- The form specification rule specifies one or more sets of properties $\{Q'_1, \ldots, Q'_m\}$, that other predicates or terms in the clause structure must contain, if the rule is applicable.
- The form specification rule specifies a polynomial time computable, honest function f, which computes from the body of the input the set of words as which the predicate is expressed.

So given a clause structure C and predicate P with set of properties Q, the contextual form specification rule computes the template-filler $\{(w_1, Q), \ldots, (w_n, Q)\}$, provided that $Q \supseteq Q'_0$ and $f(b) = \{w_1, \ldots, w_n\}$ where b is the body of P, and C contains predicates and terms with properties $Q_1 \ldots Q_m$ and $Q_1 \supseteq Q'_1, \ldots, Q_m \supseteq Q'_m$.

The auxiliary form specification rules introduce new (auxiliary) properties and predicates.

7.4.4. DEFINITION. *Auxiliary Form Specification Rules*
We define an *auxiliary form specification rule* for a predicate as follows.

- The form specification rule specifies a set of properties Q'_0. Only if the input contains these properties and these properties have not been "used" before, the rule is applicable. After the rule is applied the set of properties Q'_0 in the input are "used."
- Optionally, the form specification rule specifies a set of properties Q'_1 that are added to the properties of the input, and a set of trigger operators Q_t.
- Optionally, the form specification rule specifies a set of auxiliary predicates and terms $P'_2 \ldots P'_m$, and sets of properties for this auxiliary predicate $Q'_2 \ldots Q'_m$.

Let predicate P with set of properties Q be the input of an auxiliary form specification rule. We can extract from the set Q the subset of all used properties Q_u. Assume that the set Q contains all the properties denoted in Q'_0. Furthermore, let the sets Q_u and Q'_0 be disjunct. Then and only then this auxiliary form specification rule yields the predicate P with set of properties $Q \cup Q'_1$, and the predicates and terms $P'_2 \ldots P'_m$, with sets of properties $Q'_2 \ldots Q'_m$, respectively. The set of used properties in $Q \cup Q'_1$ is $Q_u \cup Q'_0$.

Let us now compare the formalization given above and the three examples presented in Section 7.4.1. The terms "John" and "the girl" in the first example of Section 7.4.1 may be expressed by ordinary form specification rules. The surface form of the verbal predicate may be the result of a contextual form specification rule, which specifies that

- the input should have the properties V and `AgSubj`,
- the term with function `Subj` is third person singular, and
- the predicate is expressed as the body of the input plus the suffix "es."

The surface form of the verbal predicate in the second example may be the result of a series of three form specification rules. First, an auxiliary form specification rule may have specified that

- the input should have the properties V, `GoSubj`, `Decl`, and `Pres`,
- the operator `PaP`, which stands for past particle, has to be added to the properties of the input, and
- the auxiliary predicate be_{Aux} with the properties `GoSubj`, `Decl`, and `Pres` is introduced.

Second, the predicate kiss_V with property `PaP` may be expressed as "kissed" by an ordinary form specification rule, which specifies that

- the input should have the property `PaP`,
- the predicate is expressed as the body of the input plus the suffix "ed."

Third, the auxiliary predicate be_{Aux} may be expressed as "is" by a contextual form specification rule, which specifies that

- the input should have the properties Aux and `GoSubj`,
- the term with function `Subj` is third person singular, and
- the predicate is expressed as the irregular form "is."

The last example in Section 7.4.1 shows the effect of a contextual form specification rule for terms. This form specification rule has to search the whole clause structure in order to find that x_2 is female, and $(\mathtt{A}x_2)_{\mathtt{Subj}}$ must be expressed as "she." This example may be the result of a contextual form specification rule for terms, which specifies that

- the input should have the properties `A` and `Subj`,
- the clause structure contains a term, which has the property female, with the same variable as the input, and
- the anaphoric term is expressed as the third person, singular, female pronoun "she."

In order to formulate the form specification problem, we introduce an oracle B that answers the clause structure formation problem from Definition 7.3.4.

7.4.5. DEFINITION. *The Form Specification Problem* $(\text{FSP}(L, F, R, S_f))$
Given a lexicon L, a set of fund formation rules F, a set of clause structure formation rules R, an oracle B for "$C \in \text{CSP}(L, F, R)$?," and a set of form specification rules S_f.
INSTANCE: A template-filler x, i.e., a sequence of strings with their properties.
QUESTION: Is there a clause structure $C \in \text{CSP}(L, F, R)$ such that C is expressed as x by S_f?

7.4.3 The Form Specification Problem is *NP*-complete

In this section we will prove that the universal version of the form specification problem is *NP*-complete. First, we will present a reduction from the *NP*-complete problem 3-SATISFIABILITY (3SAT) to the universal version of FSP(L, F, R, S_f). In Lemma 7.4.6 we will prove that this reduction is computable in polynomial time, and preserves answers. Second, we will show in Theorem 7.4.8 that the universal version of FSP(L, F, R, S_f) is in fact NP-complete.

We conjecture that FSP(L, F, R, S_f) is NP-hard, because NP-hardness is likely if a, locally made, choice in a computation has a global effect on a computation. The contextual operator seems to have such a global effect. To make this clear: Suppose that the template-filler x contains multiple occurrences of the string y. Let s_f and s'_f be two form specification rules that express predicate Y as y. Let s_f and s'_f contain contextual operators that search for conflicting properties of predicate Y in a clause structure. Then different occurrences of the string y cannot be expressed by different rules. Thus if we make a local choice that some occurrence of string y is expressed by rule s_f, then all occurrences of string y are expressed by rule s_f. Hence a locally made choice has a global effect on the computation of the form specification process.

Now more formally, we have to show that given a 3SAT formula φ, we can compute in polynomial time a lexicon L, a set of fund formation rules F, a set of clause structure formation rules R, a set of form specification rules S_f, and a template-filler x, such that

1. if formula φ is satisfiable, then template-filler x is the result of applying form specification rules in S_f to some clause structure $C \in CSP(L, F, R)$.
2. if formula φ is not satisfiable, then there does not exist a clause structure $C \in CSP(L, F, R)$, such that template-filler x is the result of applying form specification rules in S_f to C.

The reduction. Let φ be a 3SAT formula that consists of n variables and m clauses. Without loss of generality we say that all clauses contain three distinct literals of the form p_i or $\overline{p_i}$. The reduction is now as follows.

- Let the lexicon L of the FG-grammar contain two predicates of category F and C, respectively,
$$\text{Formula}_F(x) \quad \text{and} \quad \text{Clause}_C(x).$$
- The grammar contains no term operators and no fund formation rules.
- The set of clause-structure operators, Π, of the FG-grammar is defined as the set
$$\{\pi_i, \nu_i, \mu_0 \mid 1 \leq i \leq n\}.$$
The FG-grammar contains no syntactic or pragmatic functions.
- The clause structure formation process constructs clause structures that consist of
  - a set of operators $\Pi'' \subseteq \Pi$,
  - a predication whose category is F or C,
  - and a, possibly empty, set of satellites, which are clause structures.
- The form specification process contains $3n$ trigger operators
$$\{\mu_i, \mu_{f_i}, \mu_{t_i} \mid 1 \leq i \leq n\}.$$

The set of form specification rules consists of ordinary, contextual and auxiliary form specification rules. The ordinary form specification rules are specified as pair $(I, f,)$, where I is the set of properties that the input must contain and f is the function that computes how the predicate is expressed. Likewise, the contextual form specification rules are specified as triples (I, K, f), where K is an additional collection of sets of properties that other predicates or terms in the clause structure must contain. The auxiliary form specification rules are here specified as pairs (I, T), where I is the set of properties that the input must contain and T is the set of trigger operators added to the input. We assume that the auxiliary rules only introduce auxiliary properties and do not introduce auxiliary predicates.
Now the $2n$ auxiliary form specification rules are specified as the pairs $(1 \leq i \leq n)$:

$$\begin{array}{ll} (\{\mu_{i-1}\}, & \{\mu_i, \mu_{f_i}\}) \\ (\{\mu_{i-1}\}, & \{\mu_i, \mu_{t_i}\}.) \end{array}$$

The FG-grammar contains slightly less than $24n^3$ contextual form specification rules $(1 \leq i < j < k \leq n)$:

$$\begin{array}{lll} (\{C, \pi_i, \pi_j, \pi_k\}, & \{\{F, \mu_{t_i}\}\}, & f^7_{ijk}) \\ (\quad\vdots & , \{\{F, \mu_{t_j}\}\}, & \vdots\ ) \\ (\quad\vdots & , \{\{F, \mu_{t_k}\}\}, & \vdots\ ) \\ (\{C, \nu_i, \pi_j, \pi_k\}, & \{\{F, \mu_{f_i}\}\}, & f^6_{ijk}) \\ (\quad\vdots & , \{\{F, \mu_{t_j}\}\}, & \vdots\ ) \\ (\quad\vdots & , \{\{F, \mu_{t_k}\}\}, & \vdots\ ) \\ (\quad\vdots & , \quad\vdots & , \vdots\ ) \\ (\{C, \nu_i, \nu_j, \nu_k\}, & \{\{F, \mu_{f_i}\}\}, & f^0_{ijk}) \\ (\quad\vdots & , \{\{F, \mu_{f_j}\}\}, & \vdots\ ) \\ (\quad\vdots & , \{\{F, \mu_{f_k}\}\}, & \vdots.\ ) \end{array}$$

The only ordinary form specification rule is specified as

$$(\{\mu_n\}, \quad f_{id}.)$$

The polynomial time computable, honest functions f that are specified in the ordinary and contextual form specification rules are defined as follows. The function f_{id} is the identity function on the body of predicates. The functions f^h_{ijk} $(0 \leq h \leq 7, 1 \leq i < j < k \leq n)$ are defined as

$$\begin{array}{rcl} f^7_{ijk}(\text{body}) & = & \text{body}\text{–}p_i\text{–}p_j\text{–}p_k \\ f^6_{ijk}(\text{body}) & = & \text{body}\text{–}p_i\text{–}p_j\text{–}\overline{p_k} \\ & \cdots & \\ f^0_{ijk}(\text{body}) & = & \text{body}\text{–}\overline{p_i}\text{–}\overline{p_j}\text{–}\overline{p_k} \end{array}$$

- Map the formula $\varphi = \gamma_1 \wedge \ldots \wedge \gamma_m$ onto the template-filler

$$\{(\text{Formula}, \{F, \mu_0\}), \tau_1, \ldots \tau_m\},$$

where a clause $\gamma_i = l_1^i \vee l_2^i \vee l_3^i$ is mapped onto the template-filler

$$\tau_i = (\text{Clause–}l_1^i\text{–}l_2^i\text{–}l_3^i, \{C, \pi_1^i, \pi_2^i, \pi_3^i\})$$

where $\pi_j^i = \pi_k$ iff $l_j^i = p_k$, and $\pi_j^i = \nu_k$ iff $l_j^i = \overline{p_k}$.

7.4.6. LEMMA. *The universal version of the form specification problem is NP-hard.*

Proof.(Sketch of the proof.)
We have to show that the reduction above is computable in polynomial time, and preserves answers. It is easy to see that the reduction takes polynomial time in the length of the 3SAT formula.

We prove that the reduction preserves answers in two steps. First, we consider the case when φ is satisfiable. Second, we consider the case when φ is not satisfiable. Let $\varphi = (l_1^1 \vee l_2^1 \vee l_3^1) \wedge \ldots \wedge (l_1^m \vee l_2^m \vee l_3^m)$ be mapped onto template-filler

$$x_\varphi = \{(\text{Formula}, \{F, \mu_0\}), \{(\text{Clause–}l_1^i\text{–}l_2^i\text{–}l_3^i), \{C, \pi_1^i, \pi_2^i, \pi_3^i\}) \mid 1 \leq i \leq n\}\}.$$

If φ is satisfiable, then there is a, consistent, assignment g such that each clause contains at least one literal, p_i or $\overline{p_i}$, that is true. We claim, without proof, that the following clause structure, C_φ, is expressed as the template-filler x_φ if trigger operator μ_{t_i} is added to the word Formula and g assigns true to variable p_i, or μ_{f_i} is added and g assigns false to variable p_i. The clause structure C_φ consists of

- the set of operators $\{\mu_0\}$,
- the predication $\text{Formula}_F(x)$,
- and the set of satellites $\{(h, S_i) \mid 1 \leq i \leq n\}$, where h is some semantic function, which we will neglect, and S_i a clause structure that consists of
  - the set of operators $\{\pi_1^i, \pi_2^i, \pi_3^i\}$, where $\pi_j^i = \pi_k$ iff $l_j^i = p_k$, and $\pi_j^i = \nu_k$ iff $l_j^i = \overline{p_k}$,
  - the predication $\text{Clause}_C(x)$,
  - and no satellites.

If φ is not satisfiable, then there is no, consistent, assignment g such that each clause contains at least one literal, p_i or $\overline{p_i}$, that is true. Let us assume, however, that there exists a clause structure $C_\varphi \in OSP(L, F, R)$ such that applying the form specification rules from S_f to C_φ results in x.

Let us consider the first template-filler in x_φ: (Formula, $\{F, \mu_0\}$). The grammar shows that the word Formula stems from the predicate Formula_F. The property μ_0 indicates that Formula has property μ_0 in the underlying clause structure C_φ. From the form specification rules we see that the predicate Formula_F with property μ_0 is expresses by a sequence of n auxiliary form specification rules, which adds the trigger operators $\mu_1 \ldots \mu_n$ and, moreover, the trigger operators $\mu'_1 \ldots \mu'_n$, where μ'_i is either μ_{f_i} or μ_{t_i}. The sequence of rules cannot have added both trigger operators μ_{f_i}

and μ_{t_i} because μ_{f_i} and μ_{t_i} are triggered by the same operator μ_{i-1}, but by different rules.

Now consider any other template-filler in x_φ: (Clause–l_1^i–l_2^i–l_3^i, $\{C, \pi_1^i, \pi_2^i, \pi_3^i\}$). Let us consider a generic instance of this template-filler, for instance, let l_1^i be $\overline{p_1}$, l_2^i be $\overline{p_2}$, l_3^i be p_3. Then π_1^i is ν_1, π_2^i is ν_2, and π_3^i is π_3. The set of form specification rules shows that there are three, contextual, form specification rules that may have expressed predicate Clause$_C$ with the set of properties $\{C, \nu_1, \nu_2, \pi_3\}$ as surface form Clause–$\overline{p_1}$–$\overline{p_2}$–p_3:

$$(\{C, \nu_1, \nu_2, \pi_3\},\ \{\{F, \mu_{f_1}\}\},\ f_{123}^2\,)$$
$$(\{C, \nu_1, \nu_2, \pi_3\},\ \{\{F, \mu_{f_2}\}\},\ f_{123}^2\,)$$
$$(\{C, \nu_1, \nu_2, \pi_3\},\ \{\{F, \mu_{t_3}\}\},\ f_{123}^2.)$$

The first rule assume a context $\{F, \mu_{f_1}\}$, the second $\{F, \mu_{f_2}\}$, and the third $\{F, \mu_{t_3}\}$. Because the template-filler contains only one predicate with property F, the contexts refer to the trigger operators of the predicate Formula$_F$. So in order to express this predicate at least one of the trigger operators $\mu_{f_1}, \mu_{f_2}, \mu_{t_3}$ was added to the predicate Formula$_F$.

In the same manner, each of the remaining template-fillers claim that some trigger operator was added to the predicate Formula$_F$. We stated before that the trigger operators μ_{f_i} and μ_{t_i} cannot be added both to the predicate Formula$_F$. So the template-fillers that contain property C make consistent claims about the trigger operators of the predicate Formula$_F$. Moreover, the template-fillers claim that μ_{f_i} was added to Formula$_F$ only if they contain property ν_i, and μ_{t_i} if they contain π_i.

It should be clear that the trigger operators μ_{f_i}, μ_{t_i} $(1 \leq i \leq n)$ that are added to Formula$_F$ correspond to a satisfying assignment of formula φ. But this contradict the starting situation that φ is not satisfiable. Therefore, the assumption that there exists a clause structure $C_\varphi \in OSP(L, F, R)$ that is expressed as x_φ, is false. We conclude that the reduction preserves answers. □

The previous lemma provided a lower bound on the complexity of the form specification process. By means of the following six facts we can prove that the form specification problem is included in NP.

7.4.7. FACT. Let x be a template-filler, and let G be an FG-grammar which contains N elementary entities and M rules:

1. In the form specification process only auxiliary properties are deleted.
2. An auxiliary form specification rule introduces at most N auxiliary properties.
3. In order to express the template-filler x, the number of auxiliary form specification rules that are applied is linear with respect to the size of the grammar and the template-filler.
4. Because the form specification rules must express all predicates in the clause structure, the clause structure contains at most as many predicates as the size of the template-filler x.
5. A clause structure contains at most as many terms without a body, which are not expressed as some surface form, as the size of the template-filler because

these terms have filled an argument position in the clause structure and this position must have left a property in the clause structure.

6. The size of terms without a body is linear with respect to the size of the grammar G because terms without a body are basic terms.

7.4.8. THEOREM. *The universal version of the form specification problem is NP-complete.*

Proof. Given the previous lemma, we now only have to provide an *NP* upper bound on the complexity. Given the facts stated above the following argumentation proves this upper bound.

The six facts above explain that if C is the clause structure for template-filler x, then C and x have almost the same size. Moreover, if a sequence of form specification rules express some clause structure as template-filler x, then the size of the sequence is linear in the size of x. Hence a clause structure C and a sequence of form specification rules for template-filler x can be guessed. Once the clause structure and the series of form specification rules are given, it is easy to check that C is expressed as x because the number of auxiliary properties is linear in the size of x. □

Hence a nondeterministic, polynomial time algorithm exists that determines whether an arbitrary template-filler can be generated by an FG-grammar, and no substantially more efficient algorithm exists, unless $P = NP$.

7.5 Order Specification Process

A sketch of the situation. The order specification process orders a sequence of words linearly, on the basis of some properties of these words. The words are ordered by a means of a template, which consists of positions that may be filled by the words. The positions may only be filled if some conditions are satisfied. These conditions are formulated as rules that are associated with the template.

7.5.1 An Informal Introduction

Little has been said in the literature about the precise nature of the order specification process. The example that we will give stems from Dik (1989). We recommended Bakker's (1994, §7.1.2) discussion on the formal representation of form and order specification rules to the interested reader. In this section, we denote a word and its relevant set of properties as a pair. For instance, the word "John" with the property "$\texttt{Subj}$" is represented as (John, $\{\texttt{Subj}\}$).

Suppose that the previous form specification process has resulted in the following sequence of four words with the properties, Aux, $\texttt{Decl}$, V, $\texttt{Subj}$, and $\texttt{Obj}$:

$$(\text{had}, \{Aux\}),\ (\text{kissed}, \{\texttt{Decl}, V\}),\ (\text{John}, \{\texttt{Subj}\}),\ (\text{Mary}, \{\texttt{Obj}\}).$$

Let the order specification process contain the order specification rules for English main clauses, which is given in Table 7.1. This order specification rule consists of

Template:		V_{Aux_0} S V_{Aux_1} V_{Main} O X
Rules:	1	If the word with property V has the property `Int`, then the word with property Aux is placed in position V_{Aux_0}.
	2	The word with property `Subj` is placed in position S.
	3	If there is (another) word with property Aux, then this word is placed in position V_{Aux_1}.
	4	The word with property V is placed in position V_{Main}.
	5	If there is a word with property `Obj`, then this word is placed in position O.
	6	All remaining words are placed in position X.

Table 7.1: An order specification rule.

a template with six positions, and six associated rules. The first rule mentions the property `Int`, which stands for interrogative. This property results from the clause-structure operator that expresses the illocution of a clause structure. The second and fourth rule presuppose that there are words with property `Subj`, V. We assume that if there is no word with property `Subj` or V in the sequence, then this template is not applicable for that sequence.

According to this template and these six rules, the ordering of the four words above results in the sentence

"John had kissed Mary."

If the word "kissed" would have property `Int` instead of `Decl`, then the ordering of the four words would have resulted in the question

"Had John kissed Mary?"

7.5.2 The Formalization

Above we presented an example that should make clear what the order specification process does: Given some set of order specification rules, the order specification process maps a template-filler that resulted from the form specification process onto a string. In this section, we will formalize the order specification process. We will not formalize the input and output of the order specification process in this section. The input of the order specification process, the template-filler, is formalized in Section 7.4.2. The output of the order specification process is simply a string. Thus we will start this section with a formalization of the notion of *order specification rules*. Next we compare this formalization and the examples. We end with a formulation of the *order specification problem*.

The order specification rules consist of a template and a sequence of rules. The *template* is a series of positions. The positions are often labeled by mnemonics. These labels can be chosen completely free, as long as they are used in a consistent way.

The *rules* state under which conditions which element from the template-filler will fill which position in the template. The rules associated with a template are applied by a deterministic procedure and each element of a template-filler is place in exactly one position. In principle, the positions may be filled by more than one element of the template-filler. The ordering of the elements in such positions is then determined by an other order specification rule.

7.5.1. DEFINITION. *Order Specification Rules*
We define the *order specification rules* in the following way.

- An order specification rule is a pair that consists of a template and a sequence of rules.
- The *template* is a sequence of labeled positions.
- The *rules* are specified as triples that consist of a set of conditions, a set of properties, and a position. The intended meaning of the rule is if the template-filler satisfies each of the conditions, then place the words with the specified properties in the indicated position.
- The *conditions* are stated as boolean combination of the presence of a word with some properties in the template-filler.

Let us now compare the formalization given above and the example presented in the Table 7.1. The template given of the order specification rule consists of six positions, which are mnemonics. The first rule may be specified by the triple with the set of conditions $\{\mathtt{Int} \wedge V\}$, the set of properties $\{Aux\}$, and the position V_{Aux_0}, which means the following:

If the template-filler contains
  a word with the properties `Int` and V,
then place the words with
  property Aux
in the template position with label
  V_{Aux_0}.

The second rule may be specified in a similar way, by an empty set of conditions, the set of properties $\{\mathtt{Subj}\}$, and the position S, which means

If the template-filler contains
  anything,
then place the words with
  property `Subj`
in the template position with label
  S.

The third rule may be specified as follows because the phrase "another" is implicitly account for by the ordering of the rules.

If the template-filler contains
  a word with the property Aux,

then place the words with
property Aux
in the template position with label
V_{Aux_1}.

The fourth and fifth rules are specified similarly. The six rule can be formulated because the previous rules have used up their material. So, "all remaining words" is in fact represented as "all words."

In the examples we have seen now, no real use is made of the fact that the template-filler has to satisfy a set of conditions. However, Bakker (1994, §7.1.2) gives a source where the following condition is used: "If the verb contains three arguments, ..." The set-construction in the formalization presented above can probably be used to formulate this condition.

Now let us formulate the order specification problem. We introduce an oracle C that answers the form specification problem from Definition 7.4.5.

7.5.2. DEFINITION. *The Order Specification Problem* ($\mathrm{OSP}(L, F, R, S_f, S_o)$)
Given a lexicon L, a set of fund formation rules F, a set of clause structure formation rules R, a set of form specification rules S_f, an oracle C for "$x \in \mathrm{FSP}(L, F, R, S_f)$?," and a set of order specification rules S_o.
INSTANCE: A string w.
QUESTION: Is there a template-filler x ($x \in \mathrm{FSP}(L, F, R, S_f)$) such that x is expressed as w by S_o?

7.5.3 The Order Specification Problem is *NP*-complete

Now we will first prove a reduction from the well-known *NP*-complete problem 3-SATISFIABILITY (3SAT) to the universal version of $\mathrm{OSP}(L, F, R, S_f, S_o)$. Second, we will show that the universal version of $\mathrm{OSP}(L, F, R, S_f, S_o)$ is in fact *NP*-complete.

We have to show that given a 3SAT formula φ, we can compute in polynomial time a lexicon L, a set of fund formation rules F, a set of clause structure formation rules R, a set of form specification rules S_f, a set of order specification rules S_o, and a string w, such that

1. if formula φ is satisfiable, then string w is the result of applying order specification rules in S_o to some template-filler $x \in FSP(L, F, R, S_f)$;
2. if formula φ is not satisfiable, then there does not exist a template-filler $x \in FSP(L, F, R, S_f)$, such that string w is the result of applying order specification rules in S_o to x.

The reduction. Let φ be a 3SAT formula that consists of n variables and m clauses. Without loss of generality we say that all clauses contain three distinct literals of the form p_i or $\overline{p_i}$. The reduction is now defined as follows.

- The lexicon consists of two predicates of category F and C, respectively,

$$\mathrm{Formula}_F(x_1)\ldots(x_m) \quad \text{and} \quad \mathrm{Clause}_C(x_1).$$

- The set of term operators, Ω, is defined as

$$\{p_i, \overline{p_i} \mid 1 \leq i \leq n\}.$$

- The grammar contains only fund formation rules to construct terms with at most three term operators from predicates of category C. That is,

$$(\omega x : \mathrm{pred}_C(x))$$

is a term provided that $\omega \subseteq \Omega$, $|\omega| \leq 3$, and $\mathrm{pred}_C(x)$ is in a predicate of the lexicon.
- The set of clause-structure operators, Π, is defined as

$$\{t_i \mid 1 \leq i \leq n\}.$$

- The clause structure formation process constructs clause structures that consist of
  - a set of operators $\Pi' \subseteq \Pi$,
  - a predication whose category is F,
  - and an empty set of satellites.

  The clause structures will look like

$$\Pi' \,\mathrm{Formula}_F(\omega_1 x_1 : \mathrm{Clause}_C(x_1)) \ldots (\omega_m x_m : \mathrm{Clause}_C(x_m))$$

- The set of form specification rules will consist of $O(n^3)$ rules.
  - Regardless of the set of clause-structure operators Π', predicates of category F are mapped onto the word that is indicated by their body.
  - Depending on the set of term operators ω_i, predicates of category C are mapped onto the word that is indicated by their body plus the suffix indicated by the term operators.

  To be more precise, for all sets of clause-structure operators, Π', the predicate $\mathrm{Formula}_F$ is mapped onto the template filler (Formula, $\{F\} \cup \Pi'$). For each set of term operators, say $\omega = p_i\,\overline{p_j}\,p_k$, the term $(\omega x : \mathrm{Clause}_C(x))$ is mapped onto the template-filler (Clause–p_i–$\overline{p_j}$–p_k, $\{C, p_i, \overline{p_j}, p_k\}$).
- The grammar will contain only one order specification rule. This order specification rule is specified by $2n+2$ rules and a template with three positions. The template is defined as follows

$$C_0 \; F \; C_1$$

  The rules are defined as follows. The first rule is defined as the triple with the empty set of conditions, the set of properties $\{F\}$, and the position F. The $2i$-th rule is defined as the triple with the set of conditions $\{F \wedge t_i\}$, the set of properties $\{C, p_i\}$, and the position C_1. The $(2i+1)$-st rule is defined as the triple with the set of conditions $\{\neg(F \wedge t_i)\}$, the set of properties $\{C, \overline{p_i}\}$, and the position C_1. The final rule is defined as the triple with the empty set of conditions, the empty set of properties, and the position C_0.

The first rule can be read as: "**If** the template-filler contains anything, **then** place the words with property F **in** the template position with label F." The final rule can be read as: "**If** the template-filler contains anything, **then** place the remaining words **in** the template position with label C_0." The other rules can be read as either one of the following two paraphrases: "**If** the template-filler contains a word with the properties F and t_i, **then** place the words with property C and p_i **in** the template position with label C_1" or "**If** the template-filler contains no word with the properties F and t_i, **then** place the words with property C and $\overline{p_i}$ **in** the template position with label C_1."

- The string w is computed as follows.
  - A clause $(l_1 \vee l_2 \vee l_3)$ (l_i a literal) is mapped onto string "Clause–l_1–l_2–l_3."
  - A formula $\varphi = \gamma_1 \wedge \ldots \wedge \gamma_m$ is mapped onto the string "Formula $w_1 \ldots w_m$," where clause γ_i is mapped onto string w_i.

7.5.3. Lemma. *The universal version of the order specification problem is NP-hard.*

Proof. (Sketch of the proof.)
We have to show that the reduction above is computable in polynomial time, and preserves answers. It is easy to see that the reduction takes polynomial time in the length of the 3SAT formula.

We prove that the reduction preserves answers in two steps. First, we consider the case when φ is satisfiable. Second, we consider the case when φ is not satisfiable. Let $\varphi = (l_1^1 \vee l_2^1 \vee l_3^1) \wedge \ldots \wedge (l_1^m \vee l_2^m \vee l_3^m)$ be mapped onto string w.

If φ is satisfiable, then there is a, consistent, assignment g such that each clause contains at least one literal, p_i or $\overline{p_i}$, that is true. We claim, without proof, that the template-filler

$$\begin{array}{ll}\{(\text{Formula}, \{F\} \cup \Pi'), & (\text{Clause–}l_1^1\text{–}l_2^1\text{–}l_3^1, \{C, l_1^1, l_2^1, l_3^1\}) \ldots \\ & (\text{Clause–}l_1^m\text{–}l_2^m\text{–}l_3^m, \{C, l_1^m, l_2^m, l_3^m\})\ \}\end{array}$$

is expressed as string w, where Π' contains clause-structure operator t_i iff g assigns true to variable p_i.

If φ is not satisfiable, then there is no, consistent, assignment g such that each clause contains at least one literal, p_i or $\overline{p_i}$, that is true. Let us assume, however, that there exists a template-filler $x \in OSP(L, F, R, S_f)$ such that applying the order specification rules from S_o to x results in w. This implies that each word Clause–l_1^i–l_2^i–l_3^i is placed in position C_1. Hence each of these words has either property p_j and the word Formula has property t_j, or property $\overline{p_j}$ and the word Formula does not have property t_j. But, now, the properties of word Formula denote a, consistent, assignment for φ in the following way: g assigns true to variable p_i iff word Formula has property t_i. This contradicts the fact that φ is not satisfiable, therefore the assumption is false. It follows that the reduction preserves answers. □

The following fact will be useful to prove inclusion in NP. The fact follows because when an order specification rule is applied to a template-filler, the result is either exactly the same template-filler, or an ordered sequence of strictly smaller template-fillers.

7.5.4. FACT. Let S_o be a set of m order specification rules, and let x be a template-filler that contains n elements. If there is a sequence of order specification rules that orders the template-filler x, then this sequence is at most nm steps long.

The examples of form and order specification rules that we have seen in the literature seem to suggest that the properties of a word are elementary entities of the grammar, like functions, operators and categories. Because these entities are specified by the grammar, the maximum number of properties of a word is linear in the size of the grammar.

7.5.5. THEOREM. *The universal version of the order specification problem is NP-complete.*

Proof. Given the previous lemma, we now only have to provide an *NP* upper bound for $w \in \mathrm{OSP}(L, F, R, S_f, S_o)$. Given the assumption that the maximum number of properties is bounded by the size of the grammar, the following argumentation proves this upper bound.

1. For the string w, guess a template-filler x.
2. Guess a sequence of order specification rules that eventually express template-filler x as the string w.
3. Apply the sequence of order specification rules to the template-fillers, in a fully deterministic way.
4. Check that the template-filler x is expressed as string w by the guessed rules.
5. Given the template-filler x, consult the oracle C for $x \in \mathrm{FSP}(L, F, R, S_f)$. We claim that the oracle will answer yes iff $w \in \mathrm{OSP}(L, F, R, S_f, S_o)$.

The total number of guesses is polynomially bounded, because the sequence of order specification rules and the number of properties are both linearly bounded in the size of the grammar and string. Given a template-filler x and an order specification rule the result of the order specification rule is computed deterministically. □

Hence given the fund, the set of clause structure formation rules, the set of form specification rules, and the set of order specification rules of an arbitrary FG-grammar. Assume that we have a simple method to decide whether a template-filler is generated by the FG-grammar. Then a nondeterministic, polynomial time algorithm exists that determines whether an arbitrary string can be generated by the FG-grammar. Moreover, no substantially more efficient algorithm exists, unless $P = NP$.

7.6 Conclusions and Further Research

In this chapter, we have considered the various processes from a parsing, bottom-up, point of view. We will now, briefly, consider some of the processes from a generation, top-down, point of view.

Clause structure formation process. A nice formulation of the generation version of this process is the following problem: "Given a collection of predicates and terms, and a set of clause structure formation rules, can these predicates and terms form a clause structure?"

This problem is computable in nondeterministic polynomial time, for the following reasons. If the predicates and terms can form a clause structure, then this clause structure is approximately of the same size as the collection of predicates and terms. Hence this clause structure, say C, can be computed nondeterministically. Given this clause structure C, the above problem boils down to CSP, which is computable in nondeterministic polynomial time.

From and order specification processes. Similar formulations of the generation versions of these processes to the one given for the clause structure formation process are not interesting. The problems describe in such formulations have the trivial answer "Yes," because the two specification processes must always express the structures onto which they are applied.

Now let us consider different formulations of the generation versions for these two processes: "Given a clause structure C (template-filler x) and a set of form (order) specification rules, determine a template-filler y (string w) such that C (x) is expressed as y (w) by the specification rules."

Both of these problems are computable in nondeterministic polynomial time, for the following reasons. The output of these problems is almost of the same size as the input. Hence the output can be determined nondeterministically. Given the output, the above problems boils down to FSP and OSP, which are computable in nondeterministic polynomial time.

A prerequisite for the complexity analyses in this chapter is a formalization of the various specification- and formation-problems. Because the formalization of FG is still very sketchy, we often have formalized FG ourselves. We have confined ourselves to the kernel of the theory of Functional Grammar. We claim that the formalizations that we have given do not violate the actual use of FG by the linguistic community. The complexity analyses force us to give a precise characterization of various rules and structures of FG. We will consider it a token of recognition if the characterizations given will raise some discussions.

Furthermore, this research on the computational complexity of FG has identified several sources of intractability within FG. The complexity analyses indicate that the fund formation, the form specification and the order specification are the most difficult parts of an FG-grammar. The complexity of these processes is due to the fact that the principles of FG to avoid deletions, etcetera, (Dik 1989, §1.6) only apply to lexical material, and not to operators, or functions. For instance, a process may introduce auxiliary properties. Because these auxiliary properties need not be passed on to the next process, the information encoded by these properties may be lost. Obviously, if this information is essential, then these auxiliary properties are a source of complexity.

The complexity analyses suggest that FG should constrain the fund formation, the form specification and the order specification processes. That is, place constraints on the phenomena that cause this complexity. For example, the auxiliary specification rules of the form specification process cause the complexity because they may introduce trigger operators that trigger other auxiliary specification rules. A possible constraint would be to demand that trigger operators only trigger ordinary or contextual specification rules

We see two main directions for further research. The one direction should concentrate on the further formalization of FG. In this chapter we presented definitions of various notions in FG. Further research must indicate whether these definitions are powerful enough to describe all structures that appear in FG. The other direction should concentrate on the complexity results. In this direction the various definitions presented in this chapter serve as a starting point. Given these definitions, one should try to find linguistically motivated restrictions that would lower the complexity of the processes. In particular, the fund formation process deserves special attention.

Chapter 8

Conclusions

In the previous chapters we have applied complexity theory to six grammatical formalisms: restricted attribute-value grammars, Categorial Unification Grammar, Functional Unification Grammar, Head-driven Phrase Structure Grammar, Lexical Functional Grammar, and Functional Grammar. The first five formalisms mentioned are unification-based formalisms, which are based on feature theory. The remaining formalism, Functional Grammar, is based on predicate logic. In particular, we have studied "recognition" problems of these grammatical formalisms. We distinguished the "universal" and the "fixed" variants of these recognition problems.

Our attention aimed at the development of the theories of the grammatical formalisms that we considered. Central to our approach was the idea that the complexity analyses should contribute to this development. Therefore we did not consider isolated components that together form a grammar. Instead, we considered the description of grammars as a whole. The existing literature, however, describes isolated parts of the grammatical formalisms. We had to merge these divers descriptions into full formalizations of the grammatical formalisms. Because the descriptions did not always match, we sometimes had to use our own insight to reach a sufficient formalization of a grammatical formalism. The formalizations that we obtained in this way are one of the side-effects of our research.

In this dissertation we classified the grammatical formalisms on computational grounds. The lower bounds of these classifications are in conformity with the expectations of Barton, Berwick and Ristad (1987). In addition to these lower bounds, we have also provided upper bounds of these classifications. Barton, Berwick and Ristad (1987), by contrast, do not discuss such upper bounds.

In the first instance we consider the unification-based formalisms. Although we confined ourselves to primitive fragments of these formalisms, their recognition problems proved to be NP-complete. However, we conjectured that by means of polynomial extensions we can come from these primitive fragments to descriptions of the full formalisms. Moreover, these polynomial extensions will not increase the complexity of the recognition problems. Further research should confirm this conjecture.

By means of the close connection between the fixed variant of the recognition problems and the weak generative capacity, we were able to settle the weak generative capacity of the unification-based formalisms. The closest connection between complexity and generative capacity was given for the restricted attribute-value grammars (R-AVGs). We proved that the R-AVGs that satisfy the honest parsability constraint, a liberalization of Johnson's (1988) off-line parsability constraint, generate exactly all sets in *NP*. We also showed that relaxing or tightening this honest parsability constraint, let different types of R-AVGs coincide precisely with other well-known time complexity classes. This shows that the complexity theory provides more fine-tuned tools to determine the expressivity of a grammatical formalism than the generative capacity, which is traditionally used. In our opinion, it would be interesting to find more tight relations between complexity classes and grammatical formalisms.

We proved an *NP*-hard lower bound for Categorial Unification Grammar (CUG) by means of a reduction from 3-SATISFIABILITY. The *NP*-hard lower bounds for the other unification-based formalisms were proven by means of simulations of CUG. We showed that as a direct consequence of these simulations formal properties that hold for CUG also hold for the other unification-based formalisms. These simulations, or translations, are a second kind of side-effect of our research. A further study of the formal properties of CUG may also shed light on the formal properties of these formalisms. A related topic for future research is to compare the unification-based formalisms and Functional Grammar (cf., Hoekstra, van der Hulst and Moortgat 1981).

The intractability results that have been obtained in this thesis do not only contribute to the development of theories, but are also useful from a more practical point of view. The intractability results prove that there do not exist efficient algorithms for general implementations of the six grammatical formalisms. Hence implementations for these formalisms are either inefficient, or do not implement the full theory.

Not withstanding the intractability of the full theory, efficient implementations for specific grammars might exist. A feasible algorithm for some particular grammar may cleverly use some specific characteristics of that grammar. However, the intractability results state that this algorithm will only be useful for that particular grammar. So if an efficient algorithm is also needed for another grammar, some clever computational linguist has to redesign a fully new algorithm.

The complexity analyses of the six grammatical formalisms yielded a few indirect results, two of which we mentioned above. The main results that we obtained in this indirect way are the formalizations of the six grammatical formalisms. In addition, the analyses of the formalizations resulted in suggestions for extensions and restrictions of the theories of the grammatical formalisms. Last, but not least, we presented translations of one grammatical formalism into other grammatical formalisms. These translations themselves revealed several connections between the unification-based formalisms.

Bibliography

Baader, F., Bürckert, H.-J., Nebel, B., Nutt, W. and Smolka, G.: 1993, On the expressivity of feature logics with negation, functional uncertainty, and sort equations, *Journal of Logic, Language and Information* **2**(1), 1–18.

Bach, E.: 1983a, Generalized categorial grammars and the English auxiliaries, *in* F. Heny and B. Richards (eds), *Linguistic Categories: Auxiliaries and Related Puzzles 2*, D. Reidel, Dordrecht – Boston, pp. 101–120.

Bach, E.: 1983b, On the relationship between word-grammar and phrase-grammar, *Natural Language and Linguistic Theory* **1**, 65–89.

Bakker, D.: 1994, *Formal and Computational Aspects of Functional Grammar and Language Typology*, Ph.D. dissertation, University of Amsterdam.

Balcázar, J. L., Díaz, J. and Gabarró, J.: 1988, *Structural Complexity I*, Springer-Verlag, New York – Heidelberg – Berlin.

Barton Jr., G. E., Berwick, R. C. and Ristad, E. S.: 1987, *Computational Complexity and Natural Language*, MIT Press, Cambridge, MA.

van Benthem, J. and ter Meulen, A. (eds): To appear, *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam – New York.

Blackburn, P. and Spaan, E.: 1993, A modal perspective on the computational complexity of attribute value grammar, *Journal of Logic, Language and Information* **2**(2), 129–169.

Book, R. V.: 1978, On the complexity of formal grammars, *Acta Informatica* **9**, 171–181.

Bouma, G.: 1988, Modifiers and specifiers in categorial unification grammar, *Linguistics* **26**, 21–46.

Bouma, G.: 1993, *Nonmonotonicity and Categorial Unification Grammar*, Ph.D. dissertation, University of Groningen.

Bresnan, J. (ed.): 1982, *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, MA.

Buszkowski, W.: 1988a, Generative power of categorial grammars, *in* R. T. Oehrle, E. Bach and D. Wheeler (eds), *Categorial Grammar and Natural Language Structures*, Vol. 32 of *Studies in Linguistics and Philosophy*, D. Reidel, Dordrecht – Boston.

Buszkowski, W.: 1988b, Three theories of categorial grammar, *in* W. Buszkowski, W. Marciszewski and J. van Benthem (eds), *Categorial Grammar*, Linguistic and Literary Studies in Eastern Europe (LLSEE), John Benjamins Publishing Company, Amsterdam–Philadelphia, pp. 57–84.

Buszkowski, W., Marciszewski, W. and van Benthem, J. (eds): 1988, *Categorial Grammar*, Linguistic and Literary Studies in Eastern Europe (LLSEE), John Benjamins Publishing Company, Amsterdam–Philadelphia.

Chomsky, N.: 1956, Three models for the description of language, *IRE Transactions on Information Theory* **2**(3), 113–124.

Connolly, J. H. and Dik, S. C. (eds): 1989, *Functional Grammar and the Computer*, Vol. 10 of *Functional Grammar Series*, Foris Publications, Dordrecht.

Dailey, D. P.: 1986, The extraction of a minimum set of semantic primitives from a monolingual dictionary is NP-complete, *Computational Linguistics* **12**(4), 306–307.

Dik, S. C.: 1978, *Functional Grammar*, Vol. 37 of *North-Holland Linguistic Series*, North-Holland, Amsterdam.

Dik, S. C.: 1980, *Studies in Functional Grammar*, Academic Press, New York, NY.

Dik, S. C.: 1989, *The Theory of Functional Grammar, Part I: The Structure of the Clause*, Vol. 9 of *Functional Grammar Series*, Foris Publications, Dordrecht.

Dik, S. C.: 1992, *Functional Grammar in Prolog: An Integrated Implementation for English, French, and Dutch*, Vol. 2 of *Natural Language Processing*, Mouton de Gruyter, Berlin.

Dowty, D. R., Karttunen, L. and Zwicky, A. M. (eds): 1985, *Natural Language Parsing*, Cambridge University Press, New York, NY.

Earley, J.: 1970, An efficient context-free parsing algorithm, *Communications of the Association for Computing Machinery* **13**(2), 94–102.

Garey, M. R. and Johnson, D. S.: 1979, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, CA.

Heny, F. and Richards, B. (eds): 1983, *Linguistic Categories: Auxiliaries and Related Puzzles 2*, D. Reidel, Dordrecht – Boston.

Hoekstra, T., van der Hulst, H. and Moortgat, M. (eds): 1981, *Perspectives on Functional Grammar*, Foris Publications, Dordrecht.

Hopcroft, J. E. and Ullman, J. P.: 1979, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, MA.

Janssen, T. M. V.: 1989, Towards a universal parsing algorithm for functional grammar, *in* J. H. Connolly and S. C. Dik (eds), *Functional Grammar and the Computer*, Vol. 10 of *Functional Grammar Series*, Foris Publications, Dordrecht, pp. 64 – 75.

Johnson, M.: 1988, *Attribute-Value Logic and the Theory of Grammar*, Vol. 16 of *CSLI Lecture Notes*, CSLI, Stanford.

Kaplan, R. and Bresnan, J.: 1982, Lexical-functional grammar: a formal system for grammatical representation, *in* J. Bresnan (ed.), *The Mental Representation of*

Grammatical Relations, MIT Press, Cambridge, MA, pp. 173–281.

Karttunen, L. and Kay, M.: 1985, Parsing in a free word order language, *in* D. R. Dowty, L. Karttunen and A. M. Zwicky (eds), *Natural Language Parsing*, Cambridge University Press, New York, NY, pp. 279–306.

Kasper, R. T. and Rounds, W. C.: 1990, The logic of unification in grammar, *Linguistics and Philosophy* **13**, 35–58.

Kay, M.: 1984, Functional unification grammar: A formalism for machine translation, *Proceedings 10th International Conference on Computational Linguistics*, Stanford University Press, Stanford, CA, pp. 75–78.

Kay, M.: 1985, Parsing in functional unification grammar, *in* D. R. Dowty, L. Karttunen and A. M. Zwicky (eds), *Natural Language Parsing*, Cambridge University Press, New York, NY, pp. 251–278.

King, P.: 1994, An expanded logical formalism for head-driven phrase structure grammar, *Arbeitspapiere des sfb 340*, University of Tübingen.

Kwee Tjoe Liong: 1994, *Programmer's Reflections on Functional Grammar: More Excercises in Computational Theoretical Linguistics*, Ph.D. dissertation, University of Amsterdam.

Lambek, J.: 1988, The mathematics of sentence structure, *in* W. Buszkowski, W. Marciszewski and J. van Benthem (eds), *Categorial Grammar*, Vol. 25 of *Linguistic and Literary Studies in Eastern Europe (LLSEE)*, John Benjamins Publishing Company, Amsterdam–Philadelphia, pp. 153–172.

Landsbergen, J.: 1981, Adaption of Montague grammar to the requirements of parsing, *in* J. Groenendijk, T. M. V. Janssen and M. Stokhof (eds), *Formal Methods in the Study of Language, part 2*, Centre for Mathematics and Computer Science, Amsterdam, pp. 399 – 419. MC Tract 136.

Manandhar, S.: 1995, Deterministic consistency checking of LP constraints, *Proceedings 7th Meeting EACL*, University College Dublin.

Moortgat, M.: 1988, *Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus*, Vol. 9 of *Groningen–Amsterdam Studies in Semantics (GRASS)*, Foris Publications, Dordrecht.

Nakanishi, R. et al.: 1992, On the generative capacity of Lexical-Functional Grammars, *IEICE Transactions on Information and Systems* **75-D**(7), 509–516. ISSN: 09168532.

Oehrle, R. T., Bach, E. and Wheeler, D. (eds): 1988, *Categorial Grammar and Natural Language Structures*, Vol. 32 of *Studies in Linguistics and Philosophy*, D. Reidel, Dordrecht – Boston.

O'Shea, T. (ed.): 1985, *Advances in Artificial Intelligence*, Elsevier Science Publishers, Amsterdam – New York.

Partee, B. H., ter Meulen, A. and Wall, R. E. (eds): 1990, *Mathematical-Methods in Linguistics*, Vol. 30 of *Studies in Linguistics and Philosophy*, Kluwer Academic Publishers, Dordrecht – Boston.

Pereira, F. C. and Warren, D.: 1983, Parsing as deduction, *Proceedings 21th Annual Meeting of ACL*, pp. 137–144.

Perrault, C. R.: 1984, On the mathematical properties of linguistic theories, *Com-*

putational Linguistics **10**(3–4), 165–176.

Pollard, C. J. and Sag, I. A.: 1987, *Information-Based Syntax and Semantics: Volume 1 - Fundamentals*, Vol. 13 of *CSLI Lecture Notes*, CSLI, Stanford.

Pollard, C. J. and Sag, I. A.: 1994, *Head-Driven Phrase Structure Grammar*, Studies in Contemporary Linguistics, CSLI & University of Chicago Press, Stanford & Chicago.

Ritchie, G.: 1984, Simulating a turing machine using functional unification grammar, *in* T. O'Shea (ed.), *Proceedings ECAI.84: Advances in Artificial Intelligence*, Elsevier Science Publishers, Amsterdam – New York, pp. 127–136. Also published in Advances in Artificial Intelligence, 1985.

Ritchie, G.: 1985, Simulating a turing machine using functional unification grammar, *in* T. O'Shea (ed.), *Advances in Artificial Intelligence*, Elsevier Science Publishers, Amsterdam – New York, pp. 285–294. Edited version of proceedings ECAI.84, Pisa.

Ritchie, G.: 1986, The computational complexity of sentence derivation in functional unification grammar, *Proceedings 11th International Conference on Computational Linguistics*, Bonn, pp. 584–586.

Rosetta, M.: 1994, *Compositional Translation*, International Series in Engineering and Computer Science, Kluwer Academic Publishers, Dordrecht – Boston.

Rounds, W. C.: 1991, The relevance of computational complexity theory to natural language processing, *in* P. Sells, S. M. Shieber and T. Wasow (eds), *Foundational Issues in Natural Language Processing*, MIT Press, Cambridge, MA, pp. 9–30.

Rounds, W. C.: To appear, Feature logics, *in* J. van Benthem and A. ter Meulen (eds), *Handbook of Logic and Language*, Elsevier Science Publishers, Amsterdam – New York.

Seki, H., Nakanishi, R., Kaji, Y., Ando, S. and Kasami, T.: 1993, Parallel multiple context-free grammars, finite state translation systems, and polynomial-time recognizable subclasses of Lexical-Functional Grammars, *Proceedings 31st Annual Meeting ACL*, pp. 130–139.

Sells, P.: 1985, *Lectures on Contemporary Syntactic Theories*, Vol. 3 of *CSLI Lecture Notes*, CSLI, Stanford.

Shieber, S. M.: 1986, *An Introduction to Unification-Based Approaches to Grammar*, Vol. 4 of *CSLI Lecture Notes*, CSLI, Stanford.

Smolka, G.: 1992, Feature-constraint logics for unification grammars, *Journal of Logic Programming* **12**(1), 51–87.

Sudkamp, T. A.: 1988, *Languages and Machines: An introduction to the Theory of Computer Science*, Addison Wesley, Reading, MA.

Torenvliet, L. and Trautwein, M.: 1995a, A note on the complexity of restricted attribute-value grammars, *in* T. Andernach and A. Nijholt (eds), *Proceedings Computational Linguistics in the Netherlands Meeting*, University Twente.

Torenvliet, L. and Trautwein, M.: 1995b, A note on the complexity of restricted attribute-value grammars, *ILLC Research Report and Technical Notes Series CT-95-02*, University of Amsterdam, Amsterdam.

Trautwein, M.: 1992, On the complexity of feature structures in categorial unification grammar, *in* P. Dekker and M. Stokhof (eds), *Proceedings Eighth Amsterdam Colloquium*, Amsterdam, pp. 587 – 600.

Trautwein, M.: 1995a, Assessing complexity results in feature theories, *ILLC Research Report and Technical Notes Series LP-95-01*, University of Amsterdam, Amsterdam.

Trautwein, M.: 1995b, The complexity of structure sharing in unification-based grammars, *in* T. Andernach and A. Nijholt (eds), *Proceedings Computational Linguistics in the Netherlands Meeting*, University Twente.

Trautwein, M.: 1995c, Sources of complexity in functional grammar, *in* S. Fischer and M. Trautwein (eds), *Proceedings Accolade '95*, Dutch Graduate School in logic.

Uszkoreit, H.: 1986, Categorial unification grammars, *Proceedings 11th International Conference on Computational Linguistics*, Bonn, pp. 187–194.

Zielonka, W.: 1981, Axiomatizability of Ajdukiewicz-Lambek calculus by means of cancellation schemes, *Zeitschrift für mathematische Logik und Grundlagen der Mathematik* **27**, 215–224.

Index

application rule, *see* combinatory rule
argument position, 142, 146
arity, 142, 146
attribute, 43, 45
attribute-value grammar, 17
  restricted, 18, 63, 82, 107, 134
  satisfying HPC, 28, 37
attribute-value language, 19, 20
  formulas, 19
  terms, 19
attribute-value matrix, 17, 32, 42, 43, 54, 65, 86, 109
  lexical, 86
  phrasal, 86
  well-formed, 87, 89, 94, 95
auxiliary form specification rule, 165, 167
AVG, *see* attribute-value grammar
AVL, *see* attribute-value language
AVM, *see* attribute-value matrix

body, 142
box-label, 44, 110

Categorial Unification Grammar, 39, 54, 68, 89, 112
category
  CG, 52
  CUG, 54
  FG, 142
CFG, *see* context-free grammar
CFL, *see* language, CFG
CG, *see* classical Categorial Grammar
chain rule, 18, 20, 28
classical Categorial Grammar, 52
clause structure, 137, 153, 155, 164
clause structure constraint, 156, 157
clause structure formation problem, 155, 159
clause structure formation process, 137, 153, 155
clause structure formation rule, 155, 156
clause structure insertion rule, 156
clause-structure operator, 155
combinatory rule, 119
  CG, 52
  CUG, 55
  FUG, 71
  HPSG, 96
  LFG, 109, 120
complement-daughter, 85
complexity class
  NEXP, 18, 29
  NP, 40, 63, 83, 108, 134, 139
  NP-complete, 40, 62, 81, 107, 133, 139, 140, 146, 176, 179
  NP-hard, 40, 61, 79, 104, 130, 131, 139, 140
  PSPACE, 18
  P, 40, 139
constituent structure tree, 19, 109

annotated, 19
context-free grammar, 29
Greibach normal form, 29
context-sensitive grammar, 64, 83, 108, 135
contextual form specification rule, 165, 166
CSG, *see* context-sensitive grammar
CSL, *see* language, CSG
CSP, *see* clause structure formation problem
CST, *see* constituent structure tree
CUG, *see* Categorial Unification Grammar
CUG-category, *see* category, CUG

decidable, 138, 151
decision problem, 40, 138, 159
derive
CG, 53
CUG, 55
FUG, 75
HPSG, 99
LFG, 125
description language F_L, 46
formula, 42, 46
primitive, 46
terms, 46
deterministic polynomial space, *see* complexity class, $PSPACE$
deterministic polynomial time, *see* complexity class, P
detour, 80, 105
distinguished AVM
FUG, 70, 71
HPSG, 93, 96
distinguished category
CUG, 55
distinguished nonterminal
LFG, 120

elementary entity, 145, 179
entry
CG, 52
CUG, 54
HPSG, 93
LFG, 119
ϵ-rule, 18, 20, 28

f-description, 110, 125
inconsistent, 112
f-edge, 19
feature-graph, 17, 19, 42, 43
FFP, *see* fund formation problem
FFR, *see* fund formation rule
FG, *see* Functional Grammar
form specification problem, 164, 168, 176
form specification process, 137, 155, 163–165, 173
form specification rule, 163, 164
FRP, *see* recognition problem, fixed
FSP, *see* form specification problem
FUG, *see* Functional Unification Grammar
function
pragmatic, 154
semantic, 142, 155
syntactic, 154, 155
Functional Grammar, 137
Functional Unification Grammar, 39, 65
fund, 141, 143
fund formation problem, 145, 149
fund formation process, 137, 145
fund formation rule, 141, 143, 145, 147

generate
CG, 53
CUG, 55
FUG, 75
HPSG, 100
LFG, 126
R-AVG, 20
GNF, *see* context-free grammar, Greibach normal form
grammar
CG, 52
CUG, 55, 65, 69, 85, 91, 109, 115
FG, 137

FUG, 65, 71
HPSG, 85, 95
LFG, 109, 119, 120

Halting Problem, 138, 149
head-daughter, 85
Head-driven Phrase Structure Grammar, 39, 85
honest function, 139, 147–149, 166, 167
honest parsability constraint, 18, 28, 32, 63, 64, 82, 83, 107, 108, 134
HP-AVG, *see* attribute-value grammar, satisfying HPC
HPC, *see* honest parsability constraint
HPSG, *see* Head-driven Phrase Structure Grammar

intractable, 40, 139, 140

language
CFG, 18, 63, 64, 83, 108, 134
CG, 53
CSG, 18, 28
CUG, 55, 63, 64, 82, 107
FUG, 75, 82, 83
HP-AVG, 28
HPSG, 100, 107, 108
LFG, 126, 133, 135
R-AVG, 18, 64, 83, 108, 134
LDP, *see* linear dishonest parsability constraint
Lexical Functional Grammar, 39, 109
lexicon
AVG, 20
CG, 52
CUG, 54
FG, 137, 141, 142
FUG, 70, 71
HPSG, 93, 96
LFG, 119, 120
LFG, *see* Lexical Functional Grammar
license, 20, 21
linear bounded automaton, 28, 64, 83, 108, 135
linear dishonest parsability constraint, 29

meaning postulate, 142, 146

nondeterministic linear exponential time, *see* complexity class, *NEXP*
nondeterministic polynomial time, *see* complexity class, *NP*

off-line parsability constraint, 18, 28, 63, 82, 107, 134
off-line view, 112
OLP, *see* off-line parsability constraint
on-line view, 112
1-open predication, 146
operator, 154
oracle, 139
order specification problem, 174, 176, 179
order specification process, 137, 163, 173, 174
order specification rule, 173–175
ordinary form specification rule, 165, 166
OSP, *see* order specification problem

P-AVGL, *see* language, HP-AVG
parsability constraint, *see* HPC/ LDP/ OLP
path, 19, 44, 45
path-equation, 17, 19, 44, 110
polynomial time, 40, 139
predicate, 145, 146, 155
basic, 137, 141, 142
derived, 137, 141, 143
principle, 86
language specific, 87, 93, 95
universal, 87, 93, 95
produce
CG, 53
CUG, 55
FUG, 75
HPSG, 99
LFG, 126
production process, 89, 110, 112
pronoun, 145
proper name, 145
property

clause structure, 157
predicate, 165
satellite, 157
term, 165

R-AVG, *see* attribute-value grammar, restricted
R-AVGL, *see* language, R-AVG
recognition problem, 12, 65, 78, 85, 103, 109, 129
AVG, 18
fixed, 39, 41
CUG, 57, 61, 62
FUG, 79, 81, 82
HPSG, 104, 107
LFG, 130, 133
in *NP*, 29
universal, 39, 41
CUG, 61, 62
FUG, 79, 81
HPSG, 104, 107
LFG, 131, 133
recursively presentable
grammars, 20, 27
sets, 20, 28
reduction
polynomial time, many-one, 39, 40, 79, 104, 129, 139, 140
reentrance, 44, 110
reentrant, 44

SAT, *see* SATISFIABILITY
satellite, 154, 155
SATISFIABILITY, 22
schema, 86
selection restriction, 142, 146, 148, 161
simulation, 69, 71, 91, 95, 115, 120
start symbol
AVG, 20
R-AVG, 20
subcategory, 53
subcategory-property, 53, 56
subsumption, 42, 46
syntactic rule
AVG, 20
R-AVG, 20

template, 173–175
template-filler, 165, 174
term, 145, 155
basic, 137, 141, 142, 145
derived, 137, 141, 143, 145
term operator, 142
terminal AVM, 99
terms
basic, 148
derived, 148
3-conjunctive normal form, 41, 140
3-SATISFIABILITY, 39, 41, 140, 169, 176
3SAT, *see* 3-SATISFIABILITY
tractable, 12, 40, 138
trigger operator, 166

undecidable, 138, 149
underlying representation, 137, 153
unification, 42, 46, 47
URP, *see* recognition problem, universal

value, 43, 45

weak generative capacity, 39
CUG, 63, 64
FUG, 82, 83
HPSG, 107, 108
LFG, 133, 134

yield
FUG, 75
R-AVG, 19

Samenvatting

Computationele valkuilen in handelbare grammaticale formalismen

In dit proefschrift passen we complexiteitstheorie toe op grammaticale formalismen. Een aangename eigenschap van complexiteitstheorie is dat complexiteitsanalyses diverse soorten informatie geven. Complexiteitsanalyses kunnen zowel het feit *dat*, als de reden *waarom*, een probleem moeilijk is, aantonen. Daarnaast kunnen zij soms aanbevelingen geven *hoe* het probleem makkelijker te maken is.

Complexiteitsanalyses van grammaticale formalismen zijn op twee manieren van dienst. Enerzijds helpen de analyses bij de theorievorming van de grammaticale formalismen. Anderzijds doen de analyses algemene uitspraken over mogelijke implementaties van de grammaticale formalismen. In deze thesis ligt de nadruk op de theorievorming.

Aan de hulp die complexiteitsanalyses kunnen geven, ligt de volgende redenering ten grondslag. De manier waarop mensen natuurlijke taal verwerken suggereert dat de structuur van natuurlijke taal genoeg informatie bevat om efficiënte verwerking mogelijk te maken. Deze efficiënte verwerking impliceert dat het herkennen van een rij woorden als een zin (of als geen zin) een *handelbaar* probleem is. Voor nauwgezette formalismen van natuurlijke taal moet het herkennen van een rij woorden daarom ook handelbaar zijn. Dientengevolge zouden de formalismen die we in dit proefschrift bestuderen, bijna per definitie, *handelbare grammaticale formalismen* moeten zijn.

Het resultaat van complexiteitsanalyses moet echter op een intelligente wijze geïnterpreteerd worden. Een formalisme dat volgens de analyses *onhandelbaar* is, is niet direct een verkeerde formalisme. De onhandelbaarheid van het formalisme geeft eerder aan dat (en waar) het formalisme structuren die onvoldoende informatie bevatten, toekent aan taal uitingen. Op deze wijze helpen complexiteitsanalyses de theorievorming van grammaticale formalismen. We zijn er echter stellig van overtuigd dat de theorieën de onvolmaaktheden verhelpen kunnen die door de analyses worden blootlegt. Vanuit dat oogpunt kunnen we de onvolmaaktheden van de theorieën zien als *computationele valkuilen.*

De algemene uitspraken die complexiteitsanalyses over implementaties doen, zijn van de volgende aard: Het bewijs dat een formalisme onhandelbaar is, betekent dat er geen efficiënte algoritmen bestaan voor algemeen bruikbare implementaties van het formalisme. Desalniettemin kan een efficiënte algoritme bestaan voor één bepaalde grammatica. Deze algoritme heeft dan op een vernuftige manier gebruik gemaakt van enkele speciale eigenschappen van de specifieke grammatica. De onhandelbaarheid van het formalisme geeft echter aan dat deze algoritme niet om te vormen is tot een efficiënte algoritme voor andere grammatica's. Is men dus op zoek naar een efficiënte algoritme voor een andere grammatica, dan zal men geheel opnieuw onderzoek moeten verrichten naar eventuele specifieke eigenschappen van de betreffende grammatica.

We beëindigen deze korte samenvatting met een beknopt overzicht van de inhoud van dit proefschrift. Hoofdstuk 1 geeft de motivatie voor onze aanpak van complexiteitsanalyses. Deze aanpak is gebaseerd op de beschrijvingen van grammaticale formalismen zoals deze in de praktijk voorkomen. In Hoofdstuk 2 bestuderen we "restricted attribute-value grammars". We introduceren een versoepeling van de "off-line parsability constraint": de "honest parsability constraint." We bewijzen vervolgens dat de "restricted attribute-value grammars" onder deze versoepeling precies alle verzamelingen in de complexiteitsklasse *NP* genereren. Dit geeft ons een koppeling tussen complexiteit en zwak generatieve kracht. In Hoofdstuk 3 definiëren we "Categorial Unification Grammar" (CUG). We tonen aan dat het herkenningsprobleem van rijen woorden voor CUG *NP*-volledig (onhandelbaar) is. Met deze complexiteitsmaat kunnen we de zwak generatieve kracht van CUG vaststellen. We tonen in Hoofdstuk 4, 5 en 6 aan dat CUG gesimuleerd kan worden met respectievelijk "Functional Unification Grammar" (FUG), "Head-driven Phrase Structure Grammar" (HPSG) en "Lexical Functional Grammar" (LFG) . We bewijzen verder dat de herkenningsproblemen voor FUG, HPSG en LFG ook *NP*-volledig (onhandelbaar) zijn. Deze complexiteitsmaat stelt ons wederom in staat de zwak generatieve kracht vast te stellen. Hoofdstuk 7 behandelt "Functional Grammar" (FG). Bij het generatieproces in FG worden doorgaans vier processen onderscheiden. We formuleren ieder van de vier processen als een afzonderlijk beslissingsprobleem. Van ieder van deze beslissingsproblemen analyseren we de complexiteit. In het laatste hoofdstuk (Hoofdstuk 8) formuleren we onze algemene conclusies en geven we richtingen aan voor verder onderzoek.

Titles in the ILLC Dissertation Series:

Transsentential Meditations; Ups and downs in dynamic semantics
Paul Dekker
ILLC Dissertation series 1993-1

Resource Bounded Reductions
Harry Buhrman
ILLC Dissertation series 1993-2

Efficient Metamathematics
Rineke Verbrugge
ILLC Dissertation series 1993-3

Extending Modal Logic
Maarten de Rijke
ILLC Dissertation series 1993-4

Studied Flexibility
Herman Hendriks
ILLC Dissertation series 1993-5

Aspects of Algorithms and Complexity
John Tromp
ILLC Dissertation series 1993-6

The Noble Art of Linear Decorating
Harold Schellinx
ILLC Dissertation series 1994-1

Generating Uniform User-Interfaces for Interactive Programming Environments
Jan Willem Cornelis Koorn
ILLC Dissertation series 1994-2

Process Theory and Equation Solving
Nicoline Johanna Drost
ILLC Dissertation series 1994-3

Calculi for Constructive Communication, a Study of the Dynamics of Partial States
Jan Jaspars
ILLC Dissertation series 1994-4

Executable Language Definitions, Case Studies and Origin Tracking Techniques
Arie van Deursen
ILLC Dissertation series 1994-5

Chapters on Bounded Arithmetic & on Provability Logic
Domenico Zambella
ILLC Dissertation series 1994-6

Adventures in Diagonalizable Algebras
V. Yu. Shavrukov
ILLC Dissertation series 1994-7

Learnable Classes of Categorial Grammars
Makoto Kanazawa
ILLC Dissertation series 1994-8

Clocks, Trees and Stars in Process Theory
Wan Fokkink
ILLC Dissertation series 1994-9

Logics for Agents with Bounded Rationality
Zhisheng Huang
ILLC Dissertation series 1994-10

On Modular Algebraic Protocol Specification
Jacob Brunekreef
ILLC Dissertation series 1995-1

Investigating Bounded Contraction
Andreja Prijatelj
ILLC Dissertation series 1995-2

Algebraic Relativization and Arrow Logic
Maarten Marx
ILLC Dissertation series 1995-3

Study on the Formal Semantics of Pictures
Dejuan Wang
ILLC Dissertation series 1995-4

Generation of Program Analysis Tools
Frank Tip
ILLC Dissertation series 1995-5

Verification Techniques for Elementary Data Types and Retransmission Protocols
Jos van Wamel
ILLC Dissertation series 1995-6

Transformation and Analysis of (Constraint) Logic Programs
Sandro Etalle
ILLC Dissertation series 1995-7

Frames and Labels. A Modal Analysis of Categorial Inference
Natasha Kurtonina
ILLC Dissertation series 1995-8

Tools for PSF
G.J. Veltink
ILLC Dissertation series 1995-9

(to be announced)
Giovanna Cepparello
ILLC Dissertation series 1995-10

Instantial Logic. An Investigation into Reasoning with Instances
W.P.M. Meyer Viol
ILLC Dissertation series 1995-11

Taming Logics
Szabolcs Mikulás
ILLC Dissertation series 1995-12

Metalogics for Logic Programming
Marianne Kalsbeek
ILLC Dissertation series 1995-13

Enriching Linguistics with Statistics: Performance Models of Natural Language
Rens Bod
ILLC Dissertation series 1995-14

Computational Pitfalls in Tractable Grammatical Formalisms
Marten Trautwein
ILLC Dissertation series 1995-15

The Solution Sets of Local Search Problems
Sophie Fischer
ILLC Dissertation series 1995-16