Contents

0 Introduction **1**
0.1 Motivation 1
0.2 Goals of the Thesis 3
0.3 Overview of the Thesis 4

1 Linguistic Deduction **7**
1.1 Formalism 9
1.1.1 Constraint Language 10
1.1.2 Definite Clauses 14
1.2 Linguistic Deduction Algorithms 16
1.2.1 Relationship between NLP and Logic Programming 16
1.2.2 Alternatives to Prolog's Search Strategy 18
1.2.3 Direction of Processing 19
1.2.4 Selection Function 24
1.2.5 Memoing 25
1.2.6 Constraints in Linguistic Deduction 35
1.2.7 Search Strategy 39
1.2.8 Bidirectional NLP Algorithms 40
1.3 Conclusion 41

2 From Principle-Based Grammars to Rule-Based Grammars **43**
2.1 Principle-Based Versus Rule-Based Grammars 43
2.2 Partial Deduction 45
2.2.1 Partial Deduction Example: DCG 46
2.2.2 Partial Deduction Applied to GB 50
2.3 HPSG as a Principle-Based Grammar 56
2.3.1 Formalisations of HPSG 56
2.3.2 Principles of HPSG 61
2.3.3 Conclusion on HPSG principles 70
2.4 Partial Deduction Applied to HPSG 71
2.4.1 PD Experiment 1: Binary Branching HPSG 72

2.4.2 PD Experiment 2: COMP-DTRS as a List-Valued Feature . . 76
2.5 Lexical Rule Expansion as Partial Deduction 80
2.6 Conclusion 81

3 Bottom-Up Earley Deduction 85
3.1 The Algorithm 86
3.1.1 Lookup (Scanning) 87
3.1.2 Indexing 89
3.1.3 Best-First Search 92
3.1.4 Goal Types 94
3.2 Earley Deduction for Discontinuous Constituency 95
3.2.1 Johnson's Combine Operator 96
3.2.2 Head-Wrapping 97
3.2.3 Sequence Union 98
3.2.4 Necessity for Tabulation 99
3.2.5 Inadequacy of Top-Down Earley Deduction 100
3.2.6 Guides versus Indexing 100
3.3 Application to Generation 103
3.3.1 Semantically Monotonic Grammars 103
3.3.2 Semantically Non-Monotonic Grammars 106
3.3.3 Conclusion on Generation 108
3.4 Properties of the Algorithm 109
3.4.1 Correctness 109
3.4.2 Completeness 110
3.4.3 Complexity and Termination 113
3.5 Incrementality 115
3.5.1 Motivation 115
3.5.2 Left-Right Incrementality 116
3.5.3 "Full" Incrementality 117
3.5.4 Incremental Addition of Non-Unit Clauses 123
3.6 Conclusion 123

4 Preference-Driven Linguistic Deduction 125
4.1 Preferences and Best-First Processing 125
4.1.1 Models of Preference for Constraint-Based Grammars . . . 128
4.1.2 A Model of Preference 133
4.1.3 Preferences and best-first processing 137
4.1.4 Word Order 139
4.1.5 Application to Disambiguation 148
4.1.6 Application to Generation 151
4.1.7 Determination of Preference Values 153
4.2 Conclusion 153

5 Implementation **155**
5.1 Sorted Feature Terms: ProFIT 158
5.1.1 The ProFIT Language 158
5.1.2 From ProFIT Terms to Prolog Terms 165
5.1.3 ProFIT Implementation 166
5.2 Extensions of the Constraint Language 168
5.2.1 Set Descriptions and Set Constraints 168
5.2.2 Guarded Constraints 168
5.2.3 Linear Precedence Constraints 170
5.2.4 Interaction between Constraint Handling and Tabulation . 171
5.3 The Generalised Linguistic Deduction (GeLD) System 172
5.3.1 Control Information 172
5.3.2 Coroutining 177
5.3.3 Preference Values 177
5.3.4 Compilation 178
5.3.5 Partial Deduction 179
5.4 The Deduction Engine 181
5.4.1 Top-down processing 182
5.4.2 Head-Driven Processing 183
5.4.3 Chart-Based Algorithms 183
5.4.4 Prolog Call 184
5.5 Best-First Search 184
5.6 Performance of the System 186
5.7 Conclusion 186
5.8 Availability 187

6 Conclusion and Future Research **189**
6.1 Summary 189
6.2 Future Research 192

Appendix **193**

A Programs with Control Information **195**
A.1 Partial deduction for DCG 195
A.2 Partial deduction for GB 196

B Performance of the Algorithms **199**
B.1 Compilation to Prolog Terms 199
B.2 Partial Deduction 202
B.3 Bottom-Up Earley Deduction 204

C Prolog Code of the Deduction Algorithms **207**
C.1 Prolog goals 207
C.2 Top-down goals 207
C.3 Head-driven Processing 209
C.4 Top-down Earley Deduction 210
C.5 Bottom-Up Earley Deduction 210
C.6 Shared Code for Bottom-Up and Top-Down Earley Deduction . . . 211
C.7 Handling of Items 212

D GeLD Interface Specification **215**
D.1 Proving Goals 215
D.2 Loading Programs 215
D.3 Inspection of Clauses and Items 216

Bibliography **216**

Acknowledgements

I am very grateful to my supervisors Hans Uszkoreit and Henry Thompson, who worked through several versions of this thesis, and provided fruitful criticism, discussion and support.

Special thanks to Suresh Manandhar, who read a draft of this thesis and provided useful comments, taught me a lot about constraints, and worked out the formal foundations and implemented the constraint solvers for the extended constraint language used in this thesis (set constraints, LP constraints and guarded constraints).

Four anonymous reviewers of the *Journal of Logic Programming* provided very detailed and constructive criticism of some of the work presented here; their comments were very helpful in pointing this work in the right direction.

During the work on this thesis, I was financially supported by

- German Research Center for Artificial Intelligence (DFKI)
- Cray Systems, Luxembourg through a consulting contract
- Deutsche Forschungsgemeinschaft (DFG) through project N3 *Bidirectional Linguistic Deduction* (BiLD) in the Special Research Division (Sonderforschungsbereich 314) “Artificial Intelligence - Knowledge-Based Systems.”
- The Commission of the European Communities through project LRE-61-061 *Reusable Grammatical Resources* in the programme “Linguistic Research and Engineering” (LRE).
- Universität des Saarlandes, FR 8.7 Computerlinguistik
- IBM Deutschland GmbH through the project LILOG

I benefitted very much from exchanges with the following research centres:

- The Human Communication Research Centre at the University of Edinburgh, where I spent six weeks in 1994 and three weeks in 1995.
- The Austrian Research Institute for Artificial Intelligence where I spent one month in the summer of 1995.

I would like to thank the people who helped me with this thesis by providing ideas, criticism, inspiration and support, and β-testing the software: Tania Avgustinova, Sergio Balari, Thorsten Brants, Christian Braun, Chris Brew, Bob Carpenter, Jo Calder, Matt Crocker, Luca Dini, Jochen Dörre, Hannes Fischer, Ralph Flassig, Dale Gerdemann, James Hannigan, Corinna Johanns, Reinhard Karger, Martin Kay, Tibor Kiss, Tatjana Klajic, Andrea Kowalski, Brigitte Krenn,

Uli Krieger, Holger Maier, Suresh Manandhar, Johannes Matiasek, Detmar Meurers, Sebastian Millies, Drew Moshier, Klaus Netter, Günter Neumann, Gertjan van Noord, Stephan Oepen, Karel Oliva, Hannes Pirker, Gerrit Rentier, Herbert Ruessink, Ivan Sag, Christer Samuelsson, Werner Saurer, Wojciech Skut, Craig Thiersch, Harald Trost, Mats Wirén.

Some results presented in this thesis have been published before. The description of the extension of the feature constraint language is adapted from the paper *Extending Unification Formalisms* [Erbach *et al.*, 1995b]. Section 3.1 is based on my COLING-94 paper *Bottom-Up Earley Deduction* [Erbach, 1994a], chapter 4 makes use of material from my paper *Preference Values in Typed Feature Structures* [Erbach, 1993b] and my CLAUS report *Towards a Theory of Degrees of Grammaticality* [Erbach, 1993a], section 5.1 is based on my EACL-95 paper *ProFIT: Prolog with Features, Inheritance and Templates* [Erbach, 1994c].

I benefitted from the comments I received when presenting parts of this work at these conferences, and also from presentations to the following audiences: LILOG project workshop (Bad Herrenalb), Polish Academy of Sciences (Warsaw), IBM T.J. Watson Research Center (Yorktown Heights), workshop "Coping with Ambiguity in Typed Feature Formalisms" (ECAI, Vienna), Human Communication Research Centre (University of Edinburgh), International Conference on Mathematical Linguistics (Tarragona), HPSG workshop (Columbus, OH), Spring School in Language and Logic (Tbilisi), KONVENS (Vienna), ALEP User Group meeting (Luxembourg), RGR project workshop (South Queensferry), Swedish Institute for Computer Science (Kista), workshop "Constraint-Based Formalisms and Grammar Writing" (LLI Summer School, Barcelona), the Tbilisi Symposium in Language, Logic and Computation, and various seminars in Saarbrücken.

Chapter 0

Introduction

0.1 Motivation

This thesis is concerned with processing models for declarative grammar formalisms. Compared to their procedural predecessors, in which there was hardly a separation between a grammar and the programs that were used to analyse and generate sentences, declarative grammar formalisms enjoy the following advantages:

- A declarative formalism can be given a precise formal semantics, which makes it possible to give formal proofs of properties of the formalism.
- The processing (analysis or generation) result does not depend on the order of processing. This allows optimisations of the processing algorithm or the use of compilation techniques.
- Bidirectionality (use of the same grammar for analysis and generation) becomes possible; it is not necessary to write separate grammars and programs for analysis and generation.

A declarative grammar is a set of statements (expressed in a logical language) about a (natural) language.

The logical language used to express declarative grammars is referred to as a *grammar formalism*, and the set of statements expressed in such a formalism as a *grammatical theory.* In some systems, the formalism allows only certain types of statements to be expressed, so that the formalism constrains the form of grammatical theories that can be expressed in it.[1]

[1] The relationship between grammar formalism, grammatical theory and grammar of a language is discussed in [Pereira and Shieber, 1984; Shieber, 1988a].

Various declarative grammatical theories have been developed in the past decade and a half, the most prominent being

- Functional Unification Grammar (FUG) [Kay, 1984],
- Definite Clause Grammar (DCG) [Pereira and Warren, 1980],
- Lexical-Functional Grammar (LFG) [Kaplan and Bresnan, 1982],
- Generalized Phrase Structure Grammar (GPSG) [Gazdar *et al.*, 1985],
- Head-Driven Phrase Structure Grammar (HPSG) [Pollard and Sag, 1987; Pollard and Sag, 1994],
- Categorial Grammar in its various forms [Ajdukiewicz, 1935; Oehrle *et al.*, 1988],
- Tree Adjoining Grammar (TAG) [Joshi *et al.*, 1975; Joshi and Vijay-Shanker, 1985],
- Government and Binding Theory (GB) [Chomsky, 1981].

We will concentrate on Definite Clause Grammars and HPSG in the following, as they are very clear examples of linguistic theories expressed in very general formalisms, without blurring the distinction between formalism and theory.[2]

The inception of HPSG marks a turning point in the field of computational linguistics. Prior to HPSG, computational models of natural language were all rule-based; there were grammar rules that described syntactic constructions and the constraints associated with them. Prior formalisms such as GPSG are explicitly based on context-free grammar rules, over which generalisations could be expressed by means of meta-rules and feature instantiation principles. Government and Binding theory is principle-based, but not really intended as a formalism for *computational* linguistics.

It was HPSG that finally brought principle-based grammars into natural language processing. Ever since, there have been two "camps" in the area of syntactic processing.

On the one hand, there are those who build on previous work in parsing technology and extend this to unification-based and constraint-based formalisms, but rely on rule-based grammars, and often on a "context-free backbone," i.e., concatenation as the only operation for the combination of constituents. Their approach to grammar development is to write rules that do respect the principles of HPSG.

On the other hand, there are those who reject the writing of rules and base inference directly on the principles of the grammar. In this tradition, principles

[2]Note that DCG does not make any claims about natural language, but is a general string processing formalism, whereas HPSG does make claims about natural languages by formulating principles, rule schemata etc. in a general-purpose constraint language.

are generally stated as implications on types, and processing algorithms are based on type inference. These approaches are well suited for grammars that do not rely on a context-free backbone, but unfortunately they are quite inefficient in general.

The motivation of the work described here is to reconcile these two camps by showing that the gap between principle-based and rule-based grammars is not as wide as it is sometimes claimed to be; in fact, there is a continuous spectrum between purely rule-based approaches to grammar such as context-free grammars and principle-based approaches such as Government-Binding theory. We will bridge the gap by showing that principle-based grammars can be transformed into rule-based grammars by means of partial deduction techniques that are well-known from logic programming.

Once this is done, many of the useful techniques developed for rule-based grammars with a context-free backbone (especially in the area of chart parsing) can be generalised to principle-based grammars.

A context-free backbone, even though convenient for many parsing algorithms, may be too strong a restriction for the adequate description of many languages. Bottom-up algorithms (or algorithms that combine bottom-up structure building with top-down prediction such as the head-corner algorithms) have proved suitable for grammars without a context-free backbone. This is why a bottom-up chart-based approach will be pursued and improved upon in this thesis.

Processing HPSG and other principle-based grammars constitutes the premier motivation for this thesis. Additional motivation comes from linguistic engineering — the quest for systems that process natural language to perform useful tasks such as querying a database or acquiring knowledge from texts. Because of its inherent incrementality, Earley deduction is a favourable computation model for linguistic engineering tasks.

Obviously, incremental processing of spoken or typed input is a benefit of this algorithm. A bit less obvious is the fact that an augmentation enables a chart-based algorithm to handle destructive changes in the input (e.g., deletion or change of words in a word processor) quite naturally.

Moreover, the incrementality can be exploited for best-first search, since less preferred computation steps can be delayed, and — if necessary — be performed later incrementally. Processing ill-formed input is an application for which best-first search is crucial, since the less restrictive grammar rules, which allow for ill-formed structures, would otherwise lead to a combinatory explosion of the search space.

0.2 Goals of the Thesis

The goal of the thesis is to present a model of syntactic processing that

- is applicable to principle-based grammars,

- can handle string operations beyond concatenation,
- is usable for parsing and generation (bidirectional),
- is efficient enough to be useful for applications,
- supports incremental processing,
- allows the exploitation of preference values for best-first processing.

Our approach divides this goal into two subgoals: first we tackle the problem of principle-based grammars by showing that they can be transformed into rule-based grammars, and secondly, we develop a processing algorithm for the resulting rule-based grammar, namely a bottom-up version of Earley deduction, which allows us to integrate work done in bottom-up chart parsing with work from the area of NLP as deduction.

The crucial step in bottom-up processing is to select the initial clauses from which the processing starts, e.g., lexical lookup. Unless this selection of clauses is suitably restricted, a lot of useless computation is performed which cannot contribute to a solution for a given goal.

Our processing model extends the indexing scheme for chart parsers (which encodes to the start and end positions of continuous constituents) to discontinuous constituents, and to semantic indexing for generation.

We show that our algorithm is usable for incremental NLP in which the input becomes successively further instantiated. Following work on “full incrementality” by Wirén in a chart parsing framework, we can augment our algorithm to keep track of dependencies between lemmata in order to handle destructive changes of the input.

The algorithm supports best-first processing. In order to make this notion of “goodness” explicit, we augment the clauses of the grammar with preference values which can be interpreted as probabilities. We fix a control strategy which guarantees that the best solution is always enumerated first.

0.3 Overview of the Thesis

Chapter 1 presents the theoretical and formal foundations and establishes the terminology and formalism to be used throughout the thesis. A feature constraint language with sorts is introduced and extended with set constraints, linear precedence constraints and guarded constraints. We review linguistic deduction algorithms in the tradition of logic grammars (DCG) and constraint-based grammars, and discuss the dimensions in which these algorithms can be varied (direction of processing, selection function, memoing, constraint handling, search strategy).

Chapter 2 reviews the difference between principle-based and rule-based grammars, and presents an algorithm for transforming the former into the latter.

Principle-based grammars capture linguistic generalisations, but are not ideally suited for efficient processing because of the amount of computation needed for computing the structures described by the principles. Rule-based grammars, on the other hand, describe these structures directly. In this chapter, we show how this computation can be performed at the time when a grammar is compiled by making use of partial deduction. As a result, a principle-based grammar is transformed into a rule-based one for which efficient processing algorithms exist and are further developed in chapter 3. As an illustration of the technique, partial deduction is applied to turn a small principle-based GB fragment into a rule-based grammar.

The larger part of the chapter is concerned with an application of partial deduction to HPSG. We contrast our formalisation of HPSG as definite clauses with alternative formalisations which make use of type constraints, show the equivalence of the formalisations, and argue that ours is preferable because it allows the straightforward application of (Constraint) Logic Programming techniques, and stands to benefit directly from progress in logic programming.

Chapter 3 presents Bottom-Up Earley Deduction,[3] a tabular bottom-up deduction algorithm that is related to bottom-up chart parsers in the same way that (top-down) Earley Deduction is related to Earley's context-free parsing algorithm or to top-down chart parsers. We argue that a bottom-up algorithm is advantageous for lexicalised grammars, for discontinuous constituency, for incremental processing, and for best-first search.

We discuss handling discontinuous constituency and string operations other than concatenation. We review previously proposed algorithms for handling grammars which employ such operations, especially head-driven parsing and generation, and show how the new algorithm can be used for such grammars.

We discuss the use of the bottom-up Earley deduction algorithm for generation, show how it is applicable to semantically non-monotonic grammars, and compare it with other algorithms that can be used bidirectionally for both parsing and generation.

The properties of the new algorithm (correctness, completeness, complexity and termination) are discussed.

We show that the algorithm enjoys a property of incrementality which makes it very useful for practical NLP systems. In this context, incrementality means that addition of a new lemma (item) to the set of initial and derived lemmata causes all computation relevant to this new lemma to be performed by combining it with the old lemmata.

The incrementality property can of course be used for the classic problem of

[3] Bottom-Up Earley Deduction may be seen as a self-contradictory name if it is believed that the top-down direction is one of the defining characteristics of Earley Deduction. We do not think so. Quite to the contrary, we find that the flexibility to be usable in the bottom-up and the top-down direction, and with different search strategies (depth-, breadth-, best-first) is a major advantage of the Earley Deduction approach.

incremental NLP: processing a portion of the input as soon as it is received. We also consider the case where the input is destructively changed, and present an algorithm for updating the chart in case of destructive changes.

In Chapter 4, the incremental nature of the Bottom-Up Earley deduction algorithm is exploited for best-first search by delaying the addition of less promising lemmata. This is possible because in an incremental algorithm it does not matter at what point in the overall computation a lemma is added. In order to formalise how good or promising a lemma is, we introduce the notion of a preference value. Preference values can be regarded as probabilities, and we discuss how preference values for grammars could be obtained.

The partial deduction system, and the deduction algorithms described in the thesis, along with a typed feature structure system, have been implemented in Prolog as a flexible, yet efficient framework for experimenting with linguistic deduction strategies. The implementation is described in chapter 5.

The final chapter 6 summarises the results presented in the thesis and outlines directions for future work.

Chapter 1

Linguistic Deduction

This chapter covers the formal and theoretical foundations on which the work described in this thesis is built, in particular the logic grammar framework, and parsing and generation algorithms for logic grammars. The reader is expected to have a working knowledge of formal language theory, first-order logic, (constraint) logic programming and logic grammars, sorted feature structures, and unification-based grammar formalisms.[1]

We view linguistic processing (generation and parsing) as a process of deduction, i.e., proving theorems about signs of a natural language. This view allows us to make use of results in theorem proving, and in particular in the field of logic programming, for the design of NLP algorithms. The theories about which we want to prove theorems are natural language grammars: sets of statements about NL signs expressed in a suitable logical language.

The logical language (or grammar formalism) that will be used is definite clauses together with a constraint language. Our grammatical formalism shares many properties with the "powerful grammar formalism" of van Noord [van Noord, 1993] which is defined as in instance of Höhfeld and Smolka's Constraint Logic Programming schema [Höhfeld and Smolka, 1988]. The difference is that we use a more powerful constraint language (see section 1.1).

General-purpose theorem proving methods are not necessarily applicable for useful NLP applications because they may be too inefficient. Therefore we investigate specialised instances of general algorithms that are well adapted to the

[1]The following works provide excellent introductions to these areas. **Formal Language Theory:** [Hopcroft and Ullman, 1979; Partee *et al.*, 1990]; **First Order Logic:** [Partee *et al.*, 1990; Fitting, 1990]; **(Constraint) Logic Programming:** [Lloyd, 1984; Robinson, 1992; Sterling and Shapiro, 1986; O'Keefe, 1990; Höhfeld and Smolka, 1988]; **Logic Grammars:** [Pereira and Shieber, 1987; Shieber *et al.*, 1994]; **(Sorted) Feature Structures** [Johnson, 1988; Smolka, 1992; Uszkoreit, 1988; Carpenter, 1992] **Unification-based grammar formalisms:** [Shieber, 1986; Sells, 1985; Abeillé, 1993; Pollard and Sag, 1994]

structure of the linguistic theories about which we want to prove theorems. In particular, we will see that most well-known parsing and generation algorithms for natural language can be regarded as instances of deduction algorithms.

We adopt a sign-based view of natural language. Under this perspective, a language is not seen as a set of strings (as it would be in formal language theory), but as a set of meaning-string associations. This view is crucial since we are interested not only in determining whether or not a string is in the language, but primarily in determining which string can be used to express a given meaning (generation) and vice versa which meaning if conveyed by a given string (parsing).

We take a grammar of a natural language to be a recursively defined relation between strings and their meanings, with the elements of the relation being linguistic signs. A sign has features (or attributes) for the string with which it is expressed (in Saussure's terms: the *signifiant*) and for its meaning potential (the *signifié*). The signifiant is in spoken language a phonetic form[2] and in written language a sequence of characters; and the signifié is generally represented as a logical form in formal linguistics, i.e., as a formula of first-order logic, property theory, or situation theory, or a discourse representation structure of DRT, λ-DRT or whatever logical framework is considered adequate for the treatment of natural language semantics.

Linguistic deduction involves proving that a given underspecified structure is indeed a sign of the language defined by the grammar. In general, there are many ways in which a sign can be underspecified. The prototypical ways in which a sign will be underspecified in NLP are the following.

1. The string feature is instantiated, but not the logical form feature. This case is generally called *analysis* or *parsing*.

2. The logical form feature is instantiated, but not the feature for the string with which it is expressed. This case is called *synthesis* or *generation*.[3]

The following queries (in Prolog notation) correspond to generation and parsing, respectively. We take `sign/2` to be a two-place relation between strings (represented as lists) and logical forms (represented as first-order terms).

If the same algorithm can be used for both parsing and generation, we speak of *bidirectional linguistic deduction*.

There are other linguistic processing tasks, when (parts of) the grammar and/or the lexicon are unknown, which involve inferring (clauses of) the program from examples. Since these are not examples of deduction, but rather abduction or induction, we shall not discuss them in this thesis.

[2] For reasons of simplicity, we represent phonetic form by its graphemic representation in written language.

[3] Natural language generation is generally seen to consist of two stages: a strategic (what-to-say, discourse planning) and a tactical (how-to-say) component. When we speak of generation in this thesis, we refer to the latter.

```
generation:   ?- sign(STRING,call_up(john,friends)).
parsing:      ?- sign(⟨john,calls,friends,up⟩,LF).
```

Figure 1.1: Parsing and generation as linguistic deduction

In the following section, we present the grammar formalism that we use to define linguistic theories, and turn to linguistic deduction algorithms afterwards.

1.1 Formalism

In this thesis, we will use a highly expressive grammar formalism, extending the formalism defined by van Noord [van Noord, 1993, p. 27]. Van Noord's formalism has the following properties.

- The formalism consists of definite clauses, as in Prolog; instead of first-order *terms,* the data structures of the formalism are *feature structures.*

- The formalism does not assume that concatenation is the sole string-combining operation (in contrast to FUG, DCG, PATR II, LFG, GPSG and UCG).

- The formalism is defined in an abstract framework, which facilitates the extendibility of the techniques [developed in his thesis] to other (more powerful) constraint languages.

The formalism we use here differs in two respects that have to do with our interest in processing principle-based grammars (such as HPSG).

1. We do use a more powerful constraint language, which includes *sorted* feature terms, finite domains, and set descriptions and set constraints, linear precedence constraints and guarded constraints in addition to first-order terms. Sorted terms and set descriptions are included because they are key ingredients of HPSG grammars, and finite domains are needed because because they allow a reduction of otherwise unmotivated non-determinism.

2. We allow relational dependencies between arbitrary feature values, not just as relations between strings like van Noord's string operations beyond concatenation (cf. section 1.1.2).

1.1.1 Constraint Language

In this section, we introduce the constraint language which we will use throughout this thesis.[4]

1.1.1.1 Sorted Feature Terms

Our conception of sorted feature terms is based on Carpenter's *Logic of Typed Feature Structures*[5] from which it differs only in one respect: in our system, the sort hierarchy is not required to be a bounded complete partial order. As a consequence, a feature structure can be described by two or more different sorts which do not have a common subsort; and two different sorts can have several common subsorts, even though they do not have a unique most general common subsort. In this respect, our constraint language is similar to the CUF formalism [Dörre and Dorna, 1993]. More details about our sorted feature term language can be found in [Erbach, 1994b; Erbach, 1994c] and in section 5.1.

We follow the distinction between *feature terms* as syntactic objects and *feature structures* which are the abstract mathematical objects described by the feature terms. The feature term language consists of five different kinds of sorted feature terms (SFT):

- A constraint that the described feature structure must be subsumed (symbolised by the "smaller than" sign `<`) by a given sort (`<Sort`); e.g. the structure described by the term `<phrasal_sign` must be subsumed by the sort *phrasal_sign*.
- A feature constraint (`F!SFT`), which constrains the value of the feature `F` of the described feature structure to be subsumed by the structure described by `SFT`. For example, the feature constraint `subcat! <elist` denotes a structure whose SUBCAT-value is an empty list.
- A Prolog term. Besides sort constraints, Prolog terms are the only other way to "bottom out". Prolog terms can be either variables, atoms, or compound terms. This arrangement makes it possible to take advantage of the

[4] This constraint language was developed in the project "The Reusability of Grammatical Resources", and is the joint work of Suresh Manandhar, Wojciech Skut and the author. It is described more fully in [Manandhar, 1994; Manandhar, 1995; Erbach *et al.*, 1994a; Erbach *et al.*, 1994b; Erbach *et al.*, 1995b].

[5] We prefer to speak of *sorted feature structures* since the usage of the term *type* in logic programming and computational linguistics is ambiguous: For Carpenter, a type denotes a subset of the domain (for example the type *sign*, for which the features PHON, SYNSEM, QSTORE and RETRIEVED are appropriate); for other authors like Emele and Zajac [Emele and Zajac, 1990; Zajac, 1992], a type can be a complex recursively defined relation (e.g., the type *append/2* or *sign_of_english/1*). To avoid terminological confusion, we follow the usage established in CLP (cf. [Dörre and Seiffert, 1991; Aït-Kaci and Podelski, 1991]) and the recent HPSG literature [Pollard and Sag, 1994], and use the term *sort* for symbols that denote a subset of the domain, and *relation* or *predicate* for defined relations.

```
SFT  := <Sort              Term of a sort Sort
      | Feature!SFT        Feature-Value pair
      | PROLOGTERM         Any Prolog term
      | SFT & SFT          Conjunction of terms
      | SFT or SFT         Disjunction
```

Figure 1.2: Syntax of sorted feature terms

term language (e.g. cyclic terms) and constraint language (e.g. inequality constraints) of the Prolog system underlying the implementation of our constraint language. The use of normal first-order Prolog terms in the feature term constraint language constitutes no problem, since an n-ary Prolog with functor F term can be regarded as a notational convention for a sorted feature structure of the sort F, for which the features ARG$_1$ ... ARG$_n$ are appropriate, and which has no sortal restrictions for the features.[6]

- A conjunction of terms (`SFT1 & SFT2`), which constrains the described structure to be subsumed by both `SFT1` and `SFT2`, i.e., the set of described structures is the intersection of the structures described by `SFT1` and `SFT2`.

- A disjunction of terms (`SFT1 or SFT2`), which constrains the described structure to be subsumed by either `SFT1` or `SFT2`, i.e., the set of described structures is the union of the structures described by `SFT1` and `SFT2`.

The syntax of the sorted feature term language is summarised in figure 1.2. In later chapters, we will alternatively make use of the more readable notation for sorted feature terms as attribute-value matrices.

1.1.1.2 Finite Domains

Finite domains provide a way of handling certain subclasses of disjunctions in logic programs without the creation of choice points. Finite domains are disjunctions with a finite set of possible values. Finite domains have been introduced for the logic programming language CHIP [van Hentenryck and Dincbas, 1986; van Hentenryck, 1989]; they can also be expressed by more powerful sort inheritance hierarchies in languages such as CUF or LIFE.

A finite domain variable is a variable that can only take on one of a fixed finite (and reasonably small) set of values. A description can constrain the value of the variable to be any subset of this set. When two finite domain variables are unified,

[6] In the implementation, we actually go in the opposite direction, and compile sorted feature terms into a Prolog term representation (cf. section 5.1).

the resulting value's constraint is the intersection of the possible values for both variables. The unification fails if the intersection is empty.

Finite domains are useful for providing efficient processing for many cases of disjunction that arise in NLP without the need for computationally expensive treatments of disjunction, such as distributed disjunctions.[7]

The following is the syntax for defining finite domains:

$$\text{Name fin_dom } [\text{Val}_{1.1}, \ldots, \text{Val}_{1.n}] * \ldots * [\text{Val}_{m.1}, \ldots, \text{Val}_{m.l}]. \quad (1.1)$$

The following example defines a finite domain which contains all possible combinations of the agreement features PERSON, NUMBER and GENDER.

$$\text{agr fin_dom [first,second,third] * [sg,pl] * [masc,fem,neut].} \quad (1.2)$$

The resulting finite domain consists of 3*2*3 = 18 values, all possible combinations of one person, number and gender value. Subsets of this finite domain can be described by making use of the logical connectives $\wedge$ (conjunction), $\vee$ (disjunction), and $\neg$ (negation), e.g., $\neg(\text{third} \wedge \text{sg})$.

1.1.1.3 Inequations

In addition to equality constraints between sorted feature structures, we allow also inequality constraints. Inequality has been introduced in CLP with Prolog II [Colmerauer, 1982; Giannesini *et al.*, 1985], and is supported by all modern logic programming languages.

1.1.1.4 Set Descriptions and Set Constraints

In HPSG [Pollard and Sag, 1994], sets are used for the nonlocal features SLASH, REL and QUE, for quantifier storage QSTORE, for the context feature BACKGROUND, for conjuncts in a coordination structure, for restrictions on semantic indices; and in other HPSG proposals also for features such as SUBCAT. Since we have a strong interest in handling HPSG grammars, which make heavy use of sets, we allow set constraints in our formalism, following Manandhar's attributive logic of set descriptions [Manandhar, 1994]. The set descriptions and set constraints shown in figure 1.3 are allowed in definite clauses.

Disjoint union is not available in the logic, but it can be defined as follows by employing set disjointness and set union operations:

$$x \uplus y =_{def} \quad disjoint(x, y) \sqcap (x \cup y) \quad (1.3)$$

[7](cf. [Eisele and Dörre, 1990; Böttcher, 1993; Maxwell III and Kaplan, 1991; Matiasek, 1993; Trost, 1993])

Set Constraint	Meaning	Syntax for variable X
empty set	X is the empty set	{ }
element	E is an element of X	exist(E)
set description	X contains the elements $E_1 \ldots E_n$ (but they need not be disjoint)	$\{E_1, \ldots, E_n\}$
fixed cardinality set	X contains the disjoint elements $E_1 \ldots E_n$	$\{E_1, \ldots, E_n\}_=$
subset	X is a subset of Y	subset(Y)
union	X is the union of Y and Z	Y ∪ Z
intersection	X is the intersection of Y and Z	Y ∩ Z
disjointness	X is disjoint from Y	disjoint(Y)

Figure 1.3: Syntax of set constraints

1.1.1.5 Linear Precedence Constraints

Linear precedence constraints have various uses in linguistic descriptions. Their most obvious use is the modelling of word order phenomena. Other uses are in natural language semantics in the description of temporal precedence relations and of underspecified quantifier scope.

In figure 1.4 we describe the syntax of the linear precedence constraints supported by our implementation; for the formal semantics, refer to [Manandhar, 1995].

1.1.1.6 Guarded Constraints

Guarded constraints are used in logic programming to delay a constraint if not enough information is available for its deterministic execution. Such situations arise frequently in natural language processing when the same grammar is used bidirectionally for parsing and generation.

Therefore, it is a natural move to include guarded constraints into grammar formalisms. Our constraint language for guarded constraints supports the following general purpose syntax:

$$\begin{array}{lll} \mathit{case}(& [& \mathit{condition}_1 \;\Rightarrow\; \mathit{choice}_1, \\ & & \ldots \\ & & \mathit{condition}_n \;\Rightarrow\; \mathit{choice}_n \\ &]) & \\ \mathit{else} & \mathit{choice}_{n+1} & \end{array} \tag{1.4}$$

Each of the choice_i can be any term or another guarded constraint. Each of

LP Constraint	Meaning	Syntax for variable X
precedence	X precedes Y	precedes(Y)
precedence equals	X precedes or is equal to Y	precedes_equals(Y)
first daughter	X precedes all other elements of domain Y	fst_daughter(Y)
domain precedence	(every element of) domain X precedes (every element of) domain Y	dom_precedes(Y)
guard on precedence	if X precedes Y then X is unified with S, otherwise X is unified with T	if precedes(Y) then S else T

Figure 1.4: Syntax of linear precedence

the $condition_i$ (also known as *guard*) is restricted to one of the following forms (the variables $\exists x_1, \ldots, x_n$ stand for existentially quantified variables). Our constraint language is restricted to what is known as *flat guards* since no embedding is allowed in the guard (condition) part. However, this restricted language appears to be sufficient for linguistic applications.

$$\begin{array}{lll} condition \longrightarrow & \exists x_1, \ldots, x_n \; feature_term & \\ & \mid \exists x_1, \ldots, x_n \; exists(feature_term) & \quad (1.5) \\ & \mid precedes(x, y) & \end{array}$$

Guarded constraints can be thought of as *conditional constraints* whose execution depends on the presence of other constraints. The action $choice_i$ is executed if the current set of constraints *entail* the guard $condition_i$. The action $choice_{n+1}$ is executed if the current set of constraints *disentail* the all the guards $condition_1$ through $condition_n$. If the current set of constraints neither entail nor disentail $condition_i$ then the execution is *blocked* until more information is available.

The constraint solving machinery needed for implementing guards on feature constraints has been worked out in [Smolka and Treinen, 1994] and [Aït-Kaci and Podelski, 1994]. Our constraint language extends this to permit guards on set-memberships and guards on precedence constraints.

1.1.2 Definite Clauses

A program (or grammar) consists of definite clauses, which define relations, and associated constraints. A definite clause (or *Horn clause*) is an implication whose conclusion is a (relational) atom, and whose antecedent is a (possibly empty)

conjunction of (relational) atoms.[8] The general form of a definite clause is shown in (1.6). When talking about definite clauses, we use uppercase roman letters (A, B, C ...) as meta-variables for atoms, uppercase Greek (Γ, Δ, Θ ...) letters as meta-variables for sequences of atoms, and lowercase Greek letters (σ, τ, ϕ ...) for constraints (e.g., substitutions), and sequences of lowercase Greek letters for the merging of the respective constraints, i.e., most general unifiers. $\leftarrow$ is used as the implication symbol.

Each clause is associated with a constraint expressed in the constraint language given in the preceding section. Grammars are regarded as constraint logic programs, whose declarative and procedural semantics follows the constraint logic programming schema of Höhfeld and Smolka [Höhfeld and Smolka, 1988]. When it is necessary to mention the constraint of a clause explicitly, we write it in front of the clause as in (1.6).

$$\sigma(C \leftarrow A_1 \wedge \ldots \wedge A_n) \tag{1.6}$$

Occasionally, we omit the constraint associated with a definite clause in our notation.

$$C \leftarrow A_1 \wedge \ldots \wedge A_n \tag{1.7}$$

When we talk about definite clauses in general, which can be interpreted by any proof procedure for logic programs, we use the above notation, but when we talk about about Prolog clauses (which are intended to be executed by Prolog directly), we use Prolog notation with the `:-` connective.

Since grammars are generally regarded as logic programs in this thesis, we don't introduce any special notation for grammar rules. We place no special restrictions on the form of the constraint logic programs used to express grammars, unlike van Noord, who restricts grammars "to consist of definite clauses defining only one unary relation" [van Noord, 1993, p. 43] (the relation `sign/1`), and argues that all other relations can be compiled away by partial deduction techniques. In addition to the relation `sign/1`, van Noord introduces one additional relation `cp/2` (construct_phonology), which makes the combination of the phonological values of the daughters in a rule explicit, especially in cases where it is not restricted to concatenation.

$$sign(M) \leftarrow sign(D_1) \wedge sign(D_2) \wedge cp(M, \langle D_1, D_2 \rangle). \tag{1.8}$$

[8] We use the term atom here as it is used in the logic programming literature to mean a relation symbol and its arguments — not to be confused with an atomic value in a Prolog term or feature term: a term which has no arguments or features. In the logic programming literature, a definite clause is often equivalently formalised as a disjunction ($C \vee \neg A_1 \vee \ldots \vee \neg A_n$) with at most one positive (non-negated) literal, and the proof procedure is described as a refutation proof.

In this thesis, we will not make these restrictions, but allow grammars to consist of definite clauses defining arbitrary relations. This allows us to express principle-based grammars directly. In chapter 2 we will examine van Noord's claim that all relations except the `sign/1` and the `cp/2` relation can be compiled away by partial deduction techniques.

To summarise, our formalism gives the expressive power of a definite clause language, augmented with an extended constraint language (with sorted feature terms, finite domains, inequations and set descriptions).

1.2 Linguistic Deduction Algorithms

Once the grammar formalism is fixed, a deduction strategy must be selected. In this section, we will review previous work in linguistic deduction and outline the dimensions in which different algorithms can differ. In chapter 3, we present a new model (bottom-up Earley Deduction), which combines the useful properties of various algorithms presented in this section.

1.2.1 Relationship between NLP and Logic Programming

Logic-based grammars have always enjoyed a very close relationship to logic programming; in fact the very beginnings of logic programming (Colmerauer's Q-systems [Colmerauer, 1970]) have been motivated by natural language processing. Definite Clause Grammars [Pereira and Warren, 1980] have been an integral part of standard (Edinburgh) Prolog ever since its first implementation [Clocksin and Mellish, 1981; Bowen *et al.*, 1982].

There is a duality between (constraint-based) grammars and (constraint) logic programs: a definite clause grammar is at the same time a logic program.

This duality has the effect that the same collection of clauses can be regarded as a program or as a grammar. Whenever the distinction is not crucial, we will use whatever term best fits the current context. In order to avoid confusion between grammar rules and program rules, we will always refer to the former as "rules", and to the latter as "(non-unit) clauses". The left-hand side of a grammar rule will sometimes also be referred to as "mother (node)", (taking the view of a grammar rule as a local tree) and the categories on the right-hand side will also be referred to as "daughters." In linguistics, the head is a daughter in a local tree that shares certain syntactic or semantic features with the mother. In logic programming, the head of a clause is the consequent of a conditional (i.e., the positive literal in a Horn clause). In order to avoid terminological confusion, we will use the term *head* in its linguistic sense, and always refer to the head in the logic programming sense as "consequent" of a clause.[9]

[9] It is not only possible to view a grammar as a logic program, but conversely, a logic program can be treated as a grammar [Deransart and Małuszyński, 1993], and many well-known parsing

Together with a proof procedure for logic programs, a definite clause grammar becomes a parser or a generator for natural language.[10] For example, Prolog's standard proof procedure applied to a definite clause grammar yields a recursive descent (left-to-right, depth first) parser or generator. The deductive approach to NLP is not geared towards parsing or generation, but is inherently non-directional. However, from this general insight, it was still a long way towards algorithms that are really usable for both parsing and generation. Among other causes (see section 1.2.8), this is due to the fact that Prolog's proof strategy applied directly to DCGs has several serious drawbacks:

- Non-termination for left-recursive grammars.
- Duplication of deduction steps in different branches of the search tree.
- No support for incremental processing.
- Termination problems in case of generation.

Before we review alternative proof procedures, we briefly discuss the importance of difference lists for the efficient processing of DCGs.

Difference Lists

We want to point out that DCGs together with Prolog's proof strategy are only practically usable[11] because of their use of difference lists for representing strings. Note that DCG uses only concatenation as the basic operation for combining strings. Using Prolog's usual predicate `append/3` for concatenation would lead to serious efficiency problems for DCG. Consider the following rule

$$\text{s(S) :- np(NP), vp(VP), append(NP,VP,S).} \tag{1.9}$$

Whatever order of goals is chosen, there is an efficiency problem. If the call to `append/3` is not the first goal, Prolog will generate (potentially infinitely many) NP's ad libitum; and if the call to `append/3` is the first goal, then a given input string of length n can be split $n+1$ ways. In case of generation, the converse holds: if the call to `append/3` comes first, it can generate strings ad libitum.

---

algorithms for context-free or unification grammars can be used as proof procedures for logic programs (e.g. Earley deduction as a proof procedure for logic programs based on Earley's context-free parsing algorithm).

[10] In chapter 2.2.1 we use the compilation of DCGs into recursive descent and into left-corner parsers as illustrations of the partial deduction technique.

[11] Even though unmodified DCGs are not widely used for NLP because of the mentioned problems, they enjoy widespread use among logic programmers as a general string processing mechanism, and as data structures for threading information through programs. See [Pereira and Shieber, 1987, p. 168 ff.] for some nice examples of parsing algorithms written as DCGs.

Using difference lists, however, the call to `append/3` is replaced by a *concatenation constraint* that is enforced at any time by instantiation of variables.[12] The following rule (1.10) is the encoding of rule (1.9) in difference list format. Note that we represent difference lists by a pair of variables connected with the functor `-/2`.

s(S0-S) :- np(S0-S1), vp(S1-S). (1.10)

The problem of defining a data structure that exhibits at least some of the advantages of difference lists will come up for grammars that are not limited to concatenation as the sole operation for combining strings (see section 3.2.6).

1.2.2 Alternatives to Prolog's Search Strategy

Various alternatives to Prolog's top-down, depth-first search strategy have been investigated in logic programming and in NLP. They differ from Prolog's strategy along several dimensions:

Direction of processing: The extremes are pure top-down (backward chaining), and pure bottom-up processing (forward chaining); directed (mixed) methods combine the goal-directedness of top-down processing with the data-driven aspect of bottom-up processing.

Selection function: Choice of the next goal to process from the antecedent of a clause. This can either be fixed in advance (e.g., the leftmost goal or a goal that shares essential variables with the consequent[13]) or dynamically depending on the instantiation of variables (coroutining).

Memoing: The question of memoing concerns the extent to which solutions to goals are stored and re-used either at compile time or at runtime. Techniques to be considered under this heading are partial deduction,[14] abstract interpretation, explanation-based learning, well-formed substring tables, and tabulated deduction (Earley deduction).

Constraints: This topic concerns the choice of an appropriate constraint language, and the question how constraints should be evaluated, especially the question whether constraints should be checked as soon as possible or delayed. Constraint solving can play a more or less central role for an NLP algorithm — up to the extent where the algorithm consists almost exclusively of constraint solving steps.

---

[12] Aït-Kaci (p.c.) pointed out that the use of variable bindings in logic programming languages constitutes an efficient representation of partial solutions of equality constraints.

[13] We use the term *consequent* here for the head of a clause in the logic programming sense. In the context of NLP, we wish to reserve the term *head* for the syntactic or semantic head of a phrase(cf. page 16).

[14] Partial deduction is also referred to as *partial evaluation* or *partial execution* (cf. chapter 2).

Search Strategy: The extremes in this spectrum are pure depth-first and breadth-first search. In between these extremes, there are a number of heuristically guided search strategies, for which the choice of a heuristic is an important consideration.

Shieber et al. [Shieber *et al.*, 1994] have formulated various parsing algorithms as deduction algorithms. In this system, an algorithm is specified explicitly by stating a set of inference rules. The inference rules are applied by a bottom-up proof procedure. However, their formalisation makes crucial use of string positions in the representation of items, and is therefore useful for the case of parsing, but not immediately for generation.

In a similar vein, Sikkel [Sikkel, 1994a; Sikkel, 1994b; Sikkel, 1994c] defines the notion of parsing schemata, which allow a uniform description of different parsing algorithms, thereby exhibiting similarities and differences between the algorithms. It should be noted that Sikkel's work is only concerned with context-free grammars, although it can be generalised to constraint-based grammars.

In the remainder of this chapter, we will discuss previous work that has been done on linguistic deduction algorithms by making particular choices in the above dimensions, and motivate the choices we make.

1.2.3 Direction of Processing

The issue with direction of processing in a deduction algorithm is whether the algorithm is driven by the goal to be proved (top-down, or backward chaining) or by the available input data (bottom-up, or forward chaining).

In logic programming, in general, top-down deduction is preferred for reasons of efficiency. Top-down processing is used in Prolog and logic programming languages derived from it, and in Earley deduction [Pereira and Warren, 1983]. Bottom-up processing suffers from the problem that it is not very goal-directed, and that it is in general hard to select the clauses that should be used as input to the bottom-up process, so that lots of clauses can be derived that are irrelevant to a proof of a given goal.

For NLP, however, bottom-up parsing is often preferred; this is possible because it is easy to select the clauses that should be used as input for a given goal. Normally, they should be the lexical entries of the words which occur in the input string to be parsed and the rules of the grammar.

There are useful alternatives that combine the benefits of top-down and bottom-up search. These are directed (mixed) methods that combine top-down and bottom-up processing. Wirén has performed a comparison of "rule invocation strategies" for chart parsers [Wirén, 1987], which comes to the conclusion that directed strategies are most efficient for context-free chart parsing.

Among the directed methods, we will discuss left-corner parsing (for grammars based on concatenation), semantic-head-driven generation, and head-corner

parsing (for grammars which make use of more powerful string operations than concatenation) below.

1.2.3.1 Left-Corner Parsing

Left-corner parsing starts from a given goal (to parse a string), and selects a lexical entry (a unit clause) for the leftmost word in the string (the left corner). This clause starts the bottom-up processing: if the selected unit clause is already a solution to the original goal, the algorithm returns the solution, otherwise the algorithm looks for a non-unit clause in which the selected unit clause is the first goal.[15] If the remaining goals in the non-unit clause can be proven in the same fashion, the consequent of the clause becomes the next input to the bottom-up process. This process is repeated until a solution to the original goal is found. The Prolog implementation of the algorithm is shown in figure 1.5.[16]

In logic-programming-based NLP, this algorithm has been employed in the BUP compilation scheme for definite-clause grammars [Matsumoto *et al.*, 1983]. BUP compiles a DCG into a left-corner parser for the same grammar.

The performance of the algorithm can be improved further by making use of a reachability relation. Clauses that serve as input of the bottom-up process are only used if they are reachable from the current goal as the transitive closure of the "leftmost-daughter" relation. For context-free phrase structure grammars, this reachability relation can be precompiled for a given program and serves to reduce the search space during processing.

While a left-corner strategy is useful for parsing DCGs, it is less useful for generation (where the syntactic category of the left corner may not yet be instantiated when it is processed) or for parsing languages with discontinuous constituency with grammars which use string operations beyond concatenation (where the leftmost word of the string is not in general reachable from the goal by following the transitive closure of leftmost goals of clauses whose consequent matches the goal).

1.2.3.2 Semantic-Head Driven Generation

The Semantic-Head Driven Generation Algorithm [Shieber *et al.*, 1990; van Noord, 1993] performs surface generation in an analogous fashion to the left-corner parser. Instead of the left corner relation, the algorithm uses the semantic-head relation.

[15] In verbal descriptions of non-deterministic algorithms, we often take the liberty of using deterministic formulations, and do not explicitly mention that alternative choices can be explored by backtracking.

[16] In the documentation of Prolog procedures, we use the traditional notation for the expected instantiation of arguments: + for (input) arguments that should be instantiated, − for (output) arguments for which instantiation is not expected, and ? for arguments whose instantiation does not matter. Difference lists are encoded by a pair of variables connected with the functor `-/2`. The angle brackets (⟨ and ⟩) are used in list notation instead of the usual Prolog square brackets to avoid confusion with the square bracket notation for feature structures.

```
% parse(?GoalCategory,+InputDiffList)
parse(GoalCat,S0-S) :-
        leaf(LexCat,S0-S1),
        bu_step(LexCat,GoalCat,S1-S).

% bu_step(+CurrentCategory,?GoalCategory,+InputDiffList)
bu_step(GoalCat,GoalCat,S-S).
bu_step(Small,GoalCat,S0-S) :-
        rule(Cat,⟨Small|Rest⟩),
        parse_rhs(Rest,S0-S1),
        bu_step(Cat,GoalCat,S1-S).

% parse_rhs(+ListOfCategories,+InputDiffList)
parse_rhs(⟨Cat|Cats⟩,S0-S) :-
        parse(Cat,S0-S1),
        parse_rhs(Cats,S1-S).
parse_rhs(⟨ ⟩,S-S).
```

Figure 1.5: Left corner parsing algorithm in Prolog

The semantic head of a rule is defined to be the daughter whose semantics is identical with the semantics of the mother (or stands in an easily computable relation with the semantics of the mother for grammars that do not handle construction of logical forms by simple equality constraints). Rules that do have a semantic head are called chain rules, and rules that don't have a semantic head are called non-chain rules. The computation proceeds by starting the bottom-up process by selecting and proving a non-chain rule (often a lexical entry) whose mother's semantics is identical with the semantics of the goal. The mother M of this rule becomes input to the bottom-up process by selecting a chain rule R whose semantic head has identical semantics as M, and then generating the other daughters of the rule in the same fashion. If the mother node of the chain rule R matches the original goal, the process terminates; otherwise the mother of R becomes the new input to the bottom-up process.

We give the semantic-head driven algorithm as a Prolog program (cited from [Shieber *et al.*, 1990; van Noord, 1993]). For this algorithm, chain rules are represented as `chain_rule(Head,LHS,RHS)`, and non-chain rules as `non_chain_rule(LHS,RHS)`. The procedure `connect/2` performs the bottom-up step. Nodes are represented by `node(Cat,DLin,DLout)`, where `Cat` is a term representing syntactic and semantic information, the pair of variables `DLin` and `DLout` is a difference list representing the generated string. `node_semantics/2` is

a relation between a node and its semantic content, and `chained_nodes/2` is a 'reachability relation', which is used to test whether one node can be the semantic head of another.

Semantic-head driven generation is easily applicable to grammars that make use of more powerful operations for combining strings. In this case the operation merely constrains the possible surface word orders.

Van Noord discusses possible extensions of the semantic-head driven generation algorithm such as extending the prediction step to take into account syntactic information in addition to semantic information, using memoing to improve performance, and delaying lexical choice. The compilation of a grammar and the generation algorithm above into a more efficient executable Prolog program is described in [Block, 1991].

1.2.3.3 Head-Corner Parsing

Left-corner parsing only makes sense for grammars which use concatenation as the only operation for combining strings. For grammars with discontinuous constituents that use more powerful string operations, "head-corner parsing" has been proposed. Before discussing this algorithm, we first show why top-down parsing would be extremely inefficient for such grammars.

We have already noted in section 1.2.1 that top-down processing would be inefficient even for grammars based on concatenation unless difference lists are used. For grammars based on more powerful string operations, the efficiency of top-down processing is even worse because the "reverse" application of these string operations is very non-deterministic, and a huge search space would results from this.

For instance, the sequence union operation (cf. section 3.2.3) applied in reverse to divide an input string with n words has 2^n solutions; so for an input string of only 10 words, there are 1024 different solutions.

In order to overcome these problems, head corner parsing applies a mixed bottom-up and top-down strategy similar to that employed by left-corner parsers and head-driven generators. The head-corner parser uses a notion of *syntactic head.* The syntactic head of a local tree shares certain syntactic features with its mother. Parsing starts by selecting a lexical entry for a word in the input string which can be a syntactic head of the goal category. Unlike the selection of the leftmost word of the input string in case of left-corner parsing, this step is not deterministic for head-corner parsers.

Under a backtracking proof strategy (as in Prolog), the non-determinism in selecting the lexical item which starts the bottom-up process has the effect that any deduction results based on a wrong selection are lost after backtracking and cannot be recovered. This problem has been tackled by making use of memoing (cf. section 1.2.5), i.e., maintaining well-formed substring tables for constituents built up from wrong guesses [Bouma and van Noord, 1993].

```
gen(Cat,String) :- generate(node(Cat,String,⟨ ⟩)).

generate(Root) :-
        applicable_non_chain_rule(Root,Pivot,RHS),
        generate_rhs(RHS),
        connect(Pivot,Root).

generate_rhs(⟨ ⟩).
generate_rhs(⟨H|T⟩) :-
        generate(H),
        generate_rhs(T).

connect(Pivot,Root) :-
        applicable_chain_rule(Pivot,LHS,Root,RHS),
        generate_rhs(RHS),
        connect(LHS,Rule).
connect(Pivot,Root) :-
        unify(Pivot,Root).

applicable_non_chain_rule(Root,Pivot,RHS) :-
        node_semantics(Root,Sem),
        node_semantics(Pivot,Sem),
        non_chain_rule(LHS,RHS),
        unify(Pivot,LHS),
        chained_nodes(Pivot,Root).

applicable_chain_rule(Pivot,Parent,Root,RHS) :-
        chain_rule(Parent,RHS,SemHead),
        unify(Pivot,SemHead),
        chained_nodes(Parent,Root).
```

Figure 1.6: Semantic-head driven generation algorithm in Prolog

Since the operation of a head-driven algorithm depends on the presence of the head element in order to start the bottom-up process, the algorithm is not well suited to incremental processing of natural language, i.e., processing every piece of the input as soon as it is perceived. Incremental processing is characteristic of human NL processing, and is a desirable property for NL understanding systems that should operate in real time without any noticeable delays caused by linguistic processing.

1.2.4 Selection Function

The selection function of Prolog is to choose leftmost goal in the sequence of goals that still must be proven. This selection function has been changed or improved for various purposes.

One optimisation concerns reordering of goals to ensure that they every goal is sufficiently instantiated when it is called, in order to guarantee that it has only a small number of solutions and avoid non-termination or very large search spaces. This is for example the approach taken in the Essential Arguments Algorithm [Strzalkowski, 1991], where an off-line re-ordering at compile time is used.

The processing model for the grammar formalism CUF uses a *deterministic subgoal reduction* strategy [Dörre and Dorna, 1993]. This strategy attempts to reduce the search space by preferring goals which can be resolved deterministically. While this is an attractive strategy because it restricts the size of the search space and frees the grammar developer from having to specify an order to goals, it incurs a certain inefficiency in processing by the need to determine which goals can be reduced deterministically at every step of the computation.

Another way to achieve a similar effect dynamically is by delaying goals in which certain variables are not (yet) instantiated. Prolog II and Sicstus Prolog realise this delaying of goals through their selection function: choose the leftmost *unblocked* goal. A goal is blocked until a condition attached to it is satisfied. In Prolog II the condition can be that a particular variable becomes instantiated — the goal is said to be *frozen* until the variable is instantiated. In Sicstus Prolog, the condition can be that a particular variable is either instantiated, or instantiated to a ground term, or known to be equal or different from another variable. Conjunctions and disjunctions of these conditions are possible.

Neumann's uniform tabular algorithm UTA has a dynamic selection function that chooses the first goal in which one of the essential arguments (string or logical form) is instantiated [Neumann, 1994b]. This can be handled with the blocking mechanism of Sicstus Prolog.

For top-down processing, the choice of a selection function is of crucial importance because of the way information flows between different goals. If a goal is not specified enough, it may generate an infinite number of solutions without ever enumerating the correct one. Therefore, it is crucial to execute those goals first whose solutions instantiate variables in other goals. Generally, the optimal

ordering of goals will be different for parsing and for generation. Bidirectional algorithms will either use two different goal orderings for parsing and generation, or do the goal ordering via a dynamic selection function.

For bottom-up processing, the choice of the selection function is not so crucial because the situation where solutions to an underspecified goal can be enumerated without control does not arise. Nonetheless, the choice of the selection function can bring performance benefits since it may in general be more efficient in NLP to look for arguments that match a given functor than for functors that match a given argument.

Like in head-driven processing, this selection need not be done dynamically, but can be fixed at compile time. In the following chapters, we assume that this selection has been done by ordering of the goals in a clause, and will not mention the issue any further.

Problems in which no optimal ordering of goals can be found occur for example in the case of empty heads for head-driven processing. However, if there are more powerful operations than concatenation for the combination of strings (cf. section 3.2), then it is not necessary to make use of empty heads.

For auxiliary goals such as the concatenation of lists etc., which depend on the instantiation of variables, we allow to delay them by coroutining, i.e., they are blocked until the specified conditions (instantiation or equality of variables) are satisfied. At runtime, we use the same selection function as Sicstus Prolog, namely choosing the leftmost unblocked goal (cf. section 1.2.6.3).

1.2.5 Memoing

The purpose of memoing (or *memoisation* or *tabulation*) techniques is to avoid duplication of computation. This is achieved by storing solutions of goals, so that they can be looked up when the same goal (or an instance of it) needs to be proved again.

Memoing techniques are well-known in different areas of programming [Michie, 1968]. In NL parsing, memoing is realised by the use of well-formed substring tables, and in chart parsing. Norvig presents memoing techniques for LISP programming [Norvig, 1992; Norvig, 1991], and points out that "one can achieve [the results of chart parsers] by augmenting a simple parser with memoisation" [Norvig, 1992, p. 679]. Johnson has improved Norvig's algorithm to handle grammars with left-recursion [Johnson, in press]. In the field of logic programming, there is a well-known memoing technique called Earley deduction or OLDT[17] resolution. In this section, we will first review chart parsing, and then discuss Earley deduction.

[17] The abbreviation expands to *Ordered selection strategy with Linear resolution for Definite clauses with Tabling*.

1.2.5.1 Chart Parsing

Earley's algorithm [Earley, 1970] is a tabular parsing algorithm for general context-free grammars. Earley's algorithm is important because memoing reduces the exponential complexity of backtracking parsers to polynomial complexity; the complexity of Earley's algorithm with respect to the length of the input string is $O(n^3)$.[18]

It has soon been recognised that Earley's algorithm allows of significant variation; these variations are generally referred to with the generic term *chart parsing,* which Kay [Kay, 1980] describes as an *algorithm schema* which can be turned into several different algorithms if by varying it along the following dimensions:[19]

- Direction of processing (cf. section 1.2.3)
- Search strategy (cf. section 1.2.7).

In the following, we will briefly review the data structures (items) used in chart parsers, and the inference rules for combining these data structures. Chart items are pairs ⟨*Grammar Rule,Position*⟩. *GrammarRule* is a pair ⟨*LHS,RHS*⟩ of a non-terminal symbol of the grammar (*LHS*), and a sequence of non-terminal symbols (*RHS*), and Position is a pair ⟨Begin,End⟩ that encodes the starting and ending position of the item in the input string. We generally write the pair ⟨*LHS,RHS*⟩ in the following notation: *LHS* → *RHS*. The sequence of categories in *RHS* is divided into two parts by a dot, where the portion to the right of the dot is the sequence of non-terminals that must still be found. We call the sequence of categories after the dot the *remainder*. In (1.11), we show the notation for an item which spans string positions 0 to 5, covers an NP, and would be an S if a VP were found to its right.

$$\langle S \rightarrow \mathrm{NP}\,.\,\mathrm{VP}, \langle 0, 5\rangle\rangle \tag{1.11}$$

The meaning of an item is that the substring spanning $Position$ would be a constituent of category LHS, if the sequence of categories after the dot (the remainder) were found at a position to its right.[20]

Two types of items are distinguished:

[18] Other early approaches to tabular parsing are Kay's powerful parser [Kay, 1967] and Kaplan's general syntactic processor [Kaplan, 1973].

[19] A fine example of a flexible and modular chart parsing system which allows both kinds of variation, and provides interfaces to various grammar formalisms is the MCHART system [Thompson, 1983].

[20] Other formalisations as triples ⟨*LHS, RHS, Position*⟩ or ⟨*Grammar Rule, Begin, End*⟩, quadruples ⟨*LHS, RHS, Begin, End*⟩, or quintuples ⟨*LHS,Found,To-Find,Begin,End*⟩ have also been used. We prefer the current formalisation because it allows a uniform presentation of chart parsing and Earley deduction. It is not strictly necessary to include the categories of *RHS* that have already been found as we do in the dotted rule notation because the categories which have been found do not play any role in the algorithm.

- If the remainder is empty, the item is a *passive item*.
- If the remainder is non-empty, the item is an *active item*.

There are three rules in Earley's algorithm:

1. Prediction
2. Completion (Fundamental rule)
3. Scanning

The most important operation in chart parsing is combining active and passive items according to the completion rule, or "fundamental rule."[21] In the following, `A, B, C` ... are positions, $\mathtt{T}_1$ ... are terminal symbols, $\mathtt{N}_1$ ... non-terminal symbols, and α, β ... are sequences of non-terminals. An active item $A = \langle N1 \rightarrow \alpha.N2\ \beta, \langle A, B\rangle\rangle$ and a passive item $P = \langle N2' \rightarrow \gamma., \langle B, C\rangle\rangle$ can be combined if $N2 = N2'$. The item resulting from the combination has the category of $N1$, remainder β, and position $\langle A, C\rangle$. This rule is also applicable if Prolog terms or feature structures are used instead of atomic non-terminal symbols. In this case, we require that $N2$ and $N2'$ be unifiable (cf. section 1.2.5.2. We can state this completion rule for the combination of items in the format of an inference rule (1.12).

$$\frac{\begin{array}{c}\langle \text{N1} \rightarrow \alpha\ .\ \text{N2}\ \beta,\langle \text{A,B}\rangle\rangle \\ \langle \text{N2} \rightarrow \gamma\ .,\langle \text{B,C}\rangle\rangle\end{array}}{\langle \text{N1} \rightarrow \alpha\ \text{N2}\ .\ \beta,\langle \text{A,C}\rangle\rangle} \tag{1.12}$$

Prediction and scanning are not really inference rules, but they serve the purpose of selecting grammar rules and lexical items that should be turned into items, to which the completion rule can be applied.

The prediction rule selects grammar rules to be added to the chart as items. If an item I is added to the chart whose remainder starts with category C, then C is predicted at the end position of item I, if such a prediction has not already been made. Predicting an item at position P means that for every rule $C \rightarrow R$ in the grammar whose left-hand side is category C, an item $\langle \text{C} \rightarrow \text{R},\langle \text{P,P}\rangle\rangle$ is added to the chart, written as a pseudo-inference rule in (1.13).

$$\frac{\begin{array}{c}\langle \text{N1} \rightarrow \alpha\ .\ \text{N2}\ \beta,\langle \text{Begin,End}\rangle\rangle \\ \text{N2} \rightarrow \gamma\end{array}}{\langle \text{N2} \rightarrow .\gamma,\langle \text{End,End}\rangle\rangle} \tag{1.13}$$

Scanning serves the purpose of selecting lexical entries that should be turned into chart items. The scanning step can is given as a pseudo-inference rule in (1.14).

[21] This rule is also referred to as the **reduction rule** in the literature on Earley deduction.

$$\frac{\langle \text{N1} \rightarrow \alpha \ \textbf{.}\ \text{N2}\ \beta, \langle \text{B,n-1} \rangle \rangle \quad \text{T is the } n^{th} \text{ word in the input string and there is a rule N2} \rightarrow \text{T}}{\langle \text{N2} \rightarrow \text{T}\ \textbf{.}, \langle \text{n-1,n} \rangle \rangle} \tag{1.14}$$

The algorithm consists of applying the completion (1.12), prediction (1.13), and scanning (1.14) rules until a solution has been found or no more rule is applicable.[22] The order in which the rules are to be applied can be freely chosen, so that chart parsing is an *algorithm schema* rather than an algorithm. It is this flexibility that makes chart parsing a favourable framework for experimenting with different search strategies, and for incremental processing.

Bottom-up chart parsing makes use of the same completion rule, but there is no prediction rule,[23] and the scanning rule does not depend on the presence of an item. It is easy to see that bottom-up chart parsing is less goal-directed than the top-down variant. If bottom-up chart parsing has been widely used in spite of this drawback, this is the case for the following reasons:

- Applications where it is not known in advance which goal category should be recognised.
- Grammars which make use of highly underspecified rules, so that prediction does not help much to improve performance.
- Applications such as speech recognition or processing ill-formed input where the emphasis is on making use of the available input data.

In section 1.2.5.4 on Earley Deduction, both the fundamental rule and the indexing scheme will be generalised.

These methods constitute the state of the art in processing context-free grammars. They are also applicable to unification-based grammars such as DCGs, as discussed in the following.

1.2.5.2 Chart Parsing for Unification-Based Grammars

In order to use chart parsing with unification-based grammars (such as DCG, LFG, PATR), special care must be taken at those steps in the algorithm where it is checked that the same item is not added twice to the chart, and that the same prediction is not made twice. In the case of context-free grammars, a simple equality test is enough. In case of unification-based grammars, it must be checked

[22] Of course, the same rule should never be applied twice to the same pair of items, or pair of item and word or rule.

[23] We could say that every grammar rule $A \rightarrow B$ is turned into a chart item ⟨A → B,⟨P,P⟩⟩ which is present at every string position P. In actual implementations, the operation of the completion rule would be modified to allow it to access grammar rules.

that there is no item present in the chart that *entails* the item that is about to be added.[24]

For prediction, it is not enough to check that no subsuming prediction has been made before because there can be a sequence of predictions none of which subsume each other, as for example by the following grammar rule and the goal category $s(\langle\ \rangle)$.

$$\mathrm{s(L)} \rightarrow \mathrm{s(\langle a|L \rangle)} \tag{1.15}$$

In order to avoid such prediction loops, Shieber has introduced the notion of restriction, which instantiates terms to be predicted only to a certain depth, so that eventually terms will subsume each other [Shieber, 1985]. Samuelsson proposed an alternative to restriction in the form of anti-unification [Samuelsson, 1994b].

The notion of restriction can also be used to avoid having to check for constraint entailment instead of simply subsumption of terms. This can be done by defining the restrictor in such a way that it does not only eliminate equality constraints over feature structures beyond a certain path length, but also other types of constraints. In this way prediction loops are still avoided, as well as the possibly computationally expensive check for constraint entailment.

Unlike the normal top-down, depth-first strategy of Prolog, chart parsing does not suffer from the problem of left recursion, and avoids duplication of computations on backtracking. A potential drawback is the amount of copying involved when storing items, and the cost associated with the subsumption check that is needed to avoid redundant items and non-terminating predictions.

1.2.5.3 Subsumption Checking

In Earley deduction (and in unification-based chart parsing), a subsumption check is used to ensure that an item is not added to the chart if an item that subsumes it is already present in the chart. If a derived clause is subsumed, "the derivation step is said to be blocked" [Pereira and Warren, 1983, p. 139]. We have already seen that this subsumption check — together with the restriction technique known from unification-based chart parsing — avoids prediction loops.[25] However, the question is whether the subsumption check must also be applied to items that do not arise from prediction, but from completion. It would be desirable to leave the subsumption check out in these cases because of its adverse effect on performance.

First, we note that different items in the chart have different derivations. This follows from the fact that the same item is never added twice by prediction due to subsumption checking, and that the completion rule only applies to items that

[24]In general, it is necessary to check *entailment*, of which *subsumption* is a special case when the constraint language consists only of (first-order or feature) terms.

[25]For this case, it suffices to keep track of selected literals that have been used extensively to generate prediction steps, and exclude their use as candidates for further prediction steps, as Pereira and Warren (p. 142) point out.

are freshly added to the chart and hence never combines the same two items more than once.

If the same item arises more than once from different derivations, we have a case of spurious ambiguity. The most inefficient way of dealing with it is by performing the subsumption check. If the subsumption check is left out, the situation with respect to spurious ambiguity is the same as for Prolog's top-down backtracking search strategy.

Preferred ways of dealing with spurious ambiguity are

- defining a canonical derivation among the different derivations that lead to the same result, and suppressing all other derivations (cf. the work by Pareschi [Pareschi and Steedman, 1987] for a canonical derivation for Combinatory Categorial Grammar), or

- changing linguistic analyses so that they no longer give rise to spurious ambiguities (cf. the analysis of partial verb phrase fronting in German by Nerbonne [Nerbonne, 1994], which improves upon alternative analyses by eliminating spurious ambiguities.).

Note also that grammatical theories like HPSG encode the derivation in the features structures themselves through the DAUGHTERS feature. Such an encoding defeats the very purpose of subsumption checking since it will always fail for items with different derivations.

We conclude that the subsumption check can safely be left out for items arising from the completion step for grammars that do not exhibit spurious ambiguities or for grammars that encode the derivation in the feature structures (since it would always fail in this case).

Subsumption checking for items arising from completion would only be needed for grammars which contain spurious ambiguities, and for which no canonical derivation has been defined.

1.2.5.4 Earley Deduction

In this section, we present Earley Deduction, the linguistic deduction framework which provides the basis for incremental and robust processing of natural language. Earley deduction is a proof strategy for logic programs that terminates on a larger class of programs[26] than Prolog's SLD resolution proof procedure, and whose basic idea is derived from Earley's context-free parsing algorithm [Earley, 1970].

Earley deduction is a very attractive framework for natural language processing because it has the following properties and applications.

[26] It is known that Earley deduction terminates for all datalog programs, i.e., logic programs that do not make use of function symbols, which are relevant for the field of deductive databases, but not for NL grammars. For a discussion of termination properties, cf. section 3.4.3.

- Memoisation and reuse of partial results.
- Incremental processing by addition of new items.
- Hypothetical reasoning by keeping track of dependencies between items.
- Best-first search by means of an agenda.

Warren has recognised that a chart parser for DCG also constitutes a proof procedure for definite clause programs [Warren, 1975]. The earliest published description of the algorithm is found in [Pereira and Warren, 1983].

In order to view chart parsing as parsing as deduction, we make the relation between a category and string explicit, like in going from context-free grammar rules to DCG format. For example, the application of the fundamental rule in (1.16) can be changed to the one in rule (1.17) which makes strings and the append relation explicit, or rule (1.18) which makes use of difference lists. Note that we use grammar rule notation (with the operator →) in the chart parsing case (1.16), whereas we use definite clause notation (with the operator ←) in the case of Earley deduction ((1.17) and (1.18)).

$$\frac{\begin{array}{c}\langle \mathrm{S} \rightarrow .\mathrm{NP\ VP}, \langle 0,0\rangle\rangle \\ \langle \mathrm{NP} \rightarrow \alpha., \langle 0,1\rangle\rangle\end{array}}{\langle S \rightarrow \mathrm{NP}\ .\ \mathrm{VP}, \langle 0,1\rangle\rangle} \tag{1.16}$$

$$\frac{\begin{array}{c}\langle \mathrm{s(X)} \leftarrow \mathrm{np(A)} \wedge \mathrm{vp(B)} \wedge \mathrm{append(A,B,X)}, \langle 0,0\rangle\rangle \\ \langle \mathrm{np}(\langle \mathrm{john}\rangle) \leftarrow, \langle 0,1\rangle\rangle\end{array}}{\langle \mathrm{s(X)} \leftarrow \mathrm{vp(B)} \wedge \mathrm{append}(\langle \mathrm{john}\rangle,\mathrm{B,X}), \langle 0,1\rangle\rangle} \tag{1.17}$$

$$\frac{\begin{array}{c}\langle \mathrm{s(A\text{-}C)} \leftarrow \mathrm{np(A\text{-}B)} \wedge \mathrm{vp(B\text{-}C)}, \langle 0,0\rangle\ \rangle \\ \langle \mathrm{np}(\langle \mathrm{john} \mid \mathrm{X}\rangle\text{-}\mathrm{X}) \leftarrow, \langle 0,1\rangle\ \rangle\end{array}}{\langle \mathrm{s}(\langle \mathrm{john} \mid \mathrm{B}\rangle\text{-}\mathrm{C}) \leftarrow \mathrm{vp(B\text{-}C)}, \langle 0,1\rangle\ \rangle} \tag{1.18}$$

Here it is clear to see that the fundamental rule of chart parsing is an inference rule. Also note that if strings and the append relation are made explicit, the use of the position is not needed any more to rule out any invalid deductions, but only to exclude irrelevant deductions that would not contribute to the analysis of the sentence to be parsed. Without position, any constituents in the chart can be combined, so that all grammatical permutations of the input string would be produced in the process. In fact, if there is no prohibition against using any item more than once in a derivation, an infinite number of grammatical constituents can be produced that do not contribute to an analysis of the input string.[27]

[27] This is only true if lexical signs are not instantiated with the input string for a given parse. If the difference list in the lexical entry for *john* were instantiated with the input string (i.e. for an analysis of the string *john walks slowly* it would be instantiated as np(⟨john,walks,slowly⟩-⟨walks,slowly⟩), no redundant derivations could be produced. However, such instantiation of lexical entries with the input string prevents their reuse in a fully incremental computation, which deals with changes in the input string (cf. section 3.5).

So, while encoding of the position is not necessary to exclude invalid deductions, it serves as an indexing scheme that helps to avoid irrelevant deductions.

The terminology we use follows closely the established terminology from chart parsing.

Chart: In this work, the term "chart" is used for a collection of *items*. Items are clauses (lemmata) plus additional information (index etc.). The chart consists of a subset of (instances of) the program clauses, and of derived clauses.

Item: An item is a clause plus additional information, e.g., an index. The subject of indices will be discussed below in section 3.1.2

In Earley deduction, there are only two inference rules: the prediction rule (rule (1.19), called *instantiation* in [Pereira and Warren, 1983]) and the completion rule (rule (1.20), called *reduction*); the scanning rule is not needed since scanning in the Earley parser only serves the purpose of reading the next word in the input string. Lexical lookup in case of Earley deduction is merely a prediction of a unit clause.

Since Earley deduction is a general proof procedure for logic programs, there is not necessarily an input string. If there is one, it is encoded in the query.

In the instantiation rule (1.19), A, B and B' are atoms,[28] Γ is a (possibly empty) sequence of atoms, and σ is the merged constraint (most general unifier) of B and B'. In the completion rule (1.20), X, G and G' are atoms, Ω is a (possibly empty) sequence of atoms, and σ is the merged constraint (most general unifier) of G and G'. The leftmost atom in the body of a non-unit clause is always the selected goal.

$$\frac{\begin{array}{c} A \leftarrow B \\ B' \leftarrow \Gamma \end{array}}{\sigma(B \leftarrow \Gamma)} \tag{1.19}$$

$$\frac{\begin{array}{c} X \leftarrow G \wedge \Omega \\ G' \leftarrow \end{array}}{\sigma(X \leftarrow \Omega)} \tag{1.20}$$

The algorithm for Earley deduction for a goal G can be stated very simply:

1. Predict the goal G.[29]

[28] We use the term atom here as it is used in the logic programming literature to mean a relation symbol and its arguments — not to be confused with an atomic value in a Prolog term or feature term: a term which has no arguments or features.

[29] Technically, this proceeds by adding an item $\langle \mathrm{ans}(\mathcal{V}(\mathrm{G})) \leftarrow \mathrm{G} \rangle$ to the chart and applying the prediction inference rule to it. $\mathcal{V}(\mathrm{X})$ is a sequence consisting of the free variables in X. A solution is found when the chart contains a passive item $\langle \mathrm{ans}(_) \rangle$.

2. Apply the prediction and the completion rules to the items in the chart until either a solution is found or no more rules are applicable, and add the result of the inference as an item to the chart.

Of course the inference rules is applied only once to each pair of items. If the result of an inference step (prediction or completion) is already subsumed by an item in the chart, then it is said to be blocked and not added to the chart.

The algorithm for Earley deduction is given in figure 1.7.

1.2.5.5 Recent Developments in Earley Deduction

Recently, there has been a lot of interest in Earley deduction with applications to parsing and generation. We summarise the most important developments in the following.

Chart-based methods have been used for formalisation of parsing with the Lambek calculus by Esther König [König, 1990]. The important contribution of this work is the introduction and discharge of assumptions in the Lambek calculus for the treatment of long-distance dependencies at the level of the formalism, instead of using threading techniques.

Dörre addresses the problem of coroutining in Earley deduction, i.e., goals which depend on each others' partial solutions, and introduces the notion of *bundled goals* which are treated as if they were one goal [Dörre, 1993]. In addition Dörre is also concerned with the question that not all goals are best treated by Earley deduction, and distinguishes *trigger goals* which are treated by Earley deduction (i.e. lead to predictions) and other goals which are treated by a normal top-down proof.

The same problem of coroutining — with applications to GB parsing — has been addressed by Mark Johnson [Johnson, 1993], [Johnson and Dörre, 1995].

Den presents a method for cost-based abduction, which is an instance of Earley deduction which utilises an agenda in order to choose derivations which make use of assumptions with minimal cost [Den, 1994].

Earley deduction has also been rediscovered in logic programming and extended to handle negation by Beckstein and Kim [Beckstein and Kim, 1991]. It is closely related to OLDT resolution, which is the basic deduction engine of XSB Prolog, developed by D.H. Warren [Warren, 1989; Warren, 1992]. Deduction procedures very similar to Earley deduction have also been used in the area of deductive databases [Bancilhon and et al., 1986; Ramakrishnan, 1991].

Earley deduction has also been used for generation and for bidirectional algorithms that combine parsing and generation within one system (cf. section 1.2.8).

All of the methods presented above (with the exception of deductive databases) operate top-down; an approach which has several drawbacks:

1. In grammars which use more powerful string operations than concatenation,

procedure prove($Goal$):
– predict($Goal$)
– consume-agenda
– for any item G
  – return mgu($Goal, G$) as solution if it exists

procedure add item I to agenda
– compute the priority of I
– if there is no item I' in the chart or the agenda such that I' subsumes I
  then agenda := agenda $\cup \{I\}$
  else agenda := agenda

procedure consume-agenda
– while agenda is not empty
  – remove item I with highest priority from agenda
  – add item I to chart

procedure predict G
– for all rules $G' \leftarrow \Gamma$
  – if $\sigma =$ mgu(G, G') exists
    then add item $\sigma(G' \leftarrow \Gamma)$ to agenda

procedure add item C to chart
– chart := chart $\cup \{C\}$
– if C is a unit clause
  – for all items $H \leftarrow G \wedge \Omega$
    – if $\sigma =$ mgu(C, G) exists
      then add item $\sigma(H \leftarrow \Omega)$ to agenda
– if $C = H \leftarrow G \wedge \Omega$ is a non-unit clause
  – for all items $G' \leftarrow$
    – if $\sigma =$ mgu(G, G') exists
      then add item $\sigma(H \leftarrow \Omega)$ to agenda
  – predict G

Figure 1.7: Algorithm for best-first Earley deduction

there is increased non-determinism in the prediction step (cf. section 3.2), which can result in an explosion of the search space.

2. Incremental parsing is not readily supported for grammars with more powerful string operations.
3. Subsumption checking is needed to avoid prediction loops.
4. Preference information for heuristic guidance of the search is more frequently available in the bottom-up direction.

In order to overcome these problems, we shall present a bottom-up Earley deduction algorithm in chapter 3.

1.2.6 Constraints in Linguistic Deduction

Almost all contemporary grammatical formalisms can be characterised as constraint-based. The two interesting questions are what types of constraints they use, and how and when constraints are processed. This section provides a brief overview of these issues and gives pointers to the relevant literature.

1.2.6.1 Types of Constraints

Constraint Logic Programming was introduced by Jaffar and Lassez [Jaffar and Lassez, 1987], and has been generalised in Höhfeld and Smolka's constraint logic programming scheme [Höhfeld and Smolka, 1988], which allows to treat various constraint-based grammar formalisms in a unified framework. The framework has been applied to natural language grammar formalisms in [Smolka, 1992], [Frisch, 1993] and [Crouch, 1994].

The constraints allowed in Prolog, and in grammar formalisms based on Prolog such as DCG, are equality constraints between first order terms. Merging of constraints is performed by unification of first-order terms, and the unifying substitution is the merged constraint. Constraint entailment in the case of Prolog is subsumption of terms.

In most work on constraint-based grammars, first-order terms as data structures have been replaced by *feature structures*. Early developments in this direction are Functional Unification Grammar [Kay, 1979; Kay, 1984; Kay, 1985], Lexical-Functional Grammar [Kaplan and Bresnan, 1982], the PATR-II formalism [Shieber *et al.*, 1983], and the work on the formal foundations [Kasper and Rounds, 1986; Rounds and Kasper, 1986; Johnson, 1988; Backofen and Smolka, 1995]. Current systems make use of *sorted feature structures* whose formal foundations have been worked out by Smolka [Smolka, 1988] and Carpenter [Carpenter, 1992]. Examples are Head-Driven Phrase Structure Grammar (HPSG) [Pollard and Sag, 1987; Pollard and Sag, 1994], most current grammar formalisms (STUF [Bouma *et al.*, 1988;

Dörre and Seiffert, 1991], CUF [Dörre and Eisele, 1991; Dörre and Dorna, 1993], ALE [Carpenter, 1993a; Carpenter, 1993c; Carpenter and Penn, 1994], ALEP [Alshawi *et al.*, 1991; BIM-SEMA, 1993; Meylemans, 1994], ProFIT [Erbach, 1995], TDL [Krieger and Schäfer, 1994], TFS [Zajac, 1992] and others), and CLP languages such as LIFE [Aït-Kaci, 1991] or Oz [Smolka *et al.*, 1995].

Cyclic terms have first been introduced in Prolog II [Colmerauer, 1982] as rational trees, and are allowed in most current grammar formalisms and logic programming languages. Prolog II also introduced inequality constraints (`dif/2`), which are allowed in most current formalisms.

A wide variety of other types of constraints for different kinds of applications (e.g., linear equations) have been introduced in various CLP languages. A number of these are important for linguistic applications, most notably:

Finite Domains: Finite domains [van Hentenryck, 1989] have been introduced in the CLP language CHIP. A finite domain allows handling simple disjunctions without the creation of choice points. A finite domain variable can have a fixed finite set of possible values. When two finite domain variables are unified, the result is the intersection of their possible values, and fails if the intersection is empty.

Set Descriptions: Set descriptions and set constraints are widely used in linguistic descriptions, but have only recently been formalised and introduced in grammatical formalisms by Pollard and Moshier [Pollard and Moshier, 1990; Moshier and Pollard, 1994], by Carpenter [Carpenter, 1993b] and by Manandhar [Manandhar, 1993; Manandhar, 1994].

Linear Precedence Constraints: Linear precedence constraints have various uses in linguistic descriptions. Their most obvious use is the modelling of word order phenomena. Other uses are in natural language semantics in the description of temporal precedence relations and of underspecified quantifier scope. The logical foundations and a constraint solving algorithm for linear precedence constraints have been worked out by Manandhar [Manandhar, 1995].

Guarded Constraints: Guarded constraints are constraints whose execution is delayed until the precondition for their applicability is satisfied. Guarded Constraints can be used to attach goals to variables, and have been used in NLP in the implementation of HPSG principles and to integrate morphological constraints into an HPSG [Matiasek, 1994a; Trost and Matiasek, 1994]. They are also used in the CLG(n) *Constraint Logic Grammar* framework [Balari *et al.*, 1990; Damas *et al.*, 1991; Damas and Varile, 1992].

Tree Constraints Tree constraints extend linear precedence constraints by adding constraints on dominance, and permit the underspecified representa-

tion of trees through tree descriptions. A complete first-order axiomatisation for tree descriptions has been worked out in [Backofen *et al.*, 1995].

Boolean Logic: Prolog III [Colmerauer, 1987] introduces a solver for boolean constraints. This is put to a linguistic use by Lehner [Lehner, 1993; Lehner, 1994] who uses them to achieve the same effect as finite domains.

In the following, we will abstract away from the particular types of constraints, and talk about constraints quite generally, whether they are just equality constraints between first-order terms, or a more powerful constraint language.

1.2.6.2 Order of Constraint Checking

The previous discussion has only been concerned with the types of constraints for definite clauses, but left open the question at what point in a proof the constraints are checked, or assumed that they are checked immediately at every inference step, as in Prolog and its successors. It has also left open the question in which order different constraints should be checked.

Checking constraints immediately has the advantage of detecting failure as soon as possible. The drawback is that constraint checking may be computationally expensive, and redundant if the particular branch of the search fails anyway due to constraints that are cheaper to check.

A case study for this has been done in the framework of LFG, where constraints are divided into phrasal (c-structure) and functional (f-structure) constraints [Maxwell and Kaplan, 1993]. The experience has shown that processing is most efficient if phrasal constraints are evaluated first, and all functional constraints delayed — provided that some pieces of information are moved from the f-structure into the c-structure. Similar experiences have been made in the LILOG project, where all the constraints that build up logical forms have been delayed.

Since this tradeoff between early failure and the cost of constraint checking is highly dependent on the particular grammar and on the efficiency of the constraint checking algorithms, we will not pursue the question further in this thesis, but just note that selective delaying of constraints can be applied as a means to fine-tune the performance of a system.

In the actual implementation, we have chosen to check constraints as soon as possible and to minimise the cost of checking equality constraints between sorted feature structures by compiling them into Prolog terms, and using Prolog's built-in term unification (cf. chapter 5).

Uszkoreit [Uszkoreit, 1991] proposes another model which makes use of statistical information for controlling the order of constraint checking in order to optimise performance. Conjuncts are given priority if they have a high *failure potential,* and disjuncts if they have a high *success potential.* The reason is that a conjunction fails if one of its conjuncts fails, and a disjunction is satisfied if one

of the disjuncts is satisfied. Since this model applies internally to the constraint solver, it can easily be added to the processing algorithms proposed in this thesis.

1.2.6.3 Coroutining

Coroutining is an important technique to provide for the handling of complex constraints such as relational dependencies (e.g., concatenation constraints). Coroutining ensures that these constraints are only checked when certain variables are sufficiently instantiated to guarantee termination. This is important in cases where no good ordering of the goals in a clause can be found, especially when it is not known in advance which arguments of a predicate are its input and its output arguments.

Coroutining was introduced in Prolog II with the `freeze/2` construct, which allows to delay a goal until a variable becomes instantiated. In Sicstus Prolog, the condition can be that a particular variable is either instantiated, or instantiated to a ground term, or known to be equal to or different from another variable. Conjunctions and disjunctions of these conditions are possible.

In LIFE [Aït-Kaci and Podelski, 1991], coroutining is achieved by treating functions as passive constraints, i.e., functional expressions in a ψ-term are only evaluated when their arguments become sufficiently instantiated to determine subsumption of the arguments specified in the function's definition. Otherwise the function residuates, i.e., waits for further instantiation [Smolka, 1993].

The most general form of coroutining is known in logic programming languages under the name of *guarded rules* or *guarded constraints*, where the execution of a rule is delayed until the specified conditions for execution (the guard) are satisfied (by the instantiation of variables) [Aït-Kaci and Podelski, 1994; Smolka and Treinen, 1994].

Pfahringer and Matiasek make use of constraint logic programming for parsing of HPSG grammars.[30] In their approach, the principles of HPSG are attached as constraints to variables, and are checked when these variables become instantiated by unification. Only a few relations serve as generators of structures, to which then the principles will apply to filter out the ill-formed ones. In this algorithm, processing consists almost entirely of constraint checking.

Their algorithm is implemented with a Prolog extension known as attributed variables, which allow user-defined unification and the attachment of arbitrary constraints to logic variables [Holzbaur, 1992; Pfahringer, 1992]. Attributed variables are a generalisation of metaterms known from logic programming systems such as Eclipse.

[30] [Matiasek, 1993; Matiasek and Heinz, 1993; Pfahringer and Matiasek, 1992; Matiasek, 1994a; Matiasek, 1994b]

1.2.7 Search Strategy

The space of possible search strategies is delimited by pure depth-first search at one end, and breadth-first search at the other.

Breadth-first search is a useful search strategy because it is guaranteed to find all solutions. However, it has a high computational cost compared to depth-first search. Breadth-first search is useful for applications where the search can be terminated after one solution has been found, since — unlike depth-first search — it is guaranteed to find that solution. Its usefulness for linguistic deduction is very limited because in general the purpose of linguistic deduction is to enumerate *all* solutions to a given query (all analyses in case of parsing, or all paraphrases in case of generation). In cases where *all* solutions need to be enumerated, breadth-first search brings no advantage because it runs into the same termination problems as depth-first search because there is no way to know when all solutions have been found, so the search may go on indefinitely. The same kind of termination problems can arise if no solutions exist.

Depth-first search, on the other hand, can be much more efficient, and is therefore often chosen as a search strategy for NLP.

In the framework of Earley deduction, it is easily possible to perform depth-first search, breadth-first search or any mixture between the two. This property is pointed out in [Kay, 1980] who refers to chart parsing as an *algorithm schema,* which is turned into an algorithm by choosing a rule invocation strategy (top-down, bottom-up or directed) and a search strategy. Kay introduces an agenda as a data structure for the implementation of a search strategy. An agenda is a set of pending parsing tasks (addition of items to the chart), which are ordered according to a priority. Many chart parsing systems make use of an agenda for implementing a search strategy. The question of search strategies has been explored in a system for experimenting with parsing strategies that allows the definition of strategies through assigning priorities to parsing tasks (combinations of active and passive items) [Erbach, 1991a; Erbach, 1991b], based on properties of the items such as length of the constituent, grammatical category, probability of the rule etc. The problem with the system is that assigning priorities to parsing tasks on the basis of such superficial properties of the items does not give any real advantage in practice. In chapter 4, we define a notion of preference value that can be used as a better basis for assignment of priorities.

Uszkoreit [Uszkoreit, 1991] also proposes a model that mixes depth-first and breadth-first search. In this model, numeric preference values (chosen from a predefined interval) are associated with disjuncts of a disjunction. Whenever a disjunction is processed, all disjuncts whose preference is above a certain threshold value are processed in parallel, and for those below the threshold, a choice point is created which will be used on backtracking. Purely breadth-first search is enforced by setting the threshold to the minimal preference value, and purely depth-first search is the threshold is the maximal preference value.

At the current state of the art, the algorithms for implementing different search strategies are well-known, but good criteria for assigning priorities to non-deterministic choices are still an area of active research.

1.2.8 Bidirectional NLP Algorithms

An NLP algorithm is bidirectional if it can be used for both parsing and generation. Bidirectional algorithms can be classified along the following two dimensions into four classes [Neumann, 1994b].

1. Does the system use the same grammar for both directions?
2. Is the same process used for both directions?

The approach we take in this thesis uses both the same algorithm and the same process for parsing and generation.

The Essential Arguments Algorithm makes a definite clause grammar usable for both parsing and generation by transforming it into two different logic programs — one for each direction [Strzalkowski, 1991]. The algorithm is based on re-ordering of goals at compile time (the transformation stage) in order to ensure that variables are sufficiently instantiated when a goal is called to guarantee termination.

The head-driven algorithms discussed in section 1.2.3 are bidirectional in the sense that they can be used for both parsing and generation.

1.2.8.1 Algorithms Based on Charts

Shieber [Shieber, 1988b] and Gerdemann [Gerdemann, 1991] have proposed algorithms that use Earley deduction for both parsing and generation in a bidirectional setting; the first truly uniform bidirectional algorithm in which parsing and generation can be interleaved has been presented in Neumann's dissertation [Neumann, 1994b].

Kay has developed a bottom-up chart-based generator, which makes use of semantic indexing [Kay, 1993].

Hashida presents a framework in which parsing and generation are processes which emerge from one underlying algorithm [Hashida, 1994]. Hashida combines chart-based methods with an activation network that controls the instantiation and reduction process. A direct comparison with Earley deduction is made complicated by the facts that Hashida uses a very non-standard notation which is not always easily relatable to the standard formulations, and that his formalism is under constant development, currently in order to replace the activation network with a probabilistic processing regime.

1.3 Conclusion

To conclude this chapter, we summarise our preferred choices in the five areas discussed above where linguistic deduction algorithms can be varied.

Direction of processing: A bottom-up algorithm is preferred because of its potential for data-driven and robust linguistic deduction. Wherever possible, a directed algorithm that combines the benefits of bottom-up and top-down processing should be chosen.

Selection function: The choice of a selection function is not really the focus of this thesis. For simplicity, we assume that the goals of a clause have already been ordered, and always use the first unblocked goal.

Memoing: We choose memoing in the form of Earley deduction, which has all the advantages that chart parsing has for NL parsing, and is especially well suited for search strategies such as best-first processing.

Constraints: We use the constraint language defined in section 1.1.1 and check all constraints immediately.

Search Strategy: We choose to employ a best-first search strategy, based on a probabilistic preference model elaborated in section 4.1.

Before presenting a processing model that instantiates these choices in chapter 3 (bottom-up Earley deduction), we will detour and show in chapter 2 how principle-based grammars such as GB and HPSG can be brought into a rule-based form that can be processed efficiently by the proposed deduction system.

Chapter 2

From Principle-Based Grammars to Rule-Based Grammars

2.1 Principle-Based Versus Rule-Based Grammars

This chapter is concerned with the claim that there are two different kinds of grammatical theories: rule-based grammars in which rules describe how grammatical constituents are *constructed* and principle-based grammars in which universal principles *constrain* the possible constituents.

First, we define more precisely what is meant by principle-based and rule-based grammar.

Principle-based grammar A set of (universal or language-specific) principles must hold of all constituents described by a grammar. A few very general rules may exist for forming the objects to which the principles are applied. The prime example for this kind of approach is Government-Binding Theory (GB) [Chomsky, 1981; Chomsky, 1986; Chomsky, 1993].

Rule-based grammar Every constituent must instantiate one rule of the grammar. Examples of rule based grammatical theories/formalisms are LFG, DCG, and PATR.

A reason for the move from rule-based to principle-based grammars is the desire to provide a theory of language that exhibits explanatory adequacy, not just observational adequacy (accounting for the observed data) or descriptive adequacy

(assignment of the right structural descriptions to sentences). Principle-based grammars achieve explanatory adequacy by showing that a large variety of phrase structures can be predicted from the interaction of a smaller number of general principles.

Government-Binding Theory is the best example of a principle-based syntactic theory, but the use of general principles also plays an increasing role in grammar theories arising from the tradition of phrase structure grammars. *Generalised Phrase Structure Grammar* (GPSG) [Gazdar *et al.*, 1985] uses phrase structure rules which are augmented with principles such as the *Head Feature Principle*. In *Head-Driven Phrase Structure Grammar* (HPSG) [Pollard and Sag, 1987; Pollard and Sag, 1994] phrase structure rules are largely abandoned, and the grammar rests on the lexicon, and on principles which constrain the possible structures.

A big advantage of the principle-based approach is the compactness of the representation of linguistic knowledge. To obtain the effect of k principles with n_j degrees of freedom, which can be stated in $O(n_1 + n_2 \ldots + n_k)$ lines of code, a rule-based approach has to use $O(n_1 * n_2 \ldots * n_k)$ rules (cf. [Berwick, 1991]).

While it is possible to process principle-based grammars directly in a type inference system such as CUF or TFS, such direct processing suffers from serious efficiency problems, due to the fact that many inference steps must be performed at runtime. In order to overcome these efficiency problems, many implementations of HPSG grammars often use rule-based formalisms such as ALE or ALEP, and take one of the following approaches to the handling of principles:

- the grammar rules are written in such a way that they respect the principles of the grammar, or
- the principles are added to the grammar rules as procedural attachments.

The first approach has the disadvantage that the compactness of the grammatical specification permitted by principle-based formalisms is lost. The second approach has the disadvantage that it increases the amount of processing needed at runtime because the principles often have disjunctive formulations for different types of structure.

We will present a formalism which permits a principle-based statement of the grammar, and uses program transformation techniques (partial deduction) in order to derive a rule-based grammar. This happens by specialising the principles of the grammar to particular phrase types and eliminating as much disjunction as possible at this stage. In the most extreme case, this step can transform the grammar into its disjunctive normal form (DNF), but in practice DNF is often either undesirable or impossible. It may be undesirable when it leads to a very large number of phrase structure rules, and impossible when it would lead to an infinite number of rules because of the use of recursively defined relations (such as `append/3`) in the principles of the grammar. Therefore, the outcome of the

program transformation is a grammar with a number of phrase structure rules to which some constraints and goals are attached which cannot be reduced at compile time[1] because they need to be instantiated during the processing to allow a terminating (and often deterministic) computation.

The converse operation, transformation of rule-based grammars to principle-based ones, may not be possible because there may be no commonalities between the rules, i.e. no generalisations about these rules that could be expressed by principles. Even if there are commonalities, their detection is beyond the capabilities of today's grammar learning systems.

In the following sections, we will use examples from GB and HPSG to illustrate the transformation of a principle-based grammar into a rule-based grammar by means of partial deduction techniques.

Since pure principle-based and pure rule-based grammars are not strict opposites, but rather the endpoints on a continuum, we will investigate what practical gain in terms of runtime efficiency can be achieved by transforming a grammar that is closer to the principle-based end of the scale into one that is near the rule-based end. As the starting point for this investigation, we use a grammar of HPSG that was originally encoded in the ALE formalism.

2.2 Partial Deduction

Partial deduction is a program transformation technique for logic programs that takes one logic program as input and returns an equivalent logic program in which some of the goals are replaced by their expansions, i.e., the antecedent of a clause whose consequent matches the goal. Partial deduction is sometimes also referred to as *partial execution* or *partial evaluation.* We prefer the term *partial deduction* in order to emphasise that it is a deduction step and as such it can have more than one solution. It can be used to perform certain deduction steps at compile time, in order to arrive at a larger program that can be executed more efficiently because it requires less deduction at runtime.[2] The inference rule for partial deduction (a resolution step) is shown in rule (2.1); σ is the merged constraint of B and B' (in case of Prolog the unifying substitution). The inference rule can apply recursively to its own output, so that termination cannot be guaranteed in the general case.

[1] By *compile time,* we mean the step of loading a logic program or declarative grammar and transforming it to an internal representation. By *runtime* we mean the step when the internal representation is executed or interpreted to prove goals (in particular to parse and generate sentences).

[2] For Logic Programming, partial deduction fills the role that is filled by macros in other programming languages. Partial deduction is a basic tool for logic programmers, and has been described in an overview article by D.H. Warren [Warren, 1992]. Partial deduction in Prolog systems can be done at compile time by defining clauses for `term_expansion/2`, or for `goal_expansion/2` in Sicstus Prolog 3.1.

$$\frac{\begin{array}{c} A \leftarrow \Gamma\ B\ \Delta \\ B' \leftarrow \Theta \end{array}}{\sigma(A \leftarrow \Gamma\ \Theta\ \Delta)} \tag{2.1}$$

In order to apply partial deduction for program transformation, a decision must be made which goals B are to be replaced by their expansions. In order of increasing generality, there are three possibilities for making this decision: (1) annotating individual goals to be expanded, (2) stating that goals which satisfy certain conditions (e.g., those with a particular functor) are expanded, and (3) providing a mechanism which decides automatically which goals can be expanded without running into termination problems. We prefer to make use of the second option in order to retain some control over the partial deduction process, and allow the first option for specialised cases (cf. section 5.3.1.2 for a description of the implementation).

In NLP, partial deduction has been employed to compile a parser and a grammar, into an efficient executable parsing algorithm for that particular grammar. Likewise, a generator and a grammar can be compiled into an efficient executable generation algorithm. This is done by performing the partial deduction step on the clauses that make calls to the rules of the grammar or the lexicon. In [Pereira and Shieber, 1987, pp. 172-185], examples for compilation of DCG and parsers (as interpreters) into top-down and left-corner parsers are given.

2.2.1 Partial Deduction Example: DCG

As a simple illustration of partial deduction, we show how to transform a top-down parser and a (definite clause) grammar into an executable program. The relation `concat/3` is defined by means of difference lists to avoid unneeded non-determinism (cf. page 17).

All predicates except `parse/2` are expanded at compile time by partial deduction. The output of applying partial deduction to the program (parser and grammar) in figure 2.1 is the equivalent program shown in figure 2.2, which consists only of clauses for `parse/2`.[3]

Note that partial deduction can suffer from the same problems with termination and infinite sets of solutions as Prolog's top-down deduction strategy, since the inference rule for partial deduction is nothing but a resolution step that is performed at compile time.

The solutions to this problem are the same as in practical Prolog programming: ordering goals in such a way that the solutions of one goal ensure that the next

[3]Of course the resulting program can still be optimised to make use of the fact that Prolog indexes clauses by functor; instead of clauses of the form `parse(Category,String)`, indexing is exploited by making the category the main functor and flattening the term: for example `parse(np(...),S0-S)` should be represented more efficiently as `np(...,S0,S)`.

```
% The top-down parser
parse(Cat,String) ←
        word_list(Word,String) ∧
        word(Cat,Word).
parse(Cat,String) ←
        rule(Cat,RHS) ∧
        parse_rhs(RHS,String).

parse_rhs(⟨Cat1|Cats⟩,String) ←
        concat(Prefix,Rest,String) ∧
        parse(Cat1,Prefix) ∧
        parse_rhs(Cats,Rest).
parse_rhs(⟨ ⟩,Elist) ←
        empty_list(Elist).

% Auxiliary predicates
concat(A-B,B-C,A-C).
word_list(Word,⟨Word|R⟩-R).
empty_list(A-A).

% The grammar
rule(s,⟨np(Num,Pers),vp(Num,Pers)⟩).
rule(np(Num,3),⟨det(Num),n(Num)⟩).
rule(np(Num,Pers),⟨pron(Num,Pers)⟩).
rule(vp(Num,Pers),⟨vi(Num,Pers)⟩).
rule(vp(Num,Pers),⟨vt(Num,Pers),np(_,_)⟩).

% The lexicon
word(pron(sg,third),she).
word(pron(pl,first),we).
word(det(sg),a).
word(det(_),the).
word(n(sg),house).
word(n(pl),carpenters).
word(vi(sg,third),burns).
word(vt(pl,_),build).
```

Figure 2.1: Grammar and top-down parser as input to partial deduction

```
parse(s,A-B) ←
        parse(np(C,D),A-E) ∧
        parse(vp(C,D),E-B).
parse(np(A,3),B-C) ←
        parse(det(A),B-D) ∧
        parse(n(A),D-C).
parse(np(A,B),C-D) ←
        parse(pron(A,B),C-D).
parse(vp(A,B),C-D) ←
        parse(vi(A,B),C-D).
parse(vp(A,B),C-D) ←
        parse(vt(A,B),C-E) ∧
        parse(np(F,G),E-D).

parse(pron(sg,third),⟨ she | A ⟩-A).
parse(pron(pl,first),⟨ we | A ⟩-A).
parse(det(sg),⟨ a | A ⟩-A).
parse(det(A),⟨ the | B ⟩-B).
parse(n(sg),⟨ house | A ⟩-A).
parse(n(pl),⟨ carpenters | A ⟩-A).
parse(vi(sg,third),⟨ burns | A ⟩-A).
parse(vt(pl,A),⟨ build | B ⟩-B).
```

Figure 2.2: Output of partial deduction on grammar and top-down parser

goal is properly instantiated.

In the example, this problem arises in case of the recursively defined predicate `parse_rhs/2`. Partial deduction applied to `parse_rhs/2` gives an infinite number of solutions unless the first argument is instantiated with a proper list.[4]

As the next example (figure 2.3), we apply partial deduction to a left-corner parser and the same grammar and lexicon as above to obtain a specialised left-corner parser for the grammar. The list manipulation predicates `empty_list/1`, `word_list/2` and `concat/3` are defined as in the previous example (figure 2.1). This time all predicates except `word/2` and `lc/3` are expanded by partial deduction.

```
parse(GoalCat,String) ←
        word_list(Word,WordList) ∧
        concat(WordList,RestList,String) ∧
        word(LexCat,Word) ∧
        lc(LexCat,GoalCat,RestList).

lc(Cat,Cat,String) ←
        empty_list(String).
lc(SubCat,GoalCat,String) ←
        rule(Cat,⟨SubCat|Rest⟩) ∧
        concat(Prefix,Suffix,String) ∧
        parse_rhs(Rest,Prefix) ∧
        lc(Cat,GoalCat,Suffix).

parse_rhs(⟨Cat1|Cats⟩,String) ←
        concat(Prefix,Rest,String) ∧
        parse(Cat1,Prefix) ∧
        parse_rhs(Cats,Rest).
parse_rhs(⟨ ⟩,Elist) ←
        empty_list(Elist).
```

Figure 2.3: Left-corner parser as input to partial deduction

The output of the partial deduction procedure, which is a left-corner parser specialised for the same grammar, is shown in figure 2.4.

A similar exercise could be performed for other algorithms (such as a shift-reduce parser or a semantic-head driven generator), but the two above examples

[4] A list is proper iff either it is the empty list, or it consists of a first element and a rest, and the rest is a proper list.

```
parse(A,⟨B|C⟩-D) ←
        word(E,B) ∧
        lc(E,A,C-D).
lc(A,A,B-B).
lc(np(A,B),C,D-E) ←
        parse(vp(A,B),D-F) ∧
        lc(s,C,F-E).
lc(det(A),B,C-D) ←
        parse(n(A),C-E) ∧
        lc(np(A,3),B,E-D).
lc(pron(A,B),C,D-E) ←
        lc(np(A,B),C,D-E).
lc(vi(A,B),C,D-E) ←
        lc(vp(A,B),C,D-E).
lc(vt(A,B),C,D-E) ←
        parse(np(F,G),D-H) ∧
        lc(vp(A,B),C,H-E).
```

Figure 2.4: Output of partial deduction applied to grammar and left-corner parser

should suffice to illustrate the basic idea of partial deduction. We turn therefore now to the central subject of this chapter, namely the application of partial deduction in order to derive rule-based grammars from principle-based ones.

2.2.2 Partial Deduction Applied to GB

Government-and-Binding (GB) Theory [Chomsky, 1981] is a prime example of a principle-based grammar. Unfortunately, there is no standard formalisation of GB. We will use a small example from a paper by Crocker and Lewin, in which they claim that principle-based grammars are fundamentally different from rule-based grammars. The following quotation summarises their position [Crocker and Lewin, 1992, p. 508].

> While deductive parsing techniques are well-understood for traditional rule-based grammars, they are rather more elusive for current principle-based or constraint-based grammars. [...] we argue that a major source of difficulty arises from a fundamental difference in the way such grammars should be axiomatised: While rule-based grammars typically consist of a set of *sufficient* structure-generating axioms, principle-based grammars are more naturally expressed as a set of *necessary* 'structure-

I. Two sets of nodes:
   (a) The set T of all terminals.
   (b) The set NT of all non-terminals.

II. The set B of branches:
   (a) if $X, Y, Z \epsilon NT$, then $[_X YZ]\ \epsilon B$
   (b) if $X \epsilon NT$, $Y \epsilon T$, then $[_X Y]\ \epsilon B$
   (c) there is nothing else ϵB

III. The set PB of proper branches:
   (a) $\alpha \epsilon PB$ iff $\alpha \epsilon B$, and
   (b) α meets all necessary conditions in [figure 2.6].

IV. The set Tr of well-formed trees:
   (a) if $\alpha = [_X Y]\ \epsilon PB$, then $\alpha \epsilon Tr$
   (b) if $A_t, B_t \epsilon Tr$ and $[_X AB]\ \epsilon PB$, then $X_t = [_X A_t B_t]\ \epsilon Tr$.

Figure 2.5: Fundamental definitions of Crocker and Lewin's GB specification

> licencing' conditions which in essence rule out ill-formed structures rather than generating new ones.

The specification in figure 2.5 and the GB principles in figure 2.6 are taken from Crocker and Lewin's paper (page 511) where they are contrasted with a rule-based grammar covering the same tiny language fragment. It serves as an example for illustrating the *difference* between principle-based and rule-based grammars. We will use it as a basis for compilation from a principle-based grammar into a rule-based grammar by making use of partial deduction techniques.

In figure 2.7 we translate the fundamental definitions (from figure 2.5) to definite clauses, and in figure 2.8 the principles and lexicon (from figure 2.6). These definite clauses form the basis of the partial deduction exercise. Note that the example makes use of finite domains to encode underspecification of bar levels; the corresponding finite domain consists of the values 0, 1 and 2. If such a finite domain encoding were not used, we would end up with a larger number of rules in which the bar levels are fully specified.

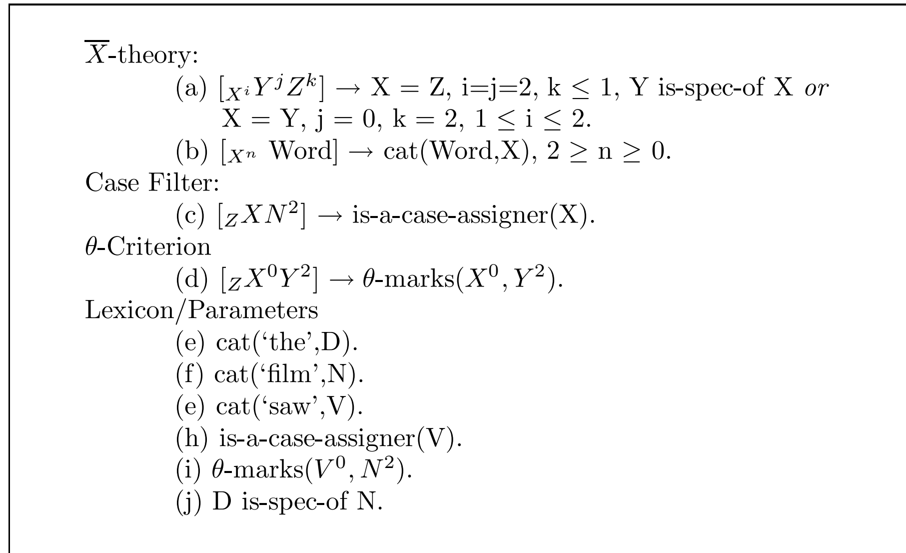
$\overline{X}$-theory:
(a) $[_{X^i} Y^j Z^k] \rightarrow$ X = Z, i=j=2, k $\leq$ 1, Y is-spec-of X *or*
X = Y, j = 0, k = 2, 1 $\leq$ i $\leq$ 2.
(b) $[_{X^n}$ Word$] \rightarrow$ cat(Word,X), 2 $\geq$ n $\geq$ 0.
Case Filter:
(c) $[_Z X N^2] \rightarrow$ is-a-case-assigner(X).
θ-Criterion
(d) $[_Z X^0 Y^2] \rightarrow \theta$-marks$(X^0, Y^2)$.
Lexicon/Parameters
(e) cat('the',D).
(f) cat('film',N).
(e) cat('saw',V).
(h) is-a-case-assigner(V).
(i) θ-marks(V^0, N^2).
(j) D is-spec-of N.

Figure 2.6: Crocker and Lewin's specification of GB principles and lexicon

The partial deduction proceeds by expanding all goals except `proper_branch/1`.[5]

The output of our partial deduction system (cf. section 5.3) is shown as definite clauses in figure 2.10. Written in more conventional grammar rule notation, these proper branches are shown below in the ruleset (2.2) as five grammar rules (two phrasal rules and three lexical entries).

$$
\begin{array}{l}
N^2 \rightarrow D^2 N^{(0 \vee 1)} \\
V^{(1 \vee 2)} \rightarrow V^0 N^2 \\
N \rightarrow [film] \\
D \rightarrow [the] \\
V \rightarrow [saw]
\end{array}
\tag{2.2}
$$

The application of the GB principles greatly reduces the number of possible rules. When the definition for `branch/1` if expanded without recourse to principles, there are 36 possible results, most of which are in violation of the principles of GB.

[5]Partial deduction is applied to reduce all calls to the predicates `branch/1, theta_marks/2, is_a_case_assigner/1, cat/2, x_bar_theory/1, theta_criterion/1, case_filter/1, is_spec_of/2, branch/1, conditions/1, word_to_list/2`, and `concat/3`.

```
top > ⟨node⟩.
node intro ⟨cat,bar,string⟩.
bar fin_dom ⟨0,1,2⟩.

non_terminal(n).
non_terminal(d).
non_terminal(v).

terminal(the).
terminal(film).
terminal(saw).
```

$$\text{branch}\left(\left\langle \begin{bmatrix}\text{cat:} & \text{M}\\ \text{string:} & \text{SM}\end{bmatrix}, \begin{bmatrix}\text{cat:} & \text{L}\\ \text{string:} & \text{SL}\end{bmatrix}, \begin{bmatrix}\text{cat:} & \text{R}\\ \text{string:} & \text{SR}\end{bmatrix} \right\rangle\right) \leftarrow$$

```
        non_terminal(M) ∧
        non_terminal(L) ∧
        non_terminal(R) ∧
        concat(SL,SR,SM).
```

$$\text{branch}\left(\left\langle \begin{bmatrix}\text{cat:} & \text{M}\\ \text{string:} & \text{S}\end{bmatrix}, D \right\rangle\right) \leftarrow$$

```
        non_terminal(M) ∧
        terminal(D) ∧
        word_to_list(D,S).

proper_branch(X) ←
        branch(X) ∧
        conditions(X).

conditions(X) ←
        x_bar_theory(X) ∧
        case_filter(X) ∧
        theta_criterion(X).
```

Figure 2.7: Fundamental definitions of GB as a logic program

$$\text{x_bar_theory}\left(\left\langle \begin{bmatrix}\text{cat: X}\\ \text{bar: 2}\end{bmatrix}, \begin{bmatrix}\text{cat: Y}\\ \text{bar: 2}\end{bmatrix}, \begin{bmatrix}\text{cat: Z}\\ \text{bar: K}\end{bmatrix}\right\rangle\right) \leftarrow$$
$$\text{K} = 0 \text{ or } 1 \wedge \text{X} = \text{Z} \wedge \text{is_spec_of(Y,X)}.$$

$$\text{x_bar_theory}\left(\left\langle \begin{bmatrix}\text{cat: X}\\ \text{bar: I}\end{bmatrix}, \begin{bmatrix}\text{cat: Y}\\ \text{bar: 0}\end{bmatrix}, \begin{bmatrix}\text{cat: Z}\\ \text{bar: 2}\end{bmatrix}\right\rangle\right) \leftarrow$$
$$\text{I} = 1 \text{ or } 2 \wedge \text{X} = \text{Y}.$$

$$\text{x_bar_theory}\left(\left\langle \begin{bmatrix}\text{cat: Cat}\\ \text{bar: Bar}\end{bmatrix}, Word\right\rangle\right) \leftarrow$$
$$\text{cat(Word,Cat)} \wedge \text{Bar} = 0 \text{ or } 1 \text{ or } 2.$$

$$\text{case_filter}\left(\left\langle _, X, \begin{bmatrix}\text{cat: n}\\ \text{bar: 2}\end{bmatrix}\right\rangle\right) \leftarrow$$
$$\text{is_a_case_assigner(X)}.$$

$$\text{case_filter}\left(\left\langle _, X, \begin{bmatrix}\text{cat: n}\\ \text{bar: } 0 \vee 1\end{bmatrix}\right\rangle\right).$$

case_filter(⟨ _,X,cat:Cat ⟩) ← dif(Cat,n).
case_filter(⟨ _,_ ⟩).
theta_criterion(⟨ _, X&bar:0, Y&bar:2 ⟩) ← theta_marks(X,Y).
theta_criterion(⟨ _,bar:(1 or 2),bar:(0 or 1) ⟩).
theta_criterion(⟨ _,_ ⟩).

cat(the,d).
cat(film,n).
cat(saw,v).
is_a_case_assigner(cat:v).

$$\text{theta_marks}\left(\begin{bmatrix}\text{cat: v}\\ \text{bar: 0}\end{bmatrix}, \begin{bmatrix}\text{cat: n}\\ \text{bar: 2}\end{bmatrix}\right).$$

is_spec_of(d,n).

Figure 2.8: GB principles and lexicon as logic program

```
concat(A-B,B-C,A-C).
word_to_list(Word,⟨Word|R⟩-R).

rec(Node) :-
        proper_branch(⟨Node,_⟩).
rec(Node) :-
        proper_branch(⟨Node, LeftDtr, RightDtr⟩) ∧
        rec(LeftDtr) ∧
        rec(RightDtr).
```

Figure 2.9: Auxiliary predicates for GB parsing

$$\text{proper_branch}\left(\left\langle \begin{bmatrix} \text{cat:} & \text{n} \\ \text{bar:} & 2 \\ \text{string:} & \text{A-C} \end{bmatrix}, \begin{bmatrix} \text{cat:} & \text{d} \\ \text{bar:} & 2 \\ \text{string:} & \text{A-B} \end{bmatrix}, \begin{bmatrix} \text{cat:} & \text{n} \\ \text{bar:} & 0 \vee 1 \\ \text{string:} & \text{B-C} \end{bmatrix} \right\rangle\right).$$

$$\text{proper_branch}\left(\left\langle \begin{bmatrix} \text{cat:} & \text{v} \\ \text{bar:} & 2 \vee 1 \\ \text{string:} & \text{A-C} \end{bmatrix}, \begin{bmatrix} \text{cat:} & \text{v} \\ \text{bar:} & 0 \\ \text{string:} & \text{A-B} \end{bmatrix}, \begin{bmatrix} \text{cat:} & \text{n} \\ \text{bar:} & 2 \\ \text{string:} & \text{B-C} \end{bmatrix} \right\rangle\right).$$

$$\text{proper_branch}\left(\left\langle \begin{bmatrix} \text{cat:} & \text{n} \\ \text{string:} & \langle \text{film} \mid \text{A} \rangle\text{-A} \end{bmatrix}, film \right\rangle\right).$$

$$\text{proper_branch}\left(\left\langle \begin{bmatrix} \text{cat:} & \text{d} \\ \text{string:} & \langle \text{the} \mid \text{A} \rangle\text{-A} \end{bmatrix}, the \right\rangle\right).$$

$$\text{proper_branch}\left(\left\langle \begin{bmatrix} \text{cat:} & \text{v} \\ \text{string:} & \langle \text{saw} \mid \text{A} \rangle\text{-A} \end{bmatrix}, saw \right\rangle\right).$$

Figure 2.10: Output of partial deduction applied to GB (as logic program)

If these were used to enumerate possible analysis trees for a given sentence, the result would be a vast number of trees. The application of partial deduction can reduce this number significantly already at compile time.

This small example of partial deduction demonstrates clearly that it is possible to compile a principle-based grammar into a rule-based grammar. The resulting rule-based grammar is of course redundant and does not capture generalisations that can be expressed in the principle-based format.

Other approaches to principle-based GB-parsing in [Berwick *et al.*, 1991] (e.g., [Fong, 1991; Johnson, 1991; Stabler, 1990] employ a covering phrase structure grammar to build up structures, and use the principles to rule out any invalid derivations.

We turn our attention now to Head-Driven Phrase Structure Grammar (HPSG), argue that it is a principle-based grammar according to the definition given above and show how partial deduction can be used in order to transform a larger HPSG grammar into a rule-based grammar to which the standard parsing and generation algorithms can be applied.

2.3 HPSG as a Principle-Based Grammar

HPSG has been labelled an *information-based* grammatical theory [Pollard and Sag, 1987], a characterisation which emphasises the fact that HPSG is concerned with providing a theory how language is used to convey information, rather than just characterising the set of well-formed sentences. More interesting for our purposes is the characterisation of HPSG as a *constraint-based* theory of language. The characterisation of a theory as constraint-based is orthogonal to the distinction rule-based versus principle-based. Both rule-based and principle-based theories can be based on constraints: in the former case the constraints apply only to one rule, whereas in the latter case the constraints are principles that apply to every structure that is allowed by the grammar. In that sense, we would call HPSG a principle-based and a constraint-based theory, whereas LFG or DCG are rule-based and constraint-based. Context-free grammars are rule-based but not constraint-based, whereas it is hard to conceive of a grammatical formalism that is principle-based, but not constraint-based in the widest sense.

Since this thesis will be exclusively concerned with constraint-based grammars (cf. section 1.2.6), it is the principle-based versus rule-based distinction that really matters.

2.3.1 Formalisations of HPSG

In this section, we review the formalisation of HPSG given in the two HPSG books ([Pollard and Sag, 1987; Pollard and Sag, 1994]), give a formalisation of HPSG as

definite clauses in the spirit of work done in the STUF and CUF formalisms, and show the equivalence of the two formalisations.

Pollard and Sag [Pollard and Sag, 1987, p. 44] give the following formalisation of HPSG ($P_1 \ldots P_n$ are universal principles, $P_{n+1} \ldots P_{n+m}$ are language-specific principles, $L_1 \ldots L_p$ are lexical signs of a language, and $R_1 \ldots R_q$ its grammar rules).

$$\text{English} = P_1 \wedge \ldots \wedge P_{n+m} \wedge (L_1 \vee \ldots \vee L_p \vee R_1 \vee \ldots \vee R_q) \tag{2.3}$$

In later HPSG work, the disjunction of lexical entries and rule schemata is made into the ID PRINCIPLE, which states that "every headed phrase must satisfy exactly one of the ID schemata" [Pollard and Sag, 1994, p. 38].

In one tradition of processing models for HPSG based on *type deduction,* (2.3) is treated as a constraint on the sort *sign.* The basic idea in this approach is that every feature structure of sort *sign* must satisfy all principles of the grammar and exactly one of the ID schemata or a lexical entry. In this approach, there is a distinction between *sorts* and *types.* Sorts are the defined in the usual sense, but types can be complex properties, such as the type *sign_of_english.*

Linguistic processing in this approach proceeds as type deduction. Examples are the Typed Feature Structure system TFS [Zajac, 1992; Emele and Zajac, 1990], and to a certain extent ALE [Carpenter, 1993a].[6]

In this thesis, we will use a different formalisation of HPSG that is closer to mainstream logic programming, namely we will represent an HPSG by definite clauses. The primary motivation for this is the fact that we want to apply program transformation techniques known from logic programming to HPSG in order to turn it from a principle-based into a rule-based one, for which more efficient processing algorithms are known to exist.

In order to arrive at a definite clause formalisation of (2.3), we interpret the equality sign (=) as logical equivalence, and, the connectives as conjunction and disjunction, and the principles, lexical entries and rules as one-place predicates. The resulting formula of first-order logic is shown in (2.4).

$$\begin{aligned} &\forall x[\text{sign_of_english}(x) \leftrightarrow \\ &\qquad P_1(x) \wedge \ldots \wedge P_{n+m}(x) \wedge \\ &\qquad (L_1(x) \vee \ldots \vee L_p(x) \vee R_1(x) \vee \ldots \vee R_q(x))] \end{aligned} \tag{2.4}$$

Logic programming is based on a closed-world assumption, i.e., everything that is not provable from a logic program is false (negation by failure). Consequently, if a proposition P is implied by a number of assumptions $A_1 \ldots A_n$, then the truth of P under the closed world assumption also implies the truth of $A_1 \wedge \ldots \wedge A_n$

[6] For a thorough discussion of the issues involved in defining typed feature formalisms for HPSG, see [Meurers, 1994; Götz and Meurers, 1995].

[Lloyd, 1984, p. 74] [Shepherdson, 1984]. So, if we see our HPSG example in a logic programming context, the biconditional (2.4) can be substituted by a Horn clause with a simple conditional (2.5).

$$\begin{aligned} \forall x[\text{sign_of_english}(x) \leftarrow & \\ & P_1(x) \wedge \ldots \wedge P_{n+m}(x) \wedge \\ & (L_1(x) \vee \ldots \vee L_p(x) \vee R_1(x) \vee \ldots \vee R_q(x))] \end{aligned} \tag{2.5}$$

What is still missing from this formulation is the fact that any substructures of X which are of the sort *sign* must also satisfy the predicate `sign_of_english/1`. Due to the appropriateness specification of HPSG, structures of sort *sign* can only occur as values of the features HEAD-DTR, ADJUNCT-DTR, MARKER-DTR, FILLER-DTR and as elements of the list that is the value of COMPLEMENT-DTRS. Since these features are referenced by the rule schemata, we replace each goal involving a rule schema by a conjunction of the rule schema and a number of goals that require that each of the daughters must themselves be signs. Instead of the goal $R_i(x)$, we use the conjunction

$$R_i(x) \wedge \text{sign_of_english}(y_1) \wedge \ldots \wedge \text{sign_of_english}(y_n)$$

where the y_i are the feature structures of sort *sign* that appear explicitly in x.

In order to transform this into Horn clauses, we expand the formula to disjunctive normal form, and make a Horn clause of each disjunct, as shown in figure 2.11.

The current version of HPSG [Pollard and Sag, 1994] differs slightly from the one given above: the disjunction of the rule schemata $R_1 \ldots R_q$ is replaced by a new principle, the *ID Principle.* Since the content of the *ID Principle* is exactly the disjunction of the rule schemata $R_1 \ldots R_q$, partial deduction applied to the *ID Principle* provides the step from the specification in (2.5) to the one in figure 2.11.[7]

Our formalisation of HPSG as definite clauses follows the tradition established in work with formalisms such as STUF [Dörre and Seiffert, 1991] and CUF [Dörre and Dorna, 1993].[8]

What remains is a formalisation of the principles. In [Pollard and Sag, 1987] principles are formulated as *conditional* feature structures. We treat these as logical conditionals of the form $A \rightarrow B$, which is equivalent to $\neg A \vee B$. Take for example the head feature principle, given in [Pollard and Sag, 1987] as the conditional feature structure (2.6).

[7] Similarly for the disjunction of lexical entries.

[8] Note that CUF uses a functional notation which represents an n-place relation as an $(n-1)$-place expression. One of the arguments of the relation is designated as the "result". If the element of a set is chosen as the "result," the relation `member` would be defined by the clauses `member(⟨X|L⟩) ⇒ X` and `member(⟨X|L⟩) ⇒ member(L)`. In case of a relation such as `sign`, it is not clear what the "result" should be. An obvious choice is taking the daughter nodes as arguments, and the mother node as "result." Note that recursion is hidden in the arguments in a (simplified) definition such as sign(head_dtr:(HD&sign)&(comp_dtr:(CD&sign)) ⇒ principles(HD,CD).

$$
\begin{array}{l}
\forall x[\text{sign_of_english}(x) \leftarrow \\
\qquad P_1(x) \wedge \ldots \wedge P_{n+m}(x) \wedge \\
\qquad L_1(x)] \\
\qquad \quad \vdots \\
\forall x[\text{sign_of_english}(x) \leftarrow \\
\qquad P_1(x) \wedge \ldots \wedge P_{n+m}(x) \wedge \\
\qquad L_p(x)] \\
\\
\forall x[\text{sign_of_english}(x) \leftarrow \\
\qquad P_1(x) \wedge \ldots \wedge P_{n+m}(x) \wedge \\
\qquad R_1(x) \wedge \\
\qquad \text{sign_of_english}(y_1) \wedge \ldots \wedge \text{sign_of_english}(y_n)] \\
\qquad \quad \vdots \\
\forall x[\text{sign_of_english}(x) \leftarrow \\
\qquad P_1(x) \wedge \ldots \wedge P_{n+m}(x) \wedge \\
\qquad R_p(x) \wedge \\
\qquad \text{sign_of_english}(y_1) \wedge \ldots \wedge \text{sign_of_english}(y_n)]
\end{array}
$$

Figure 2.11: HPSG as definite clauses

$$\textit{phrasal_sign}[\,] \Rightarrow \begin{bmatrix} \text{syn:loc:head: X} \\ \text{dtrs:head_dtr:syn:loc:head: X} \end{bmatrix} \tag{2.6}$$

We transform this to two clauses:

$$\begin{array}{|l|} \hline \text{head_feature_principle}(\neg_{\textit{phrasal_sign}}[\,]). \\ \text{head_feature_principle}\left({}_{\textit{phrasal_sign}}\begin{bmatrix} \text{syn:loc:head: X} \\ \text{dtrs:head_dtr:syn:loc:head: X} \end{bmatrix}\right). \\ \hline \end{array} \tag{2.7}$$

Given that we apply the predicate `head_feature_principle` only to feature structures of sort *sign*, we can replace the negated sort *phrasal_sign* by the sort *lexical_sign*, since it is equivalent to *sign* & ¬ *phrasal_sign*. The resulting formulation of the head feature principle as a definite clause is given in (2.8).

$$\begin{array}{|l|} \hline \text{head_feature_principle}({}_{\textit{lexical_sign}}[\,]). \\ \text{head_feature_principle}\left({}_{\textit{phrasal_sign}}\begin{bmatrix} \text{syn:loc:head: X} \\ \text{dtrs:head_dtr:syn:loc:head: X} \end{bmatrix}\right). \\ \hline \end{array} \tag{2.8}$$

It is this formulation as definite clauses to which partial deduction techniques will be applied below. The purpose of partial deduction is to resolve goals which involve checking of principles, in order to arrive at a rule format in (2.9), where Γ is a sequence of goals that cannot be reduced by partial deduction.

$$\text{sign(M)} \leftarrow \text{sign}(D_1) \wedge \text{sign}(D_2) \wedge \Gamma. \tag{2.9}$$

Our objective is for Γ to turn out to be no more than the `cp/2` relation, in which case we have arrived at the rule format (1.8) proposed by van Noord [van Noord, 1993], repeated below as rule (2.10). This is a desirable effect because efficient processing algorithms are known for rules in this format. As already mentioned

in chapter 1, van Noord restricts grammars "to consist of definite clauses defining only one unary relation" [van Noord, 1993, p. 43] (the relation `sign/1`), and argues that all other relations can be compiled away by partial deduction techniques. In addition to the relation `sign/1`, van Noord introduces an additional relation `cp/2` (construct_phonology), which makes the combination of the phonological values of the daughters in a rule explicit, especially in cases where it is not restricted to concatenation.

$$sign(M) \leftarrow sign(D_1) \wedge sign(D_2) \wedge cp(M, \langle D_1, D_2 \rangle). \tag{2.10}$$

Before reporting on the outcome of partial deduction experiments on two versions of HPSG, we will enumerate the principles of HPSG, and discuss their properties with respect to partial deduction in the following section.

2.3.2 Principles of HPSG

The formulations of the principles given are based on the sort hierarchy and the principles in [Pollard and Sag, 1994].

In the following discussion of principles, we concentrate on headed structures, which have been the focus of attention in the HPSG literature. The case of lexical signs will not be mentioned explicitly for those principles that only apply to phrases and are therefore vacuously true of lexical signs. Those principles that are seen as constraints on lexical entries can be used to perform a consistency check on the lexicon at the step when the lexicon is expanded by means of lexical rules (cf. section 2.5).

For each principle, we repeat its definition in the HPSG book, and a discuss its properties with respect to partial deduction.

When discussing partial deduction of each principle, we will do so by distinguishing different cases for the different subsorts of headed structures (head-complement, head-adjunct, head-marker, and head-filler structures). Since these subsorts are mutually incompatible, they cannot co-occur in one grammar rule in the output of the partial deduction procedure. The effect of partial deduction is to distribute some of the disjunctions found in the principles of HPSG onto different instances of grammar rules, with the exception of infinite disjunctions arising from recursively defined relations, and some other disjunctions which are better processed at runtime in order to avoid an explosion of the rule set. The latter can be handled by treating them as *guarded constraints,* which wait until the input is sufficiently instantiated to allow a deterministic execution of the constraint.

2.3.2.1 Head Feature Principle

> In a headed phrase, the values of SYNSEM|LOCAL|CATEGORY|HEAD and DAUGHTERS|HEAD-DAUGHTER|SYNSEM|LOCAL|CATEGORY|HEAD are token-identical.
> (p. 34)

Properties. The Head Feature Principle is deterministic (for all headed phrases). It can be compiled into a feature term and be unified with all rule schemata of HPSG. There is no need to retain this principle as a goal for processing at runtime.

2.3.2.2 Subcategorisation Principle

> In a headed phrase, the list value of DAUGHTERS|HEAD-DAUGHTER|SYNSEM|LOCAL|CATEGORY|SUBCAT is the concatenation of the list value of SYNSEM|LOCAL|CATEGORY|SUBCAT with the list consisting of the SYNSEM values (in order) of the elements of the list value of DAUGHTERS|COMPLEMENT-DAUGHTERS.
> (p. 34)

Properties. The Subcategorisation Principle is more problematic than the Head Feature Principle. It is applicable to all subsorts of headed phrases, even to those that do not have complement daughters, such as head-adjunct structures, head-marker structures and head-filler structures. This is due to the fact that all headed phrases in HPSG have a COMPLEMENT-DAUGHTERS (COMP-DTRS) list, even if they don't have any complement daughters. In this case, the COMP-DTRS list is empty.

The Subcategorisation Principle makes use of the (recursively defined) concatenation predicate, and implicitly of a predicate that relates a list of signs to a list of the synsem values of these signs. Both of these cannot be reduced by means of partial deduction unless the length of the list is known in advance.

We will look at two approaches for dealing with this problem. In the first approach (cf. section 2.4.1), the appropriateness specification for the sort *constituent-structure* (the value of the feature DAUGHTERS) is changed in such a way that the feature COMP-DTR(S) is only appropriate for head-complement structures, but not for head-adjunct, head-filler and head-marker structures. In addition, the value of the feature COMP-DTR is of sort *sign* instead of *list(sign)*. The result is a binary branching reformulation of HPSG.

In the second approach (cf. section 2.4.2), we retain the appropriateness specification of HPSG, but make use of lexical information about the maximal length of subcat lists in order to allow a terminating partial deduction involving the list-processing predicates that occur in the definition of the Subcategorisation Principle.

2.3.2.3 ID Principle

Every headed phrase must satisfy exactly one of the ID schemata.
(p. 37)

Schema 1 (head-complement structure)

The SYNSEM|LOCAL|CATEGORY|SUBCAT value is ⟨ ⟩, and the DAUGHTERS value is an object of sort *head-comp-struc* whose HEAD-DAUGHTER value is a phrase whose SYNSEM|NONLOCAL|TO-BIND|SLASH value is { }, and whose COMPLEMENT-DAUGHTERS value is a list of length one.
(p. 38)

Schema 2 (head-complement structure)

The SYNSEM|LOCAL|CATEGORY|SUBCAT value is a list of length one, and the DAUGHTERS value is an object of sort *head-comp-struc* whose HEAD-DAUGHTER value is a word.
(p. 39)

Schema 3 (head-complement structure)

The SYNSEM|LOCAL|CATEGORY|SUBCAT value is ⟨ ⟩, and the DAUGHTERS value is an object of sort *head-comp-struc* whose HEAD-DAUGHTER value is a word.
(p. 40)

Schema 4 (head-marker structure)

The DAUGHTERS value is an object of sort *head-marker-struc* whose HEAD-DAUGHTER|SYNSEM|NONLOCAL|TO-BIND|SLASH value is { }, and whose MARKER-DAUGHTER|SYNSEM|LOCAL|CATEGORY|HEAD value is of sort *marker*.
(p. 46)

Schema 5 (head-adjunct structure)

The DAUGHTERS value is an object of sort *head-adjunct-struc* whose HEAD-DAUGHTER|SYNSEM value is token-identical to its ADJUNCT-DAUGHTER|SYNSEM|LOCAL|CATEGORY|HEAD|MOD value and whose HEAD-DAUGHTER|SYNSEM|NONLOCAL|TO-BIND|SLASH value is { }.
(p. 56)

Schema 6 (head-filler structure)

> The DAUGHTERS value is an object of sort *head-filler-struc* whose HEAD-DAUGHTER|SYNSEM|LOCAL|CATEGORY value satisfies the description [HEAD *verb* [VFORM *finite*, SUBCAT ⟨ ⟩]], whose HEAD-DAUGHTER|SYNSEM|NONLOCAL|INHERITED|SLASH value contains an element token-identical to the FILLER-DAUGHTER|SYNSEM|LOCAL value and whose HEAD-DAUGHTER|SYNSEM|NONLOCAL|TO-BIND|SLASH value contains only that element.
> (p. 164)

Properties. The ID Principle brings all the rule schemata of HPSG together in one disjunctive formulation.

In our exercise for compiling out the principles of HPSG into a set of phrase structure rules, the ID schemata 2 and 3 are problematic because they specify a (potentially) infinite number of phrase structure rules due to the fact that there is no upper bound on the length of the COMP-DAUGHTERS list. This problem is shared with the Subcategorisation Principle. Both of the approaches for solving the problems with the Subcat Principle will also reduce the number of phrase structure rules allowed by the schemata to a finite set.

With the first approach (binary branching), there is no longer a potentially infinite set of daughters, but this set can be generated by repeated application of one binary branching phrase structure rule. In section 2.4.1, an alternative formulation of schemata 1, 2 and 3 will be presented.

With the second approach (limiting the length of subcat lists based on lexical information), the length of the COMPLEMENT-DAUGHTERS list will also be finite. Hence, only a finite subset of the potentially infinite rule set allowed by the schemata 2 and 3 needs to be generated.

2.3.2.4 Marking Principle

> In a headed phrase, the MARKING value is token-identical with that of the MARKER-DAUGHTER if any, and with that of the HEAD-DAUGHTER otherwise.
> (p. 45n)

Properties. This principle has two cases: one for head-marker structures, and one for all other headed structures. Since these give rise to different phrase structure rules anyhow, the two cases of the principle can be integrated by means of partial deduction into the appropriate phrase structure rules, and need not be retained as a goal to be called at runtime.

2.3.2.5 Spec Principle

> In a headed phrase whose non-head daughter (either the MARKER-DAUGHTER or COMPLEMENT-DAUGHTERS|FIRST has a SYNSEM|LOCAL|CATEGORY|HEAD value of sort *functional*, the SPEC value of that value must be token-identical with the phrase's DAUGHTERS|HEAD-DAUGHTER|SYNSEM value.
> (p. 51)

Properties. The Spec Principle applies to head-marker structures and head-complement structures, and is true for all other clause types.[9] The Spec Principle has two subclauses for head-complement structures: one for the case where the first complement daughter is substantive (in which case the principle imposes no constraints) , and other other case where it is functional. The same is true for marker structures and the marker daughter. The principle is true for all other kinds of headed structures.

Partial deduction of this principle gives rise to different instances: it requires coindexing of the SPEC value and the head daughter's SYNSEM value for head-marker structures and for head-complement structures which involve a functional category as complement, and is true for all other typed of structures.

2.3.2.6 Semantics Principle

The Semantics Principle is made up of three principles, the Content Principle, the Quantifier Inheritance Principle, and the Scope Principle (pp. 322/323). The principles make use of the notion of *semantic head,* which is defined as follows.

> The semantic head of a headed phrase is
>
> (1) the adjunct daughter in a head-adjunct structure,
>
> (2) the head daughter otherwise.

Properties. This definition has two disjuncts: one for head-adjunct structures, and one for all other structures. This will lead to a duplication of all other clauses which make reference to the definition of *semantic head*. Since our constraint language does not support negation, the two disjuncts are expanded into four: one for

[9] Note that the clauses that define the Spec Principle are themselves acyclic, but can lead to the creation of cyclic structures if a head daughter subcategorises the specifier, i.e., if when the Spec Principle is unified with the Subcat Principle. Such cyclic structures are supported by most current feature logics (feature structures are no longer required to be directed *acyclic* graphs) and do not pose any practical problems because state-of-the-art logic programming or typed feature languages can handle cyclic terms. Prolog appears very old-fashioned in this respect because it can handle cyclic terms at runtime, but cannot print them out.

head-adjunct structures, and three for the other phrase types (head-complement, head-marker and head-filler structures).

Content Principle

> In a headed phrase,
> (Case 1) if the semantic head's CONTENT value is of sort *psoa,* then its NUCLEUS is token-identical to the NUCLEUS of the mother;
> (Case 2) otherwise, the CONTENT of the semantic head is token-identical to the CONTENT of the mother.

Properties. The Content Principle has two clauses, one for content values of sort *psoa,* and one for the other sorts (*nom-obj* and *quant*). In this case, three clauses result. Since each of these makes use of the definition of *semantic head*, partial deduction of this clause, will lead to twelve clauses, and amounts to an expansion to disjunctive normal form.

As an alternative, one could consider using guarded constraints to delay the execution of the content principle until the CONTENT value is instantiated to one of the three subsorts of *content*.

Quantifier Inheritance Principle

> In a headed phrase, the RETRIEVED value is a list whose set of elements forms a subset of the union of the QSTOREs of the daughters, and is non-empty only if the CONTENT of the semantic head is of sort *psoa*; and the QSTORE value is the relative complement of the RETRIEVED value.
> (p. 322/323)

Properties. The Quantifier Inheritance Principle can only be processed efficiently by making use of set constraints. Just having a *subset constraint* instead of enumerating the different subsets at each point prevents the growth of the search space that would be the consequence of a naive implementation of this principle.

In addition, a procedure for collecting the elements of a set (subset of the union of the QSTOREs) is needed, and its execution must be delayed (guarded constraint) to prevent the undesired enumeration of the possible subsets.

Scope Principle

> In a headed phrase whose semantic head is of sort *psoa*, the QUANTS value is the concatenation of the RETRIEVED value with the QUANTS value of the semantic head.

Properties. Like the Content Principle, the Scope Principle distinguishes between phrases whose semantic head is of sort *psoa*, and phrases in which it has another sort (and imposes no constraints at all because there are no quantifiers involved). In order to prevent termination problems with the concatenation procedure, the execution of this principle should be delayed until the RETRIEVED value has become instantiated via the Quantifier Inheritance Principle.

2.3.2.7 Raising Principle

> Let E be a lexical entry whose SUBCAT list L contains an element X not specified as expletive. Then X is lexically assigned no semantic role in the content of E if and only if L also contains a (nonsubject) Y[SUBCAT ⟨X⟩].
> (p. 117)

Properties. Since the Raising Principle "should be interpreted as a constraint on lexical entries," there is no need to make use of it during parsing or generation, but it should be enforced for every lexical entry listed in its base form in the lexicon or generated by the application of lexical rules (cf. section 2.5).

2.3.2.8 Nonlocal Feature Principle

> In a headed phrase, for each nonlocal feature F = SLASH, QUE, or REL, the value of SYNSEM|LOCAL|INHERITED|F is the set difference of the union of the values on all the daughters and the value of SYNSEM|NONLOCAL|TO-BIND|F on the HEAD-DAUGHTER.
> (p. 164)

Properties. The Nonlocal Feature principle makes use of the notion of set difference, which cannot be implemented in a monotonic feature logic. Therefore, we reformulate the principle by making use of the notion of *disjoint union*:

> In a headed phrase, for each nonlocal feature F = SLASH, QUE, or REL, the union of the values on all the daughters is the disjoint union of SYNSEM|LOCAL|INHERITED|F and of SYNSEM|NONLOCAL|TO-BIND|F on the HEAD-DAUGHTER.

This principle can be directly expressed in our constraint language by making use of the set constraints.

2.3.2.9 Incomplete Constituent Constraint

> [INHER|SLASH *empty-set*] $<$
> COMPLEMENT[INHER|SLASH *nonempty-set*]
> (p. 190/193)

Properties. This linear precedence (LP) rule can be expressed with the combination of linear precedence constraints and guarded constraints.

2.3.2.10 Singleton Rel Constraint (parochial)

> For any sign, the SYNSEM|NONLOCAL|INHERITED|REL value is a set of cardinality at most one.
> (p. 211)

Properties. This constraint can be expressed by making use of guarded constraints and the fixed cardinality set constraints. The formalisation needs to make use of a disjunction between the empty set and a set of cardinality one.[10]

2.3.2.11 Relative Uniqueness Principle (parochial)

> For any phrase, a member of the set value of SYNSEM|NONLOCAL|INHERITED|REL may belong to the value of that same path on at most one daughter.
> (p. 212)

Properties. Unless the constraint language allows to quantify over different daughters, this constraint can only be expressed by means of a disjunctive formulation in which a member of the SYNSEM|NONLOCAL|INHERITED|REL is nondeterministically coindexed with the value of the same path on any daughter and declared to be disjoint with the same path on the other daughters.

2.3.2.12 Clausal Rel Prohibition

> For any *synsem* object, if the LOCAL|CATEGORY|HEAD value is *verb* and the LOCAL|CATEGORY|SUBCAT value is $\langle\ \rangle$, then the NONLOCAL|INHERITED|REL value must be $\{\ \}$.
> (p. 220)

[10] Alternatively, it can be expressed with a `forall` constraint with a variable as a value, which fails only if a set has two or more distinct members.

Properties. This constraint applies to all schemata except schema 2 (where the SUBCAT list is not empty). For schemata 1 and 3, the SUBCAT value is always empty, so that the requirement that the REL set be empty can be instantiated on those instances where the head-daughter is verbal. For the verbal instances of the other rule schemata, this principle is best implemented by means of a guarded constraint with a guard on the SUBCAT list being empty, and a consequent on REL being the empty set.

2.3.2.13 Control Theory

> If the SOA-ARG value of a *control-qfpsoa* is token-identical with the CONTENT value of a local object whose CATEGORY|SUBCAT value is a list of length one, then the member of that list is (1) reflexive, and (2) coindexed with the INFLUENCED (respectively, COMMITTOR, EXPERIENCER) value of the *control-qfpsoa* if the latter is of sort *influence* (respectively, COMMITMENT, ORIENTATION).
> (p. 302)

Properties. The above definition is best modelled by a guarded constraint, but expansion to disjunctive normal form would also be a possibility.

2.3.2.14 Quantifier Binding Condition

> Let X be a quantifier, with RESTINDEX|INDEX value Y on the QUANTS list of a *psoa* Z, and P a path in Z whose value is Y. Let the address of X in Z be QUANTS|Q|FIRST. Then P must have a prefix of one of the following three forms:
> (1) QUANTS|Q|FIRST|RESTINDEX|RESTRICTION;
> (2) QUANTS|Q|REST; or
> (3) NUCLEUS.
> (p. 327)

Properties. This principle makes use of a variable Q over paths, and universally quantifies over this variable. This is beyond the expressiveness of our constraint language.

2.3.2.15 Principle of Contextual Consistency

> The CONTEXT|BACKGROUND value of a given phrase is the union of the CONTEXT|BACKGROUND values of the daughters.
> (p. 333)

Properties. This principle can be expressed with the set union constraint of our constraint language.

2.3.2.16 Constituent Order Principle

The current version of HPSG [Pollard and Sag, 1994] does not mention the constituent order principle. This is clearly an omission since the current set of principles does not specify how the PHON value of a phrasal sign is derived from the PHON values of its daughters. In [Pollard and Sag, 1987], there was a specification of the constituent order principle, which made reference to a function *order-constituents*, which was not further specified.

For a language like English, it is possible to get away with a simple version of the Constituent Order Principle, which only makes use of concatenation and can be specialised for each rule schema (subjects, markers, and fillers precede their heads; heads precede non-subject complements; and adjuncts specify their direction of combination lexically).

In this case, the concatenation can be implemented by means of difference lists, and partial deduction of the Constituent Order Principle leaves no goals to be processed at runtime.

For a language such as German, more elaborate operations are needed in the statement of the Constituent Order Principle, such as the domain union mechanism proposed by Reape (cf. section 3.2).

Since these operations are normally recursively defined and non-deterministic, they cannot be reduced at compile time, and partial deduction of a more serious version of the Constituent Order Principle will leave some goals that must be processed at runtime, which are given as the relation `cp/2` in rule (1.8) on page 15.

2.3.2.17 Binding Theory

Although we will not discuss the Binding Theory in detail, we see no principled reason why the HPSG Binding Theory could not be expressed with the constraint language introduced in chapter 1. The notion of *o-command*, which relies on the obliqueness hierarchy, can be modelled with the linear precedence constraints, and equality and inequality constraints are sufficient for expressing the notions *o-bound* and *o-free*.

2.3.3 Conclusion on HPSG principles

A expressive constraint language making use of set constraints, linear precedence constraints, and guarded constraints is necessary and sufficient to express most principles of HPSG. A less expressive constraint language is not enough because HPSG makes heavy use of sets and constraints and operations on them. The use of

guarded constraints is appropriate when the applicability of a principle depends on the instantiation of certain features.[11] There are only very few principles that are not expressible, among them the Coordination Principle, which makes use of a subsumption constraint, and the Quantifier Binding Condition which universally quantifies over a variable over paths.[12]

Partial deduction is applied to HPSG by making use of the definition of a sign as given in figure 2.11, and selectively replacing the calls to the principles by the antecedents of the clauses defining the principles. If the antecedent of a clause is non-empty, it may contain other goals that are replaced by the bodies defining these clauses and so on. However, certain recursively defined clauses are not expanded because their expansion would not terminate. Among these are the call to `append/3` in the Subcat Principle, and the recursive calls to `sign/1` in the definition of a phrasal sign.

The outcome of the partial deduction exercise presented in the following section is thus a (more or less) large number of phrase structure rules, each of which has incorporates the appropriate disjunct of each principle, and has possibly some goals for recursively defined procedures such as `append/3` plus a number of set constraints, linear precedence constraints and guarded constraints derived from the principles.

2.4 Partial Deduction Applied to HPSG

In this section, we apply partial deduction to a declarative, principle-based specification of HPSG in order to derive a rule-based grammar from it.

In implementing the HPSG grammar, we have made use of the HPSG grammar developed for ALE by Gerald Penn (henceforth referred to as the ALE grammar). The sort definitions and appropriateness specification were taken over with a few modifications. The format for grammar rules and principles, however, differs significantly from the ALE grammar. Since ALE can only process rule-based grammars, the ALE grammar takes grammar rules as basic and treats the principles as procedural attachments to the rules (introduced by the `goal>` keyword). In the ALE grammar, all principles are listed separately for each rule. This can cause a loss of efficiency for the following reasons: As discussed in the previous section, most principles of HPSG have a disjunctive formulation, and only one of the disjuncts is applicable to each grammar rule. In the ALE implementation, the disjunctions in the principles lead to choice points which must be eliminated at runtime. In

[11] In a less expressive constraint language such as ALE, sets must be simulated by lists, and non-declarative control (the cut) is needed for the implementation of set operations. In section 2.4.2, we will report on our experiences with applying partial deduction to an HPSG grammar implemented in ALE.

[12] In both of these cases, Pollard and Sag acknowledge that the formalisation of these principles is still problematic; in case of the Quantifier Binding Condition they hint at an alternative analysis on the level of semantic interpretation.

our implementation, we overcome this problem by performing partial deduction of the principles already at compile time, so that any incompatible disjuncts are eliminated and need not be resolved at runtime.

We describe two experiments making use of partial deduction in order to turn a principle-based into a rule-based one. As discussed above, HPSG makes use of rule schemata (schemata 1 and 3) that correspond to an infinite number of phrase structure rules, so that a straightforward application of partial deduction techniques will not terminate. We will compare two approaches towards this problem. The first approach (section 2.4.1) is a reformulation of HPSG theory making use of binary branching structures, which has been proposed in Categorial Grammar analyses, and suggested for HPSG analyses of German. The second approach (section 2.4.2) sticks with the original analysis, but makes use of the fact that the number of daughters in a local tree is bounded by the length of subcat lists of lexical entries. The partial deduction technique is extended by making use of information about the possible classes of lexical entries (and the lengths of their subcat lists) to ensure that the output of the partial deduction step is a finite number of rules.

In both partial deduction experiments, the goals which occur in the definition of a phrasal sign are replaced by the bodies of the clauses which define these goals. In addition, some principles will be specified as guarded constraints.

2.4.1 PD Experiment 1: Binary Branching HPSG

In this partial deduction experiment, we start out from a declarative definition of an HPSG sign that has the following format.

```
sign(X) ←
        lexical_sign(X).
sign(X) ←
        phrasal_sign(X).
```

We pay special attention to the principles which apply to phrasal signs, and only little attention to lexical entries and lexical rules, which are dealt with in section 2.5. A phrasal sign is specified by listing all the principles that it must satisfy as goals (figure 2.12); including the ID Principle, which comprises the rule schemata, and a new goal (`recursion_driver/1`, cf. figure 2.13), which enforces the constraint that all daughters of every phrasal sign must themselves also be signs of HPSG. Normally, this constraint is implicit in the notation of grammar rules, and is enforced by the respective parsing or generation algorithm. Since our goal in chapter 3 is to use a deductive approach and to abstract away from particular parsing or generation algorithms, we need to make this constraint explicit. Our grammar also differs from the ALE grammar by making constituent order explicit through a call to the constituent order principle. In the ALE grammar, the

combination of PHON values is restricted to concatenation, but this is not made explicit because it is implicit in the grammar rule notation.

```
phrasal_sign(X) ←
        id_principle(X) ∧
        head_feature_principle(X) ∧
        subcat_principle(X) ∧
        marking_principle(X) ∧
        recursion_driver(X) ∧
        spec_principle(X) ∧
        nonlocal_feature_principle(X) ∧
        trace_principle(X) ∧
        subject_condition(X) ∧
        weak_coordination_principle(X) ∧
        singleton_rel_constraint(X) ∧
        relative_uniqueness_principle(X) ∧
        clausal_rel_prohibition(X) ∧
        binding_theory(X) ∧
        control_theory(X) ∧
        semantics_principle(X) ∧
        quantifier_binding_condition(X) ∧
        principle_of_contextual_consistency(X) ∧
        constituent_order_principle(X).
```

Figure 2.12: Definition of a phrasal sign of HPSG

2.4.1.1 Modifications to HPSG

The use of a binary branching version of HPSG is not only motivated by the desire to allow an easier compilation into a rule-based grammar, but there is linguistic evidence that binary branching structures are more adequate for the description of many languages.[13]

In order to implement a binary branching version of HPSG, the appropriateness specification for the feature COMP-DTR(S) needs to be changed. Its value need no longer be a list of signs, but just a structure of sort *sign*. Only four principles need modification for a binary branching version of HPSG: the ID Principle, the Subcat Principle, the Spec Principle and the Constituent Order Principle.

[13] We cannot discuss this question in detail here. See for example Uszkoreit's work on complex fronting in German [Uszkoreit, 1987a], various other analysis of German word order [Netter, 1992; Oliva, 1992], and a number of Categorial Grammar analyses.

```
recursion_driver(X) ←
        head_daughter(X,HD) ∧
        other_daughter(X,OD) ∧
        sign(HD) ∧
        sign(OD).

head_daughter(dtrs:head_dtr:HD, HD).
other_daughter(dtrs:comp_dtr:OD, OD).
other_daughter(dtrs:adjunct_dtr:OD, OD).
other_daughter(dtrs:marker_dtr:OD, OD).
other_daughter(dtrs:filler_dtr:OD, OD).
```

Figure 2.13: Definition the recursion driver

ID Principle

The only schemata which are affected are the schemata 1 to 3, which apply to head-complement structures. These three schemata differ only in the number of elements they take from the head daughter's subcat list, and the number of elements on the mother's subcat list. Since linear order is not encoded in the schemata, but in the Constituent Order Principle, there is no difference between schemata 1, 2, and 3 if binary branching structures are assumed, in which there is exactly one complement daughter.[14] The three schemata fall together into one, which does not impose any constraints on the possible head-complement structures.

> **Revised Schema 123**
>
> An object whose DAUGHTERS value is of sort *head-comp-struc*.

With this modification, HPSG has taken the final step away from rule schemata. There is only one schema for each subsort of *headed-structure*, so that the schemata do not generate possible structures, but only constrain the possible instances of the sort to which they apply. It can be argued that schemata 4, 5, and 6 would equally well qualify as principles because they specify universal constraints on *all head-marker structures, head-adjunct-structures,* and *head-filler structures,* respectively. The binary branching version can therefore be regarded as a "HPSG without rule schemata."

[14] The only other difference between schema 1 in comparison to schemata 2 and 3 is that the head daughter is phrasal in the former, and lexical in the latter. Since this is the only difference, the schemata can be replaced by one schema in which a disjunction (*lexical* ∨ *phrasal*) is specified, or equivalently the sort *sign*.

Subcat Principle

The Subcat Principle can be simplified if the value of COMPLEMENT-DAUGHTERS is a sign instead of a list of signs:

> In a head-complement structure, the SYNSEM|LOCAL|CAT|SUBCAT value of the head daughter is the concatenation of the SYNSEM|LOCAL|CAT|SUBCAT value of the mother and the singleton list containing the SYNSEM value of the complement daughter.

Spec Principle

Only one small modification is needed for the Spec Principle: the path COMPLEMENT-DAUGHTERS|FIRST must be replaced by COMPLEMENT-DAUGHTER.

Constituent Order Principle

The COP enforces the order of heads and complements, markers and fillers. The relative order of heads and subjects, heads and non-subject complements, head and markers, and heads and fillers can be fixed by the Constituent Order Principle (for English). For adjuncts, there is no fixed order. Therefore, functors must lexically specify their direction of application, as in Categorial Unification Grammar [Uszkoreit, 1986], and the COP will enforce this lexical specification.

All other principles of HPSG can be taken over unmodified.

2.4.1.2 Outcome of Partial Deduction Experiment 1

In this experiment, we used partial deduction to transform a declaratively specified principle-based HPSG into a rule-based grammar. Several rule-based grammars with a different number of rules were produced as the output. In the most extreme case, when partial deduction is applied at every possible point, the grammar is transformed to its disjunctive normal form with up to a hundred rules, and in the other extreme, where partial deduction is only applied to the ID principle, the resulting grammar has only four highly schematic rules each of which contains a call to the remaining principles as a goal.

In practice, the grammar which is best suited for efficient processing lies between the two extremes of disjunctive normal form and only schematic rules. If the rules are too schematic, there are too many goals that must be executed at runtime, and if the rules are too specialised, a loss of efficiency will result due to the large number of rules that must be matched and due to the fact that rules will have a prefix[15] in common, so that the processing becomes less deterministic for left-to-right processing strategies.

[15] Two rules are said to have a common prefix if their right-hand sides start with the same non-empty sequence of non-terminal symbols. For example the rules [$S \rightarrow V\ NP\ PP$] and [$S \rightarrow V\ NP\ NP$] have the common prefix $V\ NP$.

Partial deduction is a method which allows to produce different kinds of rule-based grammars very easily from a principle-based specification, and to evaluate their efficiency with respect to a given processing model. The following section describes an experiment in which partial deduction has been used to optimise a given grammar for a bottom-up chart parser.

2.4.2 PD Experiment 2: COMP-DTRS as a List-Valued Feature

In the preceding section, we have assumed for ease of exposition and for linguistic reasons, that the value of the feature COMP-DTRS is of sort *sign*, and not, as in the HPSG book, a list of signs. A list as the value of COMP-DTRS makes the partial deduction procedure somewhat more complicated since many of the principles need to include calls to recursively defined auxiliary predicates that handle the COMP-DTRS list. Since the schemata 1 and 3 specify an infinite number of phrase-structure rules, a straightforward application of the partial deduction technique is not possible for this case because its output would be an infinite number of rules.

This experiment was based on the ALE grammar. Although the ALE grammar is a rule-based grammar, it is still interesting to apply partial deduction techniques to it. The reason is that the rules of the grammar are highly schematic (only six rules), and check all principles of the grammar after application of the rule as procedural attachments. The purpose of the experiment is to find out whether partial deduction can bring any additional performance benefits for a grammar that has already been written with processing performance in mind (as evidenced by the use of cuts to make procedures deterministic and by the carefully chosen order of the goals).

The outcome of the partial deduction experiment is a grammar that has more rules than the original one, and calls to principles have been replaced by goals which appear in the definitions of the principles and cannot be expanded because they would have an infinite number of solutions unless they are instantiated by the daughters of the rule, or because they make use of extra-logical control constructs like the cut that would not give the correct results when expanded by the partial deduction process.[16] While some of these cuts serve as conditional (*if-then-else*) operators, and can be replaced by a larger number of clauses (which explicitly enumerate the alternatives in the *else*-part, other cuts are more problematic because they serve to instantiate uninstantiated variables to a default value (for example uninstantiated set values are instantiated to the empty set). In figure 2.14, we summarise the status of the different principles of the ALE grammar and indicate for the declaratively specified principles whether they are deterministic, and if not how many different cases they have.

[16] The cut is the only non-declarative control construct provided in ALE 2.0. It has been implemented in ALE because it was believed to be necessary for the efficiency of syntactic processing.

A large number of procedures using the cut have to do with the handling of sets, which are implemented in ALE by using list-like structures (with the features ELEMENT and ELEMENTS). The situation can be greatly improved if these procedures are replaced by set constraints (cf. section 1.1.1.4).

Principle	declarative	deterministic
Head Feature Principle	yes	yes
Inv-minus Principle	no	—
Subcat Principle	yes	no (∞)
Marking Principle	yes	yes
Spec Principle	yes	no (2)
Semantics Principle	no	—
Universal Trace Principle	no	—
Parochial Trace Principle	no	—
Subject Condition	no	—
Nonlocal Feature Principle	no	—
Single Rel Constraint	yes	no (2)
Clausal Rel Prohibition	yes	no (7)
Relative Uniqueness Principle	no	—
Conx Consistency Principle	no	—
Deictic-cindices Principle	yes	yes

Figure 2.14: Use of non-declarative control in the ALE grammar

The input grammar (the original ALE grammar) consists of six rule schemata. Attached to each schema are up to 15 goals (principles). Each principle can be defined by several clauses which can make calls to other goals. The input grammar has nine rules, and calls to 85 principles (9.4 per rule).

Several grammars were produced as output of the partial deduction process. They differ in several respects. In one case, only the deterministic principles have been expanded by partial deduction, in another case only the principles which have finite expansions.

In a further experiment, we examined the grammar and lexicon to determine that the maximal length of a subcat list and hence the maximum number of complement daughters in a rule is three. This information has been made use of to ensure that the subcat principle and the predicates which treat a sequence of categories have only a finite number of expansions. The resulting output grammars have between 9 and 69 rules.

One clause of the input grammar (schema 3) for this experiment is given in figure 2.15. It is Schema 3 of HPSG. Schema 3 is one of the principles which are problematic because they specify an infinite number of phrase structure rules. In order to overcome this problem, the first goal of the clause has been added as additional knowledge about the grammar (the maximal number of daughters) in

order to ensure termination of the partial deduction procedure. Partial deduction has been applied to the goals `subcat_principle/3, synsems_to_signs/2,` and `signs/1,` but not `sign/1` which is reserved for processing at runtime.[17]

The somewhat non-standard, but logically correct specification of the relation `append/3` was chosen to give a unique result in cases where the second argument is the empty list, which will arise for the rule schemata 4, 5, and 6. In this case the result is `append(X,⟨ ⟩,X)` instead of the infinite disjunction which would result from the usual textbook definition and cause termination problems: `append(⟨ ⟩,⟨ ⟩,⟨ ⟩); append(⟨A⟩,⟨ ⟩,⟨A⟩); append(⟨A,B⟩,⟨ ⟩,⟨A,B⟩); append(⟨A,B,C⟩,⟨ ⟩,⟨A,B,C⟩); ...`

```
sign(Mother & subcat:(Subcat&⟨ ⟩) ) ←
        (Comp_Dtr_Synsems = ⟨_⟩ or ⟨_,_⟩ or ⟨_,_,_⟩) ∧
        sign(Head_Daughter & subcat: HD_Subcat) ∧
        synsems_to_signs(Comp_Dtr_Synsems,Comp_Dtrs) ∧
        signs(Comp_Dtrs) ∧
        subcat_principle(Subcat,Comp_Dtrs,HD_Subcat) ∧
        ... other principles

subcat_principle(Subcat,Comp_Dtrs,HD_Subcat) ←
        append(Subcat,Comp_Dtr_Synsems,HD_Subcat).

append(⟨H|T1⟩,⟨H2|T2⟩,⟨H|T⟩) ←
        append(T1,⟨H2|T2⟩,T).
append(⟨ ⟩,⟨H|T⟩,⟨H|T⟩).
append(X,⟨ ⟩,X).

signs(⟨ ⟩).
signs(⟨Dtr|Dtrs⟩) ←
        sign(Dtr) ∧
        signs(Dtrs).
```

Figure 2.15: Schema 3 as input to partial deduction

In figure 2.16, the output of partial deduction applied to schema 3 is shown. The outcome are three clauses, which can be chosen deterministically if the length of the subcat list of the head daughter is known, which is generally the case (with

[17]The original grammar had already performed partial deduction by hand, and instantiated the effect of the subcat principle on the rule schemata 2 and 3, so that the call to the subcat principle was redundant. For partial deduction of the subcat principle to make any sense at all, these instantiations were removed from the rules.

the exception of cases of argument inheritance).

```
rule(Mother & subcat:⟨ ⟩) ←
        sign(Head_Daughter & subcat: ⟨HD_Subcat⟩) ∧
        sign(synsem:HD_Subcat) ∧
        ... other principles

rule(Mother & subcat:⟨ ⟩) ←
        sign(Head_Daughter & subcat: ⟨Subj,Obj⟩) ∧
        sign(synsem:Subj) ∧
        sign(synsem:Obj) ∧
        ... other principles

rule(Mother & subcat:⟨ ⟩) ←
        sign(Head_Daughter & subcat: ⟨Subj,Obj,Obj2⟩) ∧
        sign(synsem:Subj) ∧
        sign(synsem:Obj) ∧
        sign(synsem:Obj2) ∧
        ... other principles
```

Figure 2.16: Schema 3 as output of partial deduction

The resulting output grammar has 13 rules, and 108 calls to principles (8.3 per rule).

2.4.2.1 Performance of the Compiled Grammar

In order to compare the performance of the grammar with and without the application of partial deduction, both grammars were processed by a bottom-up chart parser similar to the one used in ALE. The runtimes obtained with the original ALE grammar and with the grammar after partial deduction are summarised in appendix B.2. The grammar was tested with twelve sentences which varied in length between two and 21 words. The runtimes with the grammar to which partial deduction has been applied were between 23.8 and 67.5 percent of the runtimes for the grammar without partial deduction. On average, the runtime with partial deduction was less than half (46.4 percent) of the runtime without partial deduction,, which included an explicit call to the subcat principle.[18]

Further experiments to reduce the runtime by performing partial deduction on other principles as well did not bring any real performance benefits. This is due to a number of reasons. First of all, the ALE grammar did not allow extensive

[18] This partial deduction step was already encoded redundantly in the ALE grammar.

application of partial deduction due to its heavy use of non-declarative programming. Secondly, the goals attached to each clause in the ALE grammar are already optimised for processing since they are ordered in such a way that they can almost always be executed deterministically. If there is no non-determinism, then partial deduction can bring little performance benefit because Prolog is very good at dealing with deterministic procedures efficiently.

In some cases, the larger number of rules produced by the partial deduction process even led to a decrease in performance, which is due to the fact that too large a number of rules leads to increased non-determinism. This is the case when the rules expand out disjunctive possibilities among which a choice can only be made after a number of constituents or goals of the rule have already been processed. This is to be expected since an excessive application of partial deduction expands the grammar to its disjunctive normal form. In order for partial deduction to improve performance, it must be applied selectively and judiciously. Partial deduction yields better results when applied to real declaratively specified principle-based grammars.

A grammar making full use of the extended constraint language with set constraints, guarded constraints and linear precedence constraints, which implements Reape's HPSG analysis of German word order ([Reape, 1994], cf. section 3.2.3) has been implemented by making use of partial deduction.[19] In this grammar, all the principles were declaratively specified (as templates, which have expressive power equivalent to definite clauses without recursive definitions). Partial deduction was performed by template expansion. Due to the use of the extended constraint language, the number of rules in the output could be kept small (only two rules) because the non-determinism was not expanded out, but was encoded in constraints which wait until they are sufficiently instantiated for their execution. The output of the partial deduction in this case was a small number of rules, to which no goals are attached, but a quite large number of constraints which could not be solved at compile time. This grammar was used with a head corner parser and a semantic-head driven generator.

2.5 Lexical Rule Expansion as Partial Deduction

HPSG makes use of a mechanism of lexical rules to permit a compact and principle-based representation of the lexicon. Only the base forms of the lexical entries must be listed, and all other forms are derived by means of lexical rules. For applications in which the lexicon is small enough that the use of a full-form lexicon is feasible, it is desirable to expand the lexicon already at compile time to avoid computation of lexical rules at runtime. The expansion of lexical rules to derive a full-form lexicon can be realised as a partial deduction operation.

[19] This work was done by Wojciech Skut in the project LRE-61-061 "Reusability of Grammatical Resources".

```
lexical_sign(FD & phon:Word) ←
        lex(Word0,Fd0) ∧
        lex_rule_expansion(Word0,Fd0,Word,Fd).

lex_rule_expansion(Word,Fd,Word,Fd).

lex_rule_expansion(WordIn,FdIn,WordOut,FdOut) ←
        lexical_rule(WordIn,FdIn,Word1,Fd1) ∧
        lex_rule_expansion(Word1,Fd1,WordOut,FdOut).
```

Figure 2.17: Lexical rule expansion

We assume that the lexicon is represented as pairs of a word form and a feature description. Further, we assume that lexical rules are implemented as a predicate that relates a word form w_1 with the corresponding feature description f_1 to a word form w_2 with feature description f_2. Lexical rules can apply to the output of other lexical rules, but no infinite chains of lexical rule application are possible. The last assumption is important because it ensures the termination of the partial deduction procedure.[20] The definition of the HPSG lexicon is given in figure 2.17.

Partial deduction is used to expand the goals of the predicates `lex/2`, `lexical_rule/4` and `lex_rule_expansion/4`. This procedure has been successfully used to expand the lexicon and the lexical rules for the ALE *grammar*, which contains lexical rules for 3rd person singular present tense finite verb forms, other finite verb forms, passive formation, *it*-extraposition, and subject extraction.

2.6 Conclusion

We have successfully shown the application of partial deduction for turning a principle-based grammar into a rule-based grammar. Since it is in general not possible to expand recursively defined goals and still ensure termination of the partial deduction procedure, the resulting grammar rules will be associated with some goals.

The partial deduction methodology can be applied to various degrees. If it is applied to the fullest, the resulting set of grammar rules is the disjunctive normal form of the grammar, and if it is applied only sparingly, the resulting set of grammar rules will be highly underspecified and contain a large number of goals which must be processed at runtime. Both extremes are undesirable because

[20] In general, this property cannot be guaranteed, as Carpenter's complexity analysis of the lexical rule system of HPSG shows [Carpenter, 1991].

they lead to inefficiency in processing. In case of a large rule set, the inefficiency arises due to the number of rules that must be matched at various processing steps and due to the fact that rules share a prefix of their right-hand sides. In the case of schematic rules, the inefficiency arises from the fact that rules often can be applied, and are then ruled out after processing of the principles attached as goals.

The optimal set of rules may differ for different parsing and generation algorithms. Partial deduction provides a convenient method for generating different rule sets through a selective application of partial deduction. These different rule sets are then evaluated with respect to their efficiency. An advantage of this methodology is that the grammar writer can concentrate on a declarative statement of the principles of the grammar, and a grammar for processing can be generated from the declarative grammatical specification by means of a judicious application of partial deduction.[21]

In our experience partial deduction turned out to be a very useful tool to control the expansion of a principle-based grammar to a more or less rule-based one. The decision whether and how far to expand one principle could be done with a simple control annotation in the grammar and did not require any rewriting (cf. section 5.3.5). Generating a new instance of the grammar was just a matter of a few minutes, and permitted the experimentation with and performance evaluation of a rather large number of grammar instances in a relatively short time (two afternoons).

Partial deduction can be applied for other tasks as well, for example for the lexicalisation of a grammar by instantiating the head daughters of rule schemata with lexical entries. A technique similar to partial deduction has been employed in order to compile HPSG grammars into lexicalised TAG grammars [Kasper *et al.*, 1995]. TAG is a good target for such a compilation because efficient parsing and generation algorithms are known for TAG, and because TAG offers the possibility to combine non-contiguous strings with the adjunction operation.

In the case where the output of the partial deduction still contains goals to be called, the question arises what is the best ordering of these goals. This can be a problem if goals coming from different principles need to be interleaved, as in the case, where the SUBCAT list of the head daughter (a list of SYNSEM values) must be transformed into a list of signs, which are then to be parsed or generated. In our experiment with the ALE grammar (cf. section 2.4.2), the problem did not arise because the goals had already been ordered.

However, instead of the goal ordering by hand, it would be possible to make use of methods to determine the ordering of goals at compile time [Strzalkowski, 1991] at dynamically at runtime as in the CUF system [Dörre and Dorna, 1993].

[21] Of course, the grammar writer has to worry about efficiency through the choice of appropriate data structures, avoidance of spurious ambiguities, etc. The partial deduction methodology only frees the grammar writer from having to worry about the optimal number of rules.

However, a strategy which delays goals until some of their variables are sufficiently instantiated to guarantee deterministic execution (guarded constraints) is computationally simpler than attempts at goal reordering, and is well supported by modern constraint logic programming languages. An example of this CLP approach is the extended constraint language used in this thesis, which has been used to implement a grammar whose rules were associated only with (a large number of) constraints, and only goals for `sign/1` and `cp/2`, after partial deduction was applied to it.

Our experiment with the ALE grammar has shown that it is possible to apply partial deduction to different kinds of grammatical formalisms. The ALE grammar consists of grammar rules with procedural attachments. In order to apply partial deduction to it, we defined a translation procedure from the ALE rule format to our definite clause language, and applied partial deduction to the definite clause specification. The output of the partial deduction was translated back to the ALE rule format. Such a kind of translation is possible for other kinds of formalisms as well, and allows the use of partial deduction to optimise a grammar, in combination with specialised parsing or generation algorithms which require a particular rule format.

A technique closely related to Partial Deduction is Explanation-Based Learning, which has been employed in NLP to pack sequences of frequently occurring deduction steps into a single step [Samuelsson, 1994a; Neumann, 1994a]. The practical difference between partial deduction as used in this work and EBL is that partial deduction is applied to programs to produce new programs as output without any regard to the actually occurring input to these programs. EBL, on the other hand, only remembers deduction steps that have actually been performed in response to a query. The relationship between partial deduction and EBL is discussed in [van Harmelen and Bundy, 1988].

However, EBL does not guarantee completeness because only (generalisations of) those structures that have actually occurred in the training set are present in the compiled grammar that is the output of the EBL procedure. If a structure has not occurred in the input, it will not occur in the output. This does not have a very bad effect on the performance on the algorithm since all frequently occurring structures will be covered if the training set is large enough and carefully chosen. A fallback to the grammar that was the input to EBL is not possible since the increase in performance rests on the fact that the search is reduced by only making use of the output structures.

EBL and partial deduction have different applications: EBL is an algorithm which operates at the level of performance whereas partial deduction operates at the level of competence. It is of course possible to use EBL to further speed up the processing (of frequently occurring subparts) of the rule-based grammars that are the output of the partial deduction algorithm described in this chapter.

Further performance benefits may be gained by applying techniques from LR

parsing in order to transform the rule-based grammar to shift and reduce tables by merging rules which have a prefix of the list of daughters in common. The applicability of these techniques for unification-based grammars has been demonstrated by Samuelsson for parsing [Samuelsson, 1994a] and for generation [Samuelsson, 1995b; Samuelsson, 1995a].

Chapter 3

Bottom-Up Earley Deduction

Given the close relationship between chart parsing and Earley deduction, it is natural to assume that much of the useful work that was done in the field of chart parsing can be generalised to the Earley deduction framework. In this chapter, one such generalisation will be presented: bottom-up Earley deduction as a generalisation of bottom-up chart parsing algorithms that have been successfully applied in numerous NLP systems based on unification and constraint-based grammars.

The Earley deduction algorithm and its extensions that we presented in chapter 1 operate top-down (backward chaining), like Earley's algorithm. The interest has naturally focussed on top-down methods because they are at least to a certain degree goal-directed.

We find bottom-up methods advantageous for the following reasons:

Incrementality: Portions of an input string can be analysed as soon as they are perceived (or parts of the output can be generated as soon as the what-to-say component has decided to verbalise them), even for grammars where one cannot assume that the left-corner has been predicted before it is scanned.

Data-Driven Processing: Top-down algorithms are not well suited for processing grammatical theories like Categorial Grammar or HPSG that would only allow very general predictions because they make use of general schemata instead of construction-specific rules. For these grammars data-driven bottom-up processing is more appropriate. The same is true for large-coverage rule-based grammars which lead to the creation of very many predictions.

Subsumption Checking: Since the bottom-up algorithm does not have a predic-

tion step, there is no need for the costly operation of subsumption checking.[1]

Search Strategy: In the case where lexical entries have been associated with preference information, this information can be exploited to guide the heuristic search.

Bottom-up processing has the disadvantage that it allows a lot of correct, but irrelevant deduction steps. We want to remedy this situation by making a strict selection of items that are initially added to the chart for a bottom-up proof, and by associating each item with one or several indices. Each index represents some aspect of the information of the item, for example relating to its PHON value or its semantic content.

3.1 The Algorithm

Earley deduction [Pereira and Warren, 1983] is a general proof procedure for definite clause programs. The bottom-up variant we present here differs from the top-down variant primarily in the choice of inference rules.

The instantiation (prediction) rule of top-down Earley deduction is not needed in bottom-up Earley deduction, because there is no prediction. There is only one inference rule, namely the completion rule (3.1).[2] In rule (3.1), X, G and G' are atoms,[3] Ω is a (possibly empty) sequence of atoms, and σ is the merged constraint (most general unifier) of G and G'. The leftmost atom in the body of a non-unit clause is always the selected goal.

$$\frac{\begin{array}{c} X \leftarrow G \wedge \Omega \\ G' \leftarrow \end{array}}{\sigma(X \leftarrow \Omega)} \tag{3.1}$$

In principle, this rule can be applied to any pair of unit clauses and non-unit clauses of the program to derive any consequences of the program. In order to reduce this search space and achieve a more goal-directed behaviour, the rule is not applied to any pair of clauses, but clauses are only selected if they can contribute to a proof of the goal. The set of selected clauses is called the *chart*.[4] The selection of clauses is guided by a scanning step (section 3.1.1) and by indexing of clauses (section 3.1.2).

---

[1] Depending on the grammar, subsumption checking may still be needed to filter out spurious ambiguities.

[2] This rule is called *reduction* in [Pereira and Warren, 1983], and is also referred to as the *fundamental rule* in the literature on chart parsing.

[3] We use the term atom here as it is used in the logic programming literature to mean a relation symbol and its arguments — not to be confused with an atomic value in a Prolog term or feature term: a term which has no arguments or features.

[4] The chart differs from the *state* of [Pereira and Warren, 1983] in that clauses in the chart are indexed (cf. section 3.1.2).

3.1.1 Lookup (Scanning)

The purpose of the scanning step, which corresponds to lexical lookup in chart parsers, is to look up base cases of recursive definitions to serve as a starting point for bottom-up processing. The scanning step selects clauses that can appear as leaves in the proof tree for a given goal G.

Consider the simple definition of an HPSG in figure 3.1, with the recursive definition of the predicate `sign/1`.

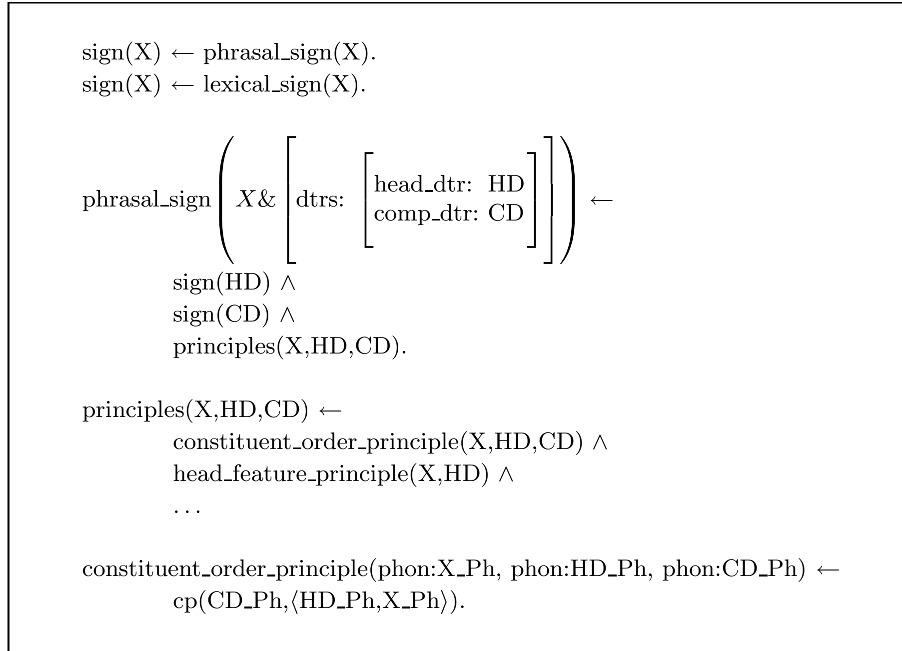

Figure 3.1: Simple definition of HPSG

The predicate `sign/1` is defined recursively, and the base case is the predicate `lexical_sign/1`. But, clearly it is not restrictive enough to find only the predicate name of the base case for a given goal. The base cases must also be instantiated in order to find those that are useful for proving a given goal. In the case of parsing, the lookup of base cases (lexical items) will depend on the words that are present in the input string. This is implied by the first goal of the predicate `principles/3`, the `constituent_order_principle`, which determines how the PHON value of a constituent is constructed from the PHON values of its daughters. In general, we assume that the constituent order principle makes use of a linear and non-erasing operation for combining strings.[5] If this is the case, then all the words contained

[5]There is an obvious connection to the Linear Context-Free Rewriting Systems (LCFRS)

in the PHON value of the goal can have their lexical items selected as unit clauses to start bottom-up processing. In addition, all empty elements specified in the grammar must be added as unit clauses as well.

For generation, an analogous condition on logical forms has been proposed by Shieber [Shieber, 1988b] as the "semantic monotonicity condition," which requires that the logical form of every base case must subsume some portion of the goal's logical form (cf. section 3.3).

Base case lookup must be defined specifically for different grammatical theories and directions of processing by the predicate `lookup/2`, whose first argument is the goal and whose second argument is the selected base case. The clause given in figure 3.2 defines the `lookup` relation for parsing with HPSG. This clause does lexical lookup by finding every word in the input string, looking up its lexical entry, and constructing a lexical sign for this word that can serve as the basis for bottom-up processing.

% lookup(+Goal,-BaseCase)

$$\text{lookup}\left(\text{sign(phon:PhonList)}, \text{lexical_sign}\left(\begin{bmatrix}\text{phon:} & \langle \text{Word} \rangle \\ \text{synsem:} & \text{Synsem}\end{bmatrix}\right)\right) \leftarrow$$

member(Word,PhonList) ∧
lexicon(Word,Synsem).

Figure 3.2: Lookup relation for HPSG parsing

Note that the base case clauses can become further instantiated in this step. If concatenation (of difference lists) is used as the operation on strings, then each base case clause can be instantiated with the string that follows it. This avoids combination of items that are not adjacent in the input string.

$$\text{lookup}\left(\text{sign(phon:PhonList)}, \text{lexical_sign}\left(\begin{bmatrix}\text{phon:} & \langle \text{Word} \mid \text{A} \rangle\text{-A} \\ \text{synsem:} & \text{Synsem}\end{bmatrix}\right)\right) \leftarrow$$

append(_,⟨ Word | A ⟩,PhonList) ∧
lexicon(Word,Synsem).

Figure 3.3: Lookup relation for HPSG parsing

In bottom-up Earley deduction, the first step towards proving a goal is perform

[Vijay-Shanker *et al.*, 1987; Weir, 1988].

lookup for the goal, and to add all the resulting (unit) clauses to the chart. Also, all non-unit clauses of the program, which can appear as internal nodes in the proof tree of the goal, are added to the chart.[6]

The scanning step achieves a certain degree of goal-directedness for bottom-up algorithms because only those clauses which can appear as leaves in the proof tree of the goal are added to the chart.

3.1.2 Indexing

An item in normal context-free chart parsing can be regarded as a pair ⟨R,S⟩ consisting of a (dotted) rule R and the substring S that the item covers (a pair of starting and ending position). The fundamental rule of chart parsing makes use of these string positions to ensure that only adjacent substrings are combined and that the result is the concatenation of the substrings.

In grammar formalisms like DCG or HPSG, the complex nonterminals have an argument or a feature (PHON) that represents the covered substring explicitly. The combination of the substrings is explicit in the rules of the grammar. As a consequence, Earley deduction does not need to make use of string positions for its clauses, as Pereira and Warren [Pereira and Warren, 1983] point out.

For example, to describe the NP *a program that halts* in the sentence *Every professor writes a program that halts sometimes*, a normal chart parser would have to represent this as a pair of the category `np(sg)` and the string position ⟨3,7⟩. In Earley deduction based on Definite Clause Grammars, an explicit representation of the string positions is not needed because the covered string is explicit in the representation of the clause and with the use of difference lists its distance from the end of the string is also uniquely identified:

np(sg,⟨writes,a,program,that,halts,sometimes⟩-⟨sometimes⟩)

Moreover, the use of string positions known from chart parsing is too inflexible because it allows only concatenation of adjacent contiguous substrings. In linguistic theory, the interest has shifted from phrase structure rules that combine adjacent and contiguous constituents to principle-based approaches to grammar that state general well-formedness conditions instead of describing particular constructions (e.g. HPSG), and make use of operations on strings that go beyond concatenation (cf. section 3.2).

The string positions known from chart parsing are also inadequate for generation, as pointed out by Shieber [Shieber, 1988b] in whose generator all items go from position 0 to 0 so that any item can be combined with any item (cf. section 3.3).

[6]Some more restrictive criteria for adding non-unit clauses to the chart would be preferable in order to restrict the search space.

However, the string positions are useful as an indexing of the items so that it can be easily detected whether their combination can contribute to a proof of the goal. This is especially important for a bottom-up algorithm which is not goal-directed like top-down processing. Without indexing, there are too many combinations of items which are useless for a proof of the goal, in fact there may be infinitely many items so that termination problems can arise.

For example, in a grammar formalism that uses an order-monotonic operation such as sequence union for the combination of strings, a combination of items would be useless which results in a sign in which the words are not in the same order as in the input string [van Noord, 1993].

We generalise the indexing scheme from chart parsing in order to allow different operations for the combination of strings. Indexing improves efficiency by detecting combinations that would fail anyway and by avoiding combinations of items that are useless for a proof of the goal.

We define an item as a pair of a clause Cl and an index Idx, written as ⟨Cl,Idx⟩.

Below, we give some examples of possible indexing schemes. Other indexing schemes can be defined if they are needed to optimise the search for a particular grammatical theory.

1. **Non-reuse of Items:** This is useful for generation where no part of the goal's logical form should be verbalised more than once in a derivation, or for LCFRS, where no word of the input string can be used twice in a proof. The operation for combining strings must be non-erasing, so that the words in the string of a mother node are always a superset of words in the strings of the daughter nodes (cf. section 3.2).

2. **Discontinuous constituency:** This indexing scheme is useful for grammars which make use of order-monotonic string operations more powerful than concatenation, such as head wrapping or sequence union.

3. **Non-directional adjacent combination:** This indexing scheme is used if only adjacent constituents can be combined, but the order of combination is not prescribed (e.g. non-directional basic categorial grammars) or in which the order is lexically specified by a functor (e.g., Categorial Unification Grammars [Uszkoreit, 1986; Calder *et al.*, 1988]).

4. **Directional adjacent combination:** This is used for grammars with a "context-free backbone." This is exactly the indexing by the starting and ending position known from chart parsers.

5. **Free combination:** Allows an item to be used several times in a proof, for example for the non-unit clauses of the program, which would be represented as items of the form $\langle X \leftarrow G_1 \wedge \ldots \wedge G_n, \text{free}\rangle$.

Figure 3.4 summarises the properties of these five combination schemes. *Index 1 (I1)* is the index associated with the non-unit clause, *Index 2 (I2)* is associated with the unit clause, and $I1 \star I2$ is the result of combining the indices.

In case 2 ("discontinuous constituency"), the indices X and Y consist of a set of string positions, and the operation $\odot$ is the union of these string positions, provided that no two string positions from X and Y do overlap.

	Index 1 $I1$	*Index 2* $I2$	*Result* $I1 \star I2$	*Note*
1. Non-Reuse	X	Y	$X \cup Y$	$X \cap Y = \emptyset$
2. Discontinuous	X	Y	$X \odot Y$	$X \odot Y = X \cup Y$ *if* $\forall x \in X \forall y \in Y$ $[\text{end}(x) \leq \text{start}(y) \vee$ $\text{end}(y) \leq \text{start}(x)]$
3. Non-directional	⟨X,Y⟩ ⟨Y,Z⟩	⟨Y,Z⟩ ⟨X,Y⟩	⟨X,Z⟩ ⟨X,Z⟩	concrete syntax for ⟨A,B⟩: $A + B$
4. Traditional	⟨X,Y⟩	⟨Y,Z⟩	⟨X,Z⟩	concrete syntax for ⟨A,B⟩: $A - B$
5. Free	X 'free'	'free' X	X X	

Figure 3.4: Overview of indexing schemes for bottom-up Earley deduction

In rule (3.2), the completion rule is augmented to handle indices. $X \star Y$ denotes the combination of the indices X and Y.

$$\frac{\begin{array}{c}\langle X \leftarrow G \wedge \Omega, I1 \rangle \\ \langle G' \leftarrow, I2 \rangle\end{array}}{\langle \sigma(X \leftarrow \Omega), I1 \star I2 \rangle} \tag{3.2}$$

With the use of indices, the `lookup` relation becomes a relation between goals and *items*. The specification of the `lookup` relation in figure 3.5 provides indexing according to string positions as in a chart parser (usable for combination schemes 2, 3, and 4).[7] The algorithm for best-first bottom-up Earley deduction is given in figure 3.6.

Of course, there can be more than one index for each item. For example, the uniform tabular algorithm UTA uses a double indexing (according to string position and to semantic content) in order to be able to use the same items for both parsing and generation [Neumann, 1994b].

In section 3.2, we shall apply these indexing schemes to grammars with discontinuous constituency.

[7] The auxiliary predicate `nth_member/4` is a version of the Prolog predicate `member/2` that returns the position of an element in the list (as start and end positions).

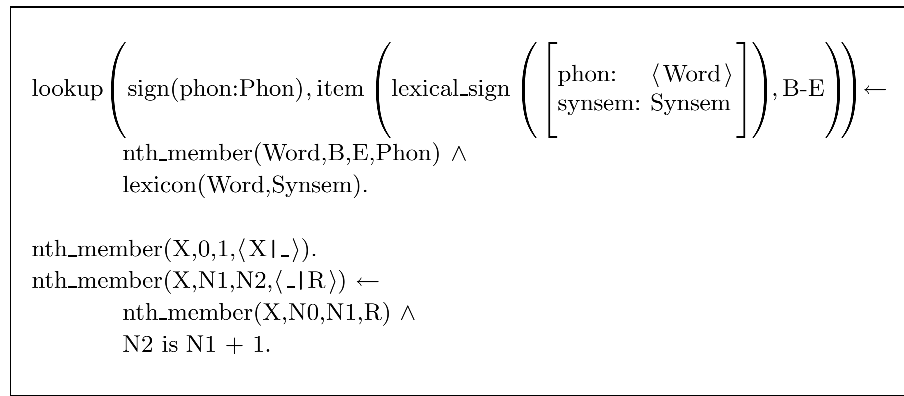

Figure 3.5: Lookup as a relation between query and items

3.1.3 Best-First Search

For practical NL applications, it is desirable to have a best-first search strategy, which follows the most promising paths in the search space first, and finds preferred solutions before the less preferred ones. There are often situations where the criteria to guide the search are available only for the base cases, for example

- weighted word hypotheses from a speech recogniser,
- readings for ambiguous words with probabilities, possibly assigned by a stochastic tagger (cf. [Brew, 1993]),
- hypotheses for correction of string errors which should be delayed [Erbach, 1993a].

In section 4.1, we associate goals and clauses with preference values that are intended to model the degree of confidence that a particular solution is the 'correct' one. Unit clauses are associated with a numerical preference value, and non-unit clauses with a formula that determines how its preference value is computed from the preference values of the goals in the body of the clause. Preference values can (but need not) be interpreted as probabilities.

The preference values are the basis for giving priorities to items. For unit clauses, the priority is identified with the preference value. For non-unit clauses, where the preference formula may contain uninstantiated variables, the priority is estimated by substituting a value for the free variables (cf. section 4.1).[8]

[8]There are also other methods for assigning priorities to items, for example on the basis of surface properties like the length of the covered string, but they have turned out not to be of great use in practice.

procedure prove($Goal$):
– initialize-agenda($Goal$)
– consume-agenda
– for any item $\langle G, I \rangle$
  – return mgu($Goal, G$) as solution if it exists

procedure initialize-agenda($Goal$):
– for every unit clause UC in lookup($Goal, UC$)
  – create the index I for UC
  – add item $\langle UC, I \rangle$ to agenda
– for every non-unit program clause $H \leftarrow Body$
  – add item $\langle H \leftarrow Body$,free$\rangle$ to agenda

procedure add item I to agenda
– compute the priority of I
– agenda := agenda $\cup \{I\}$

procedure consume-agenda
– while agenda is not empty
  – remove item I with highest priority from agenda
  – add item I to chart

procedure add item $\langle C, I1 \rangle$ to chart
– chart := chart $\cup \{\langle C, I1 \rangle\}$
– if C is a unit clause
  – for all items $\langle H \leftarrow G \wedge \Omega, I2 \rangle$
    – if $I = I2 \star I1$ exists
      and $\sigma = \text{mgu}(C, G)$ exists
      then add item $\langle \sigma(H \leftarrow \Omega), I \rangle$ to agenda
– if $C = H \leftarrow G \wedge \Omega$ is a non-unit clause
  – for all items $\langle G' \leftarrow,\ I2 \rangle$
    – if $I = I1 \star I2$ exists
      and $\sigma = \text{mgu}(G, G')$ exists
      then add item $\langle \sigma(H \leftarrow \Omega), I \rangle$ to agenda

Figure 3.6: Algorithm for bottom-up best-first Earley deduction

The implementation of best-first search does not combine new items with the chart immediately, but makes use of an agenda [Kay, 1980], on which new items are stored in order of descending priority.

3.1.4 Goal Types

In constraint-based grammars there are some predicates that are not adequately dealt with by bottom-up Earley deduction, for example the Head Feature Principle and the Subcategorisation Principle of HPSG. As we have argued in chapter 2, the Head Feature Principle just unifies two variables, so that it can be executed at compile time and need not be called as a goal at runtime. The Subcategorisation Principle involves an operation on lists (`append/3` or `insert/3` in different formalisations) that does not need bottom-up processing, but can better be evaluated by top-down resolution if its arguments are sufficiently instantiated. Creating and managing items for these proofs is too much of a computational overhead, and, moreover, a proof may not terminate in the bottom-up case because infinitely many consequences may be derived from the base case of a recursively defined relation.

In order to deal with such goals, we combine Earley Deduction with other proof procedures in the implementation (cf. section 5.3). Dörre [Dörre, 1993] proposes a system with two goal types, namely *trigger goals*, which lead to the creation of items and other goals which don't. Although our system has more goal types, we just use a similar binary distinction for the purposes of this chapter. We divide the goal types into *waiting goals* and *provable goals* (cf. figure 5.9 on page 175). Provable goals can be proved by their own deduction engine, whereas waiting goals are not actively proved, but combined with passive items by means of the completion rule of Earley Deduction.

Whenever a unit clause is combined with a non-unit clause all goals up to the first waiting goal of the resulting clause are proved according to their goal type, and then a new clause is added whose selected goal is the first waiting goal.

In the following inference rule for clauses with mixed goal types, Ξ is a (possibly empty) sequence of goals without any waiting goals, and Ω is a (possibly empty) sequence of goals starting with a waiting goal. σ is the most general unifier of G and G', and the substitution τ is the solution which results from proving the sequence of goals Ξ.

$$\frac{\begin{array}{c}\langle X \leftarrow G \wedge \Xi \wedge \Omega, I1\rangle \\ \langle G' \leftarrow, I2\rangle\end{array}}{\langle \tau\sigma(X \leftarrow \Omega), I1 \star I2\rangle} \tag{3.3}$$

A description of the different goal types, and the implementation of their inference rules can be found in section 5.4.

procedure add item $\langle C, I1 \rangle$ to chart
– chart := chart $\cup \{\langle C, I1 \rangle\}$
– if C is a unit clause
  – for all items $\langle H \leftarrow G \wedge \Xi \wedge \Omega, I2 \rangle$
    – if $I = I2 \star I1$ exists
      and $\sigma =$ mgu(C, G) exists
      and goals Ξ are provable with solution τ
      then add item $\langle \tau\sigma(H \leftarrow \Omega), I \rangle$ to agenda
– if $C = H \leftarrow G \wedge \Xi \wedge \Omega$ is a non-unit clause
  – for all items $\langle G' \leftarrow, \ I2 \rangle$
    – if $I = I1 \star I2$ exists
      and $\sigma =$ mgu(G, G') exists
      and goals Ξ are provable with solution τ
      then add item $\langle \tau\sigma(H \leftarrow \Omega), I \rangle$ to agenda

Figure 3.7: Procedure add-item for bottom-up Earley deduction with different goal types

Figure 3.7 shows the procedure add-item augmented for the handling of goals with different goal types.

The algorithm is parameterised with respect to the relation `lookup/2` and the choice of the indexing scheme, which are specific for different grammatical theories and directions of processing.

3.2 Earley Deduction for Discontinuous Constituency

Many implemented formalisms rely on a context-free backbone, e.g., DCG, PATR, (old) STUF, ALE, ALEP etc. In LFG, the context-free backbone is even part of the grammatical theory as a separate level of representation called c-structure. If a context-free backbone is used, the operation used on strings is that of concatenation. Earley deduction applied to grammars with a context-free backbone yields the same algorithm as the corresponding chart parser applied to the these grammars.

Recent analyses have used more powerful operations than concatenation (adjoining, head-wrapping, sequence union, etc.). These operations allow very elegant analyses of word order facts, but require modification of parsing algorithms.

Sections 3.2.1 through 3.2.3 present several linguistic frameworks which make use of non-concatenative operations for the combination of strings, and introduce

the operations used.[9]

In general, the rules in grammars which make use of these more powerful string operations have the following form [van Noord, 1993, p. 43]. The relation `cp/2` makes the combination of strings explicit.

$$sign(M) \leftarrow sign(D_1) \wedge sign(D_2) \wedge cp(M, \langle D_1, D_2 \rangle). \tag{3.4}$$

In section 3.2.5, we will show that top-down Earley deduction is not very well suited for these kinds of grammars because of the non-determinism in the prediction step. It is these cases where bottom-up Earley deduction presents the greatest advantages.

3.2.1 Johnson's Combine Operator

Johnson proposes an extension of DCG in order to analyse the Australian free word-order language *Guugu Yimidhirr.* In ordinary DCG, a category is associated with a pair of string positions indicating which portion of a string a constituent covers. Johnson proposes to associate constituents with sets of such pairs, so that a constituent covers a set of continuous substrings of the string. In the sentence 1 of Guugu Yimidhirr, the discontinuous constituent 'Yarraga-aga-mu-n ... biiba-ngun' (boy's father) is associated with the set of string positions $\{\langle 0,1\rangle,\langle 3,4\rangle\}$.

(1) Yarraga-aga-mu-n gudaa dunda-y biiba-ngun
boy-GEN-mu-ERG dog-ABS hit-PAST father-ERG
The boy's father hit the dog

Johnson notes that such expressions can be represented as bit vectors. In a grammar rule, the sets of locations of the daughters of the rule are combined to construct the set of locations associated with the mother node. The predicate *combines*(s_1, s_2, s) is true iff s is equal to the (bit-wise) union of s_1 and s_2, and the (bit-wise) intersection of s_1 and s_2 is null (i.e. s_1 and s_2 must be non-overlapping locations). Grammars which use only this predicate are permutation-closed. Guugu Yimidhirr is said to be permutation-closed [Haviland, 1979], with the exception of possessive-noun constructions (in which the possessive is identified by position rather than by inflectional markings), for which Johnson proposes a concatenative rule.

The implementation of this kind of grammar in the framework of bottom-up Earley deduction is straightforward.

Johnson's *combine* operation is directly implemented in the indexing scheme 2. This indexing scheme performs exactly the same combination of non-overlapping string positions as Johnson's combine operation.

[9] These presentation in these sections is partly based on the description of these operations in [van Noord, 1993, pp. 103 – 119]

The lookup step will add all lexical entries of all words in the input string. Since the combine operation is non-erasing, no other lexical entries must be added to the chart. All empty categories and grammar rules must also be present in the chart.

3.2.2 Head-Wrapping

Pollard defines a grammatical formalism called Head Grammar [Pollard, 1984] that is slightly more powerful than context-free grammars.[10] The formalism makes use of a *head wrapping* operation for combining strings. In Head Grammars, strings contain a distinguished element, the *head*. Such headed strings are a pair of an ordinary string and a pointer to the head. For example, the string $w_1 w_2 w_3 w_4$ with head w_3 is represented as $\langle w_1 w_2 w_3 w_4, 3\rangle$. String operations take n headed strings as arguments and return a headed string. A simple example is an operation (labelled LC1 by Pollard) which takes two headed strings and concatenates the first to the left of the second and makes the head of the first string the head of the combined string.

$$\text{LC1}(\langle\sigma,\text{i}\rangle, \langle\tau,\text{j}\rangle) = \langle\sigma\tau,\text{i}\rangle \tag{3.5}$$

RL2 is an operation more powerful than concatenation, in which the second argument is 'wrapped' around the first.

$$\text{RL2}(\langle\sigma,\text{j}\rangle, \langle t_1 \dots t_n,\text{i}\rangle) = \langle t_1 \dots t_i \sigma t_{i+1} \dots t_n,\text{i}\rangle \tag{3.6}$$

This 'head wrapping' operation is used in a rule for English auxiliary inversion, which analyses the string *Must Kim go* as derived from the noun phrase *Kim* and the (discontinuous) verb phrase *must go* with head *must*.

$$S[+INV] \rightarrow RL2(NP, VP[+AUX]) \tag{3.7}$$

Another example are the analyses of the sentences (2) and (3), in which *easy to please* is one (discontinuous) constituent.

(2) Kim is very easy to please

(3) Kim is a very easy person to please

Pollard also presents a wrapping analysis of Dutch cross-serial dependencies, which van Noord adapts as input for his head-corner parser.

In order to apply bottom-up Earley deduction to this grammar, we can make use of the usual operation for lookup which adds lexical entries for all the words

[10] The languages generated by Head Grammars belong to the class of *mildly context-sensitive languages,* which can be parsed in time $O(n^6)$, where n is the number of words in the input string. Other members of this class are Tree Adjoining Grammars and Linear Indexed Grammars. It has been argued that all natural languages are in the class of *mildly context-sensitive languages.*

in the input string as items to the chart, since head wrapping is a non-erasing operation.

We choose indexing scheme 2, which combines non-overlapping string positions. It would be tempting to make use of an indexing scheme which is more specialised for headed strings, but this is problematic because it is not known in advance which of the several operations for combining strings will be used, and which element will be the head of the resulting string.

3.2.3 Sequence Union

Mike Reape has developed an analysis of clause union and verb raising phenomena in German that makes use of a powerful string operation called sequence union [Reape, 1990; Reape, 1993a; Reape, 1993b; Reape, 1994]. A similar analysis has been proposed by Dowty [Dowty, 1993].

The sequence union of two sequences X and Y is a sequence Z which contains all the elements from X and Y, and in which the order of the elements of X and the order of the elements of Y is preserved. Sequence union is a non-deterministic operation: the sequence union of the two sequences ⟨a,b⟩ and ⟨c,d⟩ is any of the following:

⟨a,b,c,d⟩
⟨a,c,b,d⟩
⟨a,c,d,b⟩
⟨c,a,b,d⟩
⟨c,a,d,b⟩
⟨c,d,a,b⟩

In our extended constraint language [Manandhar, 1995], which permits underspecified representation of linear order, the sequence union operation can be applied deterministically, and the output is a set of elements which are associated with a set of ordering constraints.

Reape applies sequence union to *word order domains* in HPSG grammars for German and Dutch. In the rules, the word order domain of the mother is defined in terms of the word order domains of its daughters. In normal rules, the order domain of the mother consists just of the order domains of the daughters. In specific cases such as clause union, the word order domain of the mother consists of the elements of the order domains of the daughters. The German sentence (4) illustrates a case of domain union.

(4) . . . es_i ihm_j jemand_k zu lesen_i versprochen_j hat_k
. . . it him someone to read promised has
. . . *someone had promised him to read it*

Reape has presented generalisations of various parsing algorithms (top-down, left-corner, (tabular) shift-reduce, CKY) for discontinuous constituency [Reape, 1991]. However, Reape's algorithms are not really specialised for the case of sequence union, but cover the very general case of permutation-closed languages, so that they would be more adequate for Johnson's *combine* operation introduced in section 3.2.1.

For grammars with sequence union, the lookup step for bottom-up Earley deduction is the same as for other grammar formalisms which make use of non-erasing operations for combining strings. Indexing scheme 2 will also be used for grammars with sequence union, as it is the indexing scheme appropriate for grammars which allow non-erasing string operations other than concatenation.

The same lookup relation and indexing schema can also be applied to Tree Adjoining Grammars [Vijay-Shanker *et al.*, 1987; Joshi *et al.*, 1975; Joshi and Vijay-Shanker, 1985]. Van Noord has applied head-driven methods (head-corner parsing and semantic-head-driven generation) to Tree Adjoining Grammars [van Noord, 1993].

Having discussed a number of grammar formalisms which make use of string operations more powerful than concatenation, and shown that they can be handled by means of bottom-up Earley deduction, we now address the question whether bottom-up Earley deduction presents any advantages over other algorithms such as head-corner parsing or top-down Earley deduction.

3.2.4 Necessity for Tabulation

We have already seen that head-driven processing is a uniform bidirectional algorithm, but that it can suffer from efficiency problems and is not very well suited to incremental processing. The efficiency problems stem from the increased non-determinism of the selection of the head element of the phrase, for which there are many more choices than for selection of a lexical entry for the left corner of a string. Any consequences derived from a wrong choice at this step are lost after backtracking. Bouma and van Noord have shown that this problem can be overcome by augmenting a head-driven algorithm with tabulation (well-formed substring tables) [Bouma and van Noord, 1993]. Tabulation also provides a partial solution for the problem of incrementality because an item can be assumed to be the head element, and any wrongly computed consequences of this assumption can be re-used as subproofs of a derivation based on a correct choice of the head element.

As it turns out that tabulation is needed anyhow, the next logical step is to investigate the application of Earley deduction methods to the problem.

3.2.5 Inadequacy of Top-Down Earley Deduction

Neumann [Neumann, 1994b] has shown that top-down Earley deduction is a useful framework for efficient incremental analysis for grammars based on concatenation, but he does not consider grammars based on more powerful string operations. It is easy to see that top-down methods run into problems with more powerful string operations due to the increased non-determinism of the prediction step. Remember the general format of a grammar rule (3.8) as assumed in [van Noord, 1993], and (ideally) produced as the output of the partial deduction algorithm presented in chapter 2.

$$sign(M) \leftarrow sign(D_1) \wedge sign(D_2) \wedge cp(M, \langle D_1, D_2 \rangle). \tag{3.8}$$

The prediction step looks up clauses for the selected goal of an active item. The obvious problem is which of the goals in the body of the clause to select for the prediction step. We will see that there is no choice of a goal that leads to satisfactory results. There are two possible cases: either one of the `sign/1` goals is predicted, or the `cp/2` goal. Prediction for one of the `sign/1` goals will not be constrained at all since the PHON feature in the goal will not be instantiated. As soon as lexical items are predicted and added as passive items, an infinite number of items can be result through the repeated application of the reduction rule. If the `cp/2` goal is chosen, the algorithm will terminate, but since the the `cp/2` goal has a very large number of solutions, it will still lead to efficiency problems through an unnecessarily large number of subsequent predictions for the `sign/1` goals.

For instance, the sequence union operation (cf. section 3.2.3) applied in reverse to divide an input string with n words has 2^n solutions; so for an input string of only 10 words, there are 1024 different solutions.

The attentive reader will not have failed to notice the similarity of this argument with the one that demonstrated the necessity of handling concatenation by encoding concatenation constraints through difference lists (cf. page 17).

Since we conclude that tabulation is needed and that top-down processing is inadequate, the logical choice is to apply bottom-up Earley deduction for grammars with more powerful string operations.

3.2.6 Guides versus Indexing

This section will review the notion of a *guide* [Dymetman *et al.*, 1990a; Martinović, 1994] that is used in left-corner and head-corner processing in order to constrain the possible bottom-up inference steps, and discuss its relationship to the use of indexing in bottom-up Earley deduction.

Guides in head-driven algorithms ensure termination for parsing and generation by requiring that "the processing of a lexical entry consumes some amount of a guide; the guide used for parsing is a list of words remaining to be analysed, while

the guide for generation is a list of the semantics of constituents waiting to be generated" [Dymetman *et al.*, 1990b].

The use of difference lists in processing of DCG can be regarded as an example of the use of guides. When a difference list representation is used, the remaining words of the input string to be processed are immediately available after a word has been processed.

Of course, using guides to ensure termination presupposes that a grammar fulfill syntactic and/or semantic monotonicity requirements. For the phonological string, the monotonicity requirement is that the operation for combining strings be linear and non-erasing.[11] For semantics, a semantic monotonicity requirement has been proposed, which requires that the logical forms of the immediate constituents of a phrase be part of the logical form of the phrase itself.[12]

The representation of guides for parsing can be different for different operations for combining strings. For grammars based on concatenation, the remaining string must be represented in the guide. For head-driven grammars, the remaining strings to the right and to the left of the head must be represented, and for permutation-closed languages, it is enough to represent the remaining words, but abstract away from their linear order.[13]

The concept of *guides* is not directly applicable to bottom-up Earley deduction because the state of a guide at a particular point of the computation represents how much of the input has been consumed at that point. In bottom-up Earley deduction, the same item will be used again for different derivations, in which different parts of the input will have been processed. Therefore, the representation of items must fulfill the following requirements:

- The representation of each item must exhibit which portion of the input is consumed by that item.

Moreover, the representation must ensure

- that no portion of the input is consumed twice, and
- that no lexical entries are consumed whose phonology or semantics is not part of the input.

In context-free chart parsing, such a representation exists in the form of the string positions that are part of each item. The string position associated with each item represents the part of the input string that is consumed by that item.

---

[11] For a discussion of these properties, see [Vijay-Shanker *et al.*, 1987] and [Weir, 1988].

[12] However, the requirement of semantic monotonicity is too strong to account for non-compositional phenomena in natural languages, such as idiomaticity. This point is further elaborated in section 3.3.2.

[13] Van Noord proposes a representation of guides with the bag datatype [van Noord, 1993, p.148], as defined in the Quintus Prolog library bag.pl by O'Keefe.

The combination of indices mirrors concatenation of the PHON values represented by the indices. If the input string is also represented as a difference list in the clause part of the item, then the string position serves as a redundant index to that information.

This kind of indexing is generalised to handle operations for combining strings other than concatenation, and to handle composition of logical forms for generation.

Indexing and combination of indices serve the same purpose as guides, namely to avoid useless deduction steps and to ensure termination. While in left-corner or head-driven processing, the parts of the guide are consumed one by one, the lookup step turns them all at once into items. The second purpose of guides, making sure each part of the guide is consumed only once, is ensured by the operations for the combination of indices. The last purpose to which guides can be put, namely to ensure that *only* the parts of the guide are consumed, is guaranteed by the fact that only those lexical signs added by the lookup relation can be used as input to the deduction process.

It is important to note that the information represented in the index of an item could be redundantly added to the clause part of the item. For example, the clause part of an item in context-free chart parsing could carry information about the part of the input string that it covers, about the part of the input string that follows it and the part of the input string that precedes it. Similarly, for a permutation-closed language, the clause part can have information about the part of the input string that it covers and the part that is not covered by it. For generation, the clause part of an item can carry information about the part of the semantic content it covers and the part that still has to be generated. The clauses for combining constituents would then have to be augmented to handle this part of the information structure appropriately. This amounts to instantiating the original program clauses further for a particular parsing or generation problem.[14]

This possibility is important for the following argument which claims that the information represented in the index part of an item subsumes the information represented by the clause part, so that the signs denoted by the index part are a superset of the signs denoted by the clause part. The indexing of a clause is an abstraction over the information in the clause.[15]

$$[\![index(C)]\!] \supseteq [\![C]\!] \tag{3.9}$$

The next requirement is that the operation R_i for combining two indices $index(C_1)$ and $index(C_2)$ be constructed in such a way that the information represented by the resulting index $index(C_3)$ subsume the information of the clause

[14]See for example the presentation of DCG parsing in [Pereira and Warren, 1980], where the input string is represented as part of the program.

[15]We choose to ascribe denotations to indices directly. Alternatively, we could define a mapping from indices to sets of clauses, and use the denotation of the clause set.

C_3 that results from the combination of clauses C_1 and C_2 by the corresponding operation R.

In (3.10), this property is given in terms of subsumption, and in (3.11) in terms of denotations.

$$R_i(\text{index}(C_1), \text{index}(C_2)) \sqsupseteq R(C_1, C_2) \tag{3.10}$$

$$[\![R_i(\text{index}(C_1), \text{index}(C_2))]\!] \supseteq [\![R(C_1, C_2)]\!] \tag{3.11}$$

This property becomes important later to show that indexing does not affect completeness because any combination of items that fails due to the failure to combine the indices would have failed anyway due to the impossibility to combine the corresponding clauses (cf. section 3.4.2).

In a practical implementation, one can of course leave the information represented by the index out of the clause in which case it is the index that serves as the instantiation of an item to a particular parsing or generation problem. This is why information about the covered string can be represented only by the index in case of context-free chart parsing. In this case, the index does not only provide redundant information. Leaving out this information is also a precondition for the re-use of items in the case where the input query is changed destructively (cf. the description of the fully incremental algorithm in section 3.5).

3.3 Application to Generation

An important characteristic of the linguistic deduction approach is its inherent bidirectionality, i.e., its application to both parsing and generation. Examples of bidirectional algorithms are head-driven approaches (head-corner parsing and semantic-head driven generation) [van Noord, 1993] and the uniform tabular algorithm for top-down Earley deduction, which uses the same algorithm, but different indexing for parsing and generation [Neumann, 1994b]. In this section, we will investigate how the bottom-up Earley deduction algorithm can be used for the task of generation.

3.3.1 Semantically Monotonic Grammars

We start out with the case of semantically monotonic grammars. A grammar is called semantically monotonic if the logical forms of all the constituents of a phrase are part of the logical form of the phrase. In this sense, semantic monotonicity it strongly related to the *non-erasing* property of the operations for the combination of strings.

Shieber has proposed a uniform architecture for parsing and generation based on Earley deduction [Shieber, 1988b]. The parsing instance of this architecture is

defined as usual. The generation instance abandons the indexing of items according to the string position, and adds all items for lexical entries at a single position (e.g. $\langle 0,0 \rangle$). An additional filter ensures that only those items are added to the chart that can contribute semantically to the goal meaning. Under the assumption of semantic monotonicity, this means that "the meaning associated with the item must subsume some portion of the goal meaning" [p. 617]. This criterion is related to the use of a reachability relation in *directed* approaches to parsing, which combine bottom-up and top-down processing.

Kay [Kay, 1993] has presented a chart-based bottom-up generator which can be regarded as an instance of the bottom-up Earley deduction algorithm presented here. The generator performs a lookup step which adds those lexical entries as items to the chart whose semantics is a component of the logical form to be generated. The indexing is improved compared to Shieber's generator by making use of the semantics of the items, so that application of the completion rule is only attempted for pairs of active and passive items that can be combined semantically. Like Shieber's uniform parsing and generation algorithm, this generator is only suited for semantically monotonic grammars. This restriction stems from the way in which lookup is performed, namely by a structural decomposition of the input logical form. In case of semantically monotonic grammars, such structural decomposition accurately reflects the compositional functions used in a grammar to build up semantic representations of phrases from the semantic representations of their constituents.

Lookup for generation. Lookup proceeds by decomposition of a logical form into its components. The component of a logical form LF is either

- the logical form LF itself, or
- a component of an argument of the logical form LF (or a component of the value of a thematic-role feature, such as AGENT or PATIENT).

Figure 3.8 shows a possible logical form for the sentence *The old professor persuades a young student to buy many expensive books*, and a list of the components of this logical form.

If a logical form contains quantifiers, these must be taken into account in the decomposition procedure. Consider the logical form for the unmarked reading the sentence *Every man loves a woman.*

$$\forall X[man(X) \supset \exists Y woman(Y) \wedge love(X,Y)]$$

This logical form should has the following components:

LF: persuade(def(old(professor(
Y ∧ indef(young(student(Y))),
buy(Y,many(expensive(book(Z)))))

Components: persuade(_,_,_)
def(_)
old(_)
professor(_)
indef(_)
young(_)
student(_)
buy(_,_)
many(_)
expensive(_)
book(_)

Figure 3.8: A logical form and its components

$\forall X A \supset B$
$man(_)$
$\exists Y A \wedge B$
$woman(_)$
$love(_, _)$

This decomposition can be achieved by the following decomposition rules, which take into account the structure of quantified expressions. The decomposition of a logical form consisting of functor and arguments is a special case given in the last line. The decomposition rules are applied recursively to the components.

Formula	*Components*
$\forall X A \supset B$	universal quantifier,A, B
$\exists Y A \wedge B$	existential quantifier, A, B
$A \wedge B$	conjunction, A, B
$f(Arg_1, \ldots, Arg_n)$	$f(Arg_1, \ldots, Arg_n), Arg_1, \ldots, Arg_n$

Lookup for a goal with logical form LF proceeds by adding an item to the chart for each clause of the grammar whose consequent matches a component of the logical form LF. Semantically empty lexical items must also be added to the chart.

Since a given logical form LF has only finitely many components, and there are only finitely many clauses in the program, only a finite number of items are added at this step (in the worst case the number of components of LF times the number of program clauses).

Note that this specification of the lookup relation restricts the introduction of non-chain rules to those that are actually needed for proving a given query. Only those non-chain rules are added whose consequent matches a component of the logical form. By this definition, all chain rules are added since their consequent will match any logical form because a chain rule simply states that its mother node's (consequent's) logical form is the same as that of the head daughter, but does not make any restrictions on the content of the logical form. This restriction of the introduction of non-chain rules improves upon Kay's generator, where all grammar rules are present in the chart.

Indexing for Generation. Indexing must make sure that only items are combined which can contribute to a proof of the goal. For generation, we index the items according to the components of the input logical form (which can for example be achieved by giving each component a unique identifier).

For the combination of indices, we use the indexing scheme 1 (non-reuse of items) which ensures that no constituents are built in which the same component of the logical form of the input is used more than once. Termination is ensured because only finitely many derivations are possible without re-use of items (cf. section 3.4.3).

However, this method for indexing is not restrictive enough because it allows constituents to be constructed with a logical form that is not a component of the input. In order to improve the efficiency of the generator, an additional filtering step, as proposed in [Shieber, 1988a] must be added which prevents such items from being added to the chart.

3.3.2 Semantically Non-Monotonic Grammars

The method described above for doing lookup by a structural decomposition of the logical form is not applicable to grammars which contain semantically non-monotonic constructions, such as idiomatic expressions. By *semantically non-monotonic* constructions, we mean constructions in which logical forms are built up in such a way that constituents may contain logical forms which do not occur as components of the logical forms of the constituents that dominate them, i.e., these constructions don't enjoy a "non-erasing" property. Semantically non-monotonic constructions present a problem because the lookup step must produce items whose logical form is not a component of the logical form of the input. We will therefore propose an alternative lookup step for grammars containing semantically non-

monotonic constructions.[16]

A classic example of a semantically non-monotonic construction is the idiom *kick the bucket.* We assume a lexicalisation of such constructions [van der Linden, 1993; Abeillé, 1994; Erbach and Krenn, 1994], so that the information about the idiom is represented in the lexical entry of its head word *kick,* whose schematic lexical entry is shown in (3.12).

$$\text{lex}\left(\text{kick}, \begin{bmatrix} \text{lf:} & \text{die(Subj)} \\ \text{subcat:} & \langle\, \text{Subj,[lf:def(bucket)]} \,\rangle \end{bmatrix}\right). \tag{3.12}$$

Both semantic-head driven generation (SHDG) and Neumann's uniform tabular algorithm (UTA) use a head-driven strategy in order to deal with this problem. In SHDG, the semantic head of a phrase is given a special status by the compilation of grammars into chain rules and non-chain rules, and in the UTA the head-driven behaviour results from the choice of the selection function for the generation case: process (predict) that goal first in which the logical form feature is instantiated, which is typically the semantic head.

When these head-driven algorithms need to generate a sign with a given logical form, they first generate the head daughter, and then recursively the other daughters. When the logical form *die(john)* needs to be generated, the head daughter will be the lexical entry for *kick (the bucket)*, which instantiates the subject daughter to a nominative NP with the logical form *john* and the object daughter to an accusative NP with the logical form *def(bucket)*.[17]

Lookup for generation with semantically non-monotonic grammars. In order to define a lookup step for semantically non-monotonic grammars, a simple structural decomposition of the input logical form is not enough. Instead, the rules and lexical entries of the grammar must be used for deciding which items should be added during the lookup step.

The lookup step can be formulated in a straightforward fashion similar to the head-driven algorithms by instantiating the consequent of a non-chain rule (i.e., the mother) with a goal, and performing lookup recursively on the goals in the antecedent of the clause (i.e., the daughters). In the case of chain rules, the

[16] In general, generation for semantically non-monotonic grammars cannot be guaranteed because the compositional functions for constructing logical forms can be arbitrarily complex [Zadrozny, 1992]. Therefore, we will concentrate on some typical instances of non-monotonic rules that occur in real grammars.

[17] It would be possible to emulate a head-driven algorithm within our algorithm by selecting a matching non-chain rule during the lookup step, and proving the goals of the antecedent of the non-chain rule by any proof procedure, e.g. bottom-up Earley deduction. The consequent of the non-chain rule would then be added as an item to the chart. Since this algorithm does not differ in any interesting way from semantic-head driven generation augmented with memoing, we will not pursue it any further.

```
lookup(lf:love(X,Y), Lex ) :- lex(Lex&phon:⟨love⟩).
lookup(lf:love(X,Y), Clause ) :- lookup(lf:X,Clause).
lookup(lf:love(X,Y), Clause ) :- lookup(lf:Y,Clause).

lookup(lf:die(X), Lex ) :- lex( Lex& [ phon: ⟨kick⟩
                                       lf:   die(_) ] ).
lookup(lf:die(X), Clause ) :- lookup(lf:X,Clause).
lookup(lf:die(X), Clause ) :- lookup(lf:def(bucket),Clause).
```

Figure 3.9: Precompiled lookup relation for semantically non-monotonic grammars

consequent is instantiated with the goal, *the head daughter with a lexical entry*, and then the lookup is performed recursively on the other daughters.

In terms of implementation, this step is not straightforward because it involves a lot of search, and may return the same solution more than once. In order to overcome these problem, it would be necessary to compile the lookup step into a table through which can be used to perform the lookup at runtime.[18] The clauses in figure 3.9 illustrate the precompiled lookup step for a semantically monotonic construction (the verb *love*) and a non-monotonic construction (*kick the bucket*).

Indexing for generation with semantically non-monotonic grammars. The same indexing scheme will be used as for semantically monotonic grammars. Even though this indexing scheme ensures termination, it still leads to useless search because it allows the creation of useless items.

3.3.3 Conclusion on Generation

By defining a lookup step for semantically non-monotonic grammars, we have overcome the most serious obstacle for bottom-up chart-based generation algorithms. These algorithms can be used for cases of incremental generation where the input becomes known gradually, e.g., in the case of interactive machine translation.

In terms of efficiency, however, generation algorithms which look up lexical entries dynamically, such as SHDG or UTA are superior to bottom-up Earley deduction for generation. This is due the fact that an algorithm with dynamic lookup (prediction) will produce fewer items in the lookup step because syntactic and morphological information gets instantiated before the prediction is done. As a result, less search is needed in algorithms which do prediction dynamically.

[18] We will not provide details of the compilation step here. A related technique for compiling goto and reduce tables for an LR-style generation algorithm is described in [Samuelsson, 1994a].

This is in contrast to the case of parsing with discontinuous constituents, where the use of a lookup step reduces the amount of search that would be needed for a top-down algorithm with prediction.

Moreover, the indexing scheme defined for generation allows the production of items which are useless to the proof of a goal, so that bottom-up Earley deduction in its current form does not appear as an efficient algorithm for generation unless further improvements are made.

3.4 Properties of the Algorithm

In this section, we will discuss the three fundamental properties of the algorithm: correctness, completeness and termination. In addition, we will touch on questions of complexity.

Clearly, completeness and especially termination depend on the complexity properties of the particular grammatical theory to which the bottom-up Earley deduction algorithm is applied. If a grammar is undecidable, i.e., allows infinitely many derivations, then for example termination cannot be guaranteed. The completeness property depends on the specification of the lookup relation and on the chosen indexing scheme. Rather than attempting to show properties such as completeness and termination in general, we will outline the general structure of a completeness argument, and provide the structure of these arguments for the case of HPSG with sequence union, on the basis of particular choices for the lookup relation and the indexing scheme.

3.4.1 Correctness

An deduction algorithm is called *correct* if all the proofs derived by the algorithm are consequences of the program. We will show completeness by showing that any item derived by the algorithm follows from the program.

The arguments proceeds by induction. First, we show that the items added initially are consequences of the program, and then we show that the combination of any two items by the completion rule creates an item which is a consequence of the program. There is no other way to create items.

The items that are added to the chart initially fall into two classes: (i) all non-unit clauses of the program, and (ii) the unit clauses selected by the lookup relation. The non-unit clauses of the program are trivially consequences of the program. The unit clauses added by the lookup relation are either instances of program clauses, in which case they are consequences of the program, or they are proved from the clauses of the program by a resolution theorem prover (e.g., the Prolog proof strategy), for which it is also known that it produces only consequences of the program clauses.

In order to show that the combination of an active and a passive item by the completion rule (3.3) is a consequence of the program, we must show this (i) for the combination of the first waiting goal with the passive item, and (ii) for the proof of the remaining goals with other goal types up to the next waiting goal. The first is just a resolution step, and the proof of the remaining goals proceeds also by resolution, so that the correctness of this step is ensured.

If other goal types occur in a clause, these are processed by resolution algorithms (e.g., SLD resolution) whose correctness is well-known in logic programming.

Since the only items in the chart are either (i) added as non-unit clauses of the program, or (ii) added by the lookup operation, or (iii) produced by application of the completion inference rule (3.3) from items which are consequences of the program, every item is a consequence of the program.

The indices associated with items do not affect correctness, since they are only used as filters that blocks certain possible deduction steps that are useless for the given goal, but does not influence the resulting item.

3.4.2 Completeness

Completeness is a more interesting property because it depends crucially on the lookup relation, and on the chosen indexing scheme. Incompleteness can arise either when the lookup step does not add all the clauses that are needed for a proof, or if the combination of indices prevents the creation of an item that would be needed to prove the goal.

Without lookup and indexing (i.e., if all clauses of the program are items in the chart), the completeness argument would be even simpler than that for regular Earley deduction, since the addition of items is not even restricted by the prediction step.

3.4.2.1 Restrictions on the Lookup Operation

We show that the lookup step preserves completeness for the case of HPSG with sequence union. We make the simplified assumption that an HPSG grammar consists of clauses of the following form (ignoring the `cp/2` goal for the moment), where S stands for the predicate `sign/1`, L for `lexical_sign/1`, and P for `phrasal_sign/1`. We assume binary branching for this example.

$$\begin{array}{l} S \leftarrow L. \\ S \leftarrow P. \\ P \leftarrow S \wedge S. \end{array} \tag{3.13}$$

Given this abstract program, all branches of a proof tree have the following form.

$$S(PS)^*L \tag{3.14}$$

It is easy to see that all leaves of any proof tree are lexical signs (L). Therefore, a specification of the lookup relation is possible that adds only lexical signs to the chart if the goal is to prove that the input is a sign, as in the following abstract specification of the lookup relation.

$$\text{lookup}(S, L). \tag{3.15}$$

Addition of the `cp/2` goals is not problematic since it is not proved by bottom-up methods, but rather by normal top-down resolution for which all needed clauses are present.

We must now further refine the above argument for the case where only a subset of the lexical signs are added to the chart by showing that the lexical signs which are not added as items cannot be leaves of the proof tree.

In the parsing case, the argument builds on the fact that a non-erasing operation is used for the combination of strings. As a consequence, no lexical sign can be part of the proof tree that does not cover a word of the input string.[19] Therefore, words that are not in the input string need not be added as lexical items to the chart.

Completeness for the parsing case

> If every reading of every word in the input string and every non-unit clause of the program are added to the chart, then completeness of parsing is guaranteed.

For the case of generation, the same argument can be made on the basis of the semantic content of the goal, of the lexical signs, and of the compositional operation for the combination of semantic contents of the constituents. In case of a semantic monotonicity requirement, any semantic content of a leaf of the proof tree becomes part of the semantic content of nodes that dominate it. Therefore, any lexical sign whose semantic content is not a component of the semantic content

[19] If a grammar makes use of empty categories, these must also be added to the chart since they can be part of any input string. All grammar rules which introduce lexical entries must also be added to the chart.

of the goal need not be added as items to the chart.

Completeness for semantically monotonic generation

> If every lexical entry for every component of the input logical form and every non-unit clause of the program are added to the chart, then completeness of generation with semantically monotonic grammars is guaranteed.

In the case of semantically non-monotonic grammars, the completeness depends crucially on the fact that the lookup step adds all the lexical entries that are needed for a proof of the goal, even if their semantics is not a component of the logical form to be generated. This is ensured by a precompilation of the lookup relation by making use of the non-chain rules and of lexicalised chain rules of the grammar, as illustrated in section 3.3.2.

Completeness for semantically non-monotonic generation

> If every lexical entry for every decomposition (via non-chain rules and lexicalised chain rules) of the input logical form and every non-unit clause of the program are added to the chart, then completeness of generation with semantically non-monotonic grammars is guaranteed.

We conclude that a specification of the lookup relation preserves completeness if it adds all items that can occur as leaves in the proof tree for a given goal.

3.4.2.2 Restrictions on Indexing

Indexing serves the purpose of preventing the creation of items that cannot be part of the proof of a given goal. In order to show completeness, it must be shown that the indexing operation does not rule out any items that could be part of the proof.

In section 3.2.6, we have argued that indexing is redundant in the sense that it only represents information that is an instantiation of the program clauses for a particular parsing or generation instance. The clearest example of this is the case where the PHON value in a chart parser with a context-free backbone is represented in the clause part as well as in the index part of each item. In this case, the combination of indices will only fail if the combination of the clauses by the fundamental rule also fails for that particular problem instance.

$$[\![\mathit{index}(C)]\!] \supseteq [\![C]\!] \tag{3.16}$$

$$[\![R_i(\mathrm{index}(C_1), \mathrm{index}(C_2))]\!] \supseteq [\![R(C_1, C_2)]\!] \tag{3.17}$$

Hence, if the requirements postulated in section 3.2.6 on the choice of indices in (3.16), and on the operation for combining indices in (3.17) are fulfilled, indexing will only block derivation steps that do not contribute to a proof of the particular problem instance, and the use of indexing does not affect completeness.

3.4.3 Complexity and Termination

The complexity of the algorithm, and hence its possibility for termination, is dependent on particular grammatical theories, and cannot be given in general. For context-free grammars, since the algorithm becomes identical to a bottom-up chart parser, its complexity is known to be $O(n^3)$, where n is the length of the input string. For other grammatical theories, the complexity calculation must be done separately. However, for particular grammars the complexity may be better than the worst-case complexity of the grammar formalism in which they are encoded. This is a question of current research, and recent results concern the relationship between HPSG and TAG [Kasper *et al.*, 1995] and between feature-based grammars and indexed grammars [Burheim, 1995].

The argument for termination is also relative to particular grammatical theories and cannot be given in general. The termination argument requires that proof trees for a given goal cannot exceed a certain size, and that there are only finitely many proof trees smaller or equal to this size. Then we need to show that bottom-up Earley Deduction is an algorithm for enumerating a subset of the trees of this maximal size.

In order to demonstrate that bottom-up Earley deduction only produces a finite number of proof trees up to a maximal size for a given goal, we need to show that

(i) the lookup step for a goal produces only a finite number of passive items,

(ii) indexing ensures that the size of a derivation tree is bounded by the number of items with distinct indices produced by the lookup step, and

(iii) there are only a finite number of proof trees smaller or equal to a given maximal size.

For the parsing case, (i) follows from the fact

- that there are finitely many words in the input string,
- that each word is only finitely ambiguous,
- that lookup adds only lexical signs for the words in the input string, and
- that there are only finitely many empty categories.

For the generation case, the argument is similar, with the notable difference

- that there are only finitely many components of a given logical form in the goal,
- that there are only finitely many lexical entries that match each logical form, and
- that lookup adds only lexical signs for words whose semantic content is a component of the goal's semantic content.

Part (ii) of the argument, the fact that proof trees for a given goal have a maximal size rests on certain properties of the grammar, most importantly that it does not make unlimited use of cyclic derivations, and restricts the introduction of empty categories. For LFG, this property is formulated as the off-line parsability constraint. For HPSG, it can be shown by an analysis of the rule schemata and principles.

Part (iii) of the argument, that there are only a finite number of proof trees which are equal or smaller than this size, cannot be made in general, since in feature-based grammars the number of non-terminal symbols is potentially infinite. Once again, this must be shown for particular grammars. It would follow from the fact that a derivation of a given size would also put an upper bound on the complexity of the recursively defined feature structures involved in it.

From the premises (i), (ii), and (iii), it follows that there are only a finite number of proof trees for a given goal. Now it remains to be shown that the algorithm enumerates a subset of this finite set of proof trees. This fact is guaranteed by the use of indexing. The indexing schemes presented in figure 3.4 don't allow the use of any item (except the initial active items created from the non-unit clauses of the program, which have the "free" index) more than once in a derivation, and the use of non-branching trees is strictly restricted in a grammar that obeys the offline-parsability constraint (or an equivalent of it). Therefore, the available passive items will eventually get "used up," so that the proof tree cannot grow beyond a maximal size (measured in the number of nodes) on the order of $O(2*W*E*N)$, where W is the number of words returned by the lookup relation, E is the number of empty categories present in the grammar, N is the maximal length of non-branching derivations, and the factor 2 is the number of nodes which are constructed by branching derivations.

Head-driven algorithms and Neumann's uniform tabular algorithm have termination problems in case of grammars which make use of empty heads, e.g., analyses of verb-second phenomena in Dutch or German (e.g., [Netter, 1992]). These termination problems arise because the lexical entry of the empty head does not place any constraints on its complements, so that prediction and completion loops can occur.

For the bottom-up algorithm presented here, empty heads do not engender such termination problems because even though the empty head does not constrain its

arguments, it can only combine with a finite number of them arising from bottom-up steps, and prediction of the unconstrained elements is never attempted.

Nevertheless, instantiation of the empty head with the potential fillers can reduce the search space by pruning of derivations. This can be achieved by delaying the lookup of empty heads until all other lookups have been done, and then unifying the empty head with the potential fillers found among the other predicted items. Such a procedure has been suggested by Johnson and Kay, who introduce empty elements only if there is a "sponsor" for the empty element [Johnson and Kay, 1994].

3.5 Incrementality

In this section, we will argue that Earley deduction is very well suited for incremental linguistic deduction. Incremental linguistic deduction means that as more of the input (phonetic form for parsing, and semantic representation for generation) becomes known, the appropriate deduction steps are performed immediately, even if the complete input is not yet completely known. Stated in terms of logic programming, incremental deduction means that further instantiation of the query leads to further deduction steps that can contribute to a proof of the query. We also discuss the question of "full incrementality," that is the situation where the input is not only further instantiated, but can also be modified destructively.

3.5.1 Motivation

Incremental parsing of natural language input is a crucial to make NL applications more acceptable to the user by reducing the waiting time to a minimum. This is achieved by performing most of the computation while the input is still being entered. This is particularly important in applications with speech input where the parser has to deal with a large number of concurrent word hypotheses.[20]

The need for incremental generation is not quite as evident since there are not so many scenarios where the input to a generation component is entered incrementally. One situation in which both incremental parsing and generation are important is machine-translation of face-to-face dialogues (as pursued in projects like the interactive telephony translation at ATR, or Verbmobil [Wahlster, 1993]), where output should already be produced as the input is still being processed. Of course, incremental generation is also important as a potential model of human cognitive processes.

[20] Just like interactive graphical user interfaces have become a reality as soon as enough computing power has become available to perform the necessary tasks (cursor movement etc.) in real time, natural language interfaces will become widespread when there is enough computing power to support incremental real-time NL understanding.

In his study of incremental natural language analysis, Wirén makes a distinction between two senses in which the term "incremental" is used [Wirén, 1992, p. 2]:

1. Analysis of piecemeal, *left-to-right*[21] extensions of a text. We call this *left-to-right incrementality*, or *LR incrementality* for short.

2. Analysis of *arbitrary* piecemeal changes (insertions, deletions and replacements) of a text. We call this *full incrementality.*

LR incrementality is a special case of full incrementality.

Full incrementality is useful when the input can change while it is being processed. This is for example the case in word processing. An application are systems for grammar checking that analyse a text to check it for errors and suggest corrections while the text is still being entered and edited.

3.5.2 Left-Right Incrementality

Wirten's algorithm for handling incremental changes in the input is based on chart parsing. LR incrementality can easily be handled with a standard chart parser, which operates as usual, with the exception that new input which is added to the right of an already analysed string causes the application of the scanning step to that new input, and the consequent addition of new items, which combine with the old items in the usual way by means of the completion rule. LR incrementality is compatible with both bottom-up and top-down processing, and with arbitrary search strategies.

LR incrementality has often been used in interleaved approaches, in which syntax and semantics work in parallel such that each word or phrase is given an interpretation immediately upon being recognised[22]

Incremental algorithms in which the input is processed as soon as it is received have also been developed for natural language generation[23]

3.5.2.1 Earley Deduction and LR Incrementality

In the Linguistic Deduction framework, incremental input corresponds to further instantiation of the query. As the query gets instantiated, more items are added to the chart and their consequences computed. This is of course contrary to the traditional intuitions about logic programming, where a more fully instantiated query leads to fewer solutions and vice versa. However, in a situation where an insufficiently instantiated query leads to an infinite set of solutions it is necessary to

[21] More precisely front-to-back or beginning-to-end.

[22] Cf.[Mellish, 1983; Mellish, 1985; Bobrow and Webber, 1980; Haddock, 1987].

[23] Cf. [Finkler and Neumann, 1989; Neumann and Finkler, 1990; Harbusch *et al.*, 1991; Reithinger, 1992a; Reithinger, 1992b].

wait until (some part of) the input is instantiated to permit further computation steps. LR-incrementality can be implemented by the use of delayed goals and guarded constraints in the definition of the lookup relation.

Earley Deduction is a suitable framework for LR incrementality, because its basic operation (completion) involves the addition of information and the computation of its consequences.

3.5.3 "Full" Incrementality

Wirén has developed a theory of "fully incremental" chart parsing [Wirén, 1992; Wirén and Rönnquist, 1993; Wirén, 1994], which covers not only the case where words are incrementally added to the input string, but also the case where words are removed or replaced.

The key idea of the algorithm is to record dependencies between items, so that any items which become invalid through a change in the input string can be retracted by following the dependency links. The dependency relation is a binary relation on the set of chart items and the set of tokens in the input. The following dependencies are induced by the three operations of a chart parser:

- A *predicted item* depends on the item that has triggered it and on the items that have redundantly proposed it, unless it is an initial top-down prediction, which does not depend on any item.
- A *combined item* depends on the active and and inactive item that have formed it.
- A *lexical item* depends on the token in the input that triggered it.

On the basis of the dependency relation, Wirén and Rönnquist define the disturbance set of a token in the input as the set of items which are in the transitive close of the dependency relation for that token. If a token is deleted from the input the items in its disturbance set must be removed from the chart. In the case of the insertion of a token, all items which cross the point of the insertion must be removed, a new edge must be added to the chart, and a scanning step must be performed for the inserted item. As a result of the scanning step, a new lexical item is added to the chart which combines with the existing items.

In later work, the fully incremental algorithm is improved by reducing the disturbance set [Wirén and Rönnquist, 1993], and by presenting an algorithm for bounded incremental parsing, whose complexity is bounded in the size of the change of the input and the change of the chart. The bounded algorithm treats the disturbed edges as "sleeping," but does not remove them so that they can be re-used. The notion of boundedness is a recently developed criterion in the study of incremental computation [Ramalingam and Reps, 1991], and Wirén's work is its first application to natural language processing.

There is a close relationship between fully incremental chart parsing algorithms and Assumption-Based Truth Maintenance Systems (ATMS) [de Kleer, 1986], which is discussed in detail in [Wirén, 1992].

3.5.3.1 Earley Deduction and Full Incrementality

In the Earley deduction framework used here, the question arises whether it is necessary to use such a fully incremental algorithm. After all, all items in the chart are valid consequences of the grammar, even if they are not solutions to the given query. So, from a logical point of view, it is not necessary to retract any items.

But, as argued in chapter 3, the main point of entering only a subset of the program clauses into a chart for bottom-up deduction is to make the deduction process more efficient by considering only those clauses which can be part of the proof tree of the given goal.

If we allow, in a fully incremental setting, items in the chart which can no longer contribute to a proof of the given goal, then efficiency will degrade. This is due to the incremental nature of the algorithm; the "useless" items can nevertheless combine with other items to produce more items that are equally useless for proving the given query.

It is such useless combination of items that needs to be prevented. In order to do this, we keep a record of dependencies between items. The dependency relation is defined as follows:

> An item X depends on another item X' if X has been produced by applying the fundamental rule to X' and another item. The dependency relation is reflexive and transitive. An item depends on itself and on any items on which the items from which it was constructed depend.

The information about the dependency relation can be stored with each item.

3.5.3.2 Recomputation of Indices

So far, only the clause part of the items has been considered, but not the index part. The clause part need not be changed, but it is only necessary to prevent further combination of items that are dependent on items which are no longer relevant. It must be noted that the re-use of items is only possible at all because some information is not instantiated, e.g., information about left and right context in case of lexical signs. Part of this information is represented in the index part of the items (cf. section 3.2.6).

In the following example we show what would happen to a unit clause if the context were instantiated during lookup.

lexical_sign(phon:⟨einen,dicken,Hund,begraben⟩-⟨dicken,Hund,begraben⟩)

lexical_sign(phon:⟨einen,Hund,begraben⟩-⟨Hund,begraben⟩)

The example is taken from a parsing problem; strings are represented by difference lists. The first line of the example shows the lexical entry for the word *einen* in the context of parsing the sentence: *... einen dicken Hund begraben*. The second line shows the same lexical entry after the word *dicken* is removed from the input. Such a representation with context would not allow re-use of items since they are different for each problem instance.

In the next example, the information about the context is not instantiated. Therefore, the item can be re-used even if the context of the query changes. The information about the context will be represented in the index of the corresponding item.

lexical_sign(phon:⟨einen | A⟩-A)

When the query is changed, e.g., by removing a word in a parsing instance, the indices of items must change accordingly because they would otherwise disallow the combination of items that should be able to combine. The indices of all items which depend on items with changed indices must be recomputed.

In the following we present the data structures for representing items and indices that supports this kind of incremental recomputation. The key idea is that the same derivation can occur several times, but with different indices. In order to uniquely identify a derivation, we store the derivation tree with each item. A prerequisite for this is that the clauses from which an item is constructed can be uniquely identified. Therefore each clause of the program is associated with a unique identifier.

Each item is uniquely identified by its derivation tree. The representation of the items will be as a pair of clause and derivation:

$$\langle \text{Clause, Derivation} \rangle \tag{3.18}$$

Since the derivation is frequently used in the algorithm to access items, special attention must be paid to a representation of the derivations so that two derivations can be compared with a minimum of computational effort.

Note that the information about the index is missing because this is the part that may, or may not, change when indices must be recomputed. When an item is removed, it is enough to remove the information about its index to make it unavailable for further computation while the representation of the item itself can remain intact to be re-used later. This has useful applications for text editing operations where some part of an input string is moved (deleted in one place and inserted in another).

The fully incremental algorithm operates on three data structures:

Chart Item. An item in the chart is a pair ⟨Clause, Derivation⟩. This information is persistent and can be re-used with different indices. Derivation is either the identifier of a program clause or a pair of the derivations of the active and the passive item from which the chart item is constructed. The derivation of an item uniquely identifies the item.

Index Set. An entry in the index set is a triple ⟨Derivation,Active,Index⟩. Derivation uniquely identifies an item, Active is *act* for active items and *pas* for passive items, Index is an index as introduced in section 3.1.2. This set is used to establish a relation between chart entries and their indices. This information is destructively modified when items are removed.

Dependency Set. An entry in the dependency set is a pair ⟨Derivation,Derivation0⟩ of derivations which uniquely identifies a pair of items. It is used in the operation `remove-item` to remove all entries in the index set which depend on an item that is removed.

In figure 3.10 we show the modified algorithm for fully incremental bottom-up Earley deduction. In order to simplify the description of the algorithm, we do not use an agenda for the fully incremental algorithm. This avoids the necessity to remove information from the chart and from the agenda in case of an update. It also avoids the necessity to pass information about items and about index sets through the agenda. There is, however, no principled problem with an agenda-based fully incremental algorithm.

The procedure add-item(Item,Index) takes an item and an index, and looks for other items with which the item can be combined by checking whether the indices of the items can be combined. If the indices can be combined, the procedure construct-item(Active,Passive,Derivation,Index) is called to take care of the combination of the items.

Since an item with the same derivation, but possibly a different index, may already have been added to the chart, the procedure construct-item first checks if such an item already exists, and updates the index set to associate the existing item with the new index. Also the procedure add-item is called since the new item may be able to combine with other items with which it could not have been combined with its previous index. If no item with the same derivation exists, the first goal of the active item is unified with the passive item, and if the unification succeeds, the chart, the index set and the dependency set are updated. Since the dependencies are between items, not between indices, they don't have to be updated in the case where an item already exists.

The principal difference to the algorithm given in chapter 3 is that the fully incremental algorithm always checks whether an item with a given derivation already exists before trying to construct it. If such an item already exists, only the

procedure add-item($\langle C, D0\rangle$,$Idx0$)
- if C is a unit clause
- then
  for every $\langle D1, act, Idx1\rangle$ in index-set such that $Idx = Idx1 \star Idx0$ exists
  - for the item $\langle NUC, D1\rangle$
    - construct-item($NUC, C, \langle D1, D0\rangle, Idx$)
- if C is a non-unit clause
- then
  for every $\langle D1, pas, Idx1\rangle$ in index-set such that $Idx = Idx0 \star Idx1$ exists
  - for the item $\langle UC, D1\rangle$
    - construct-item($C, UC, \langle D0, D1\rangle, Idx$)

procedure construct-item($H \leftarrow G \wedge \Omega, UC, \langle D0, D1\rangle, Idx$)
- if an item $\langle C, \langle D0, D1\rangle\rangle$ exists
  then
  - if C is a unit clause, then $A =$ pas
  - else if C is a non-unit clause, then $A =$ act
  - index-set := index-set $\cup$ $\langle\langle D0, D1\rangle, A, Idx\rangle$
  - add-item($\langle$C,$\langle$D0,D1$\rangle\rangle$, Idx)
- else if $\sigma =$ mgu(UC, G) exists
  - if $H \leftarrow \Omega$ is a unit clause, then $A =$ pas
  - else if $H \leftarrow \Omega$ is a non-unit clause, then $A =$ act
  - chart := chart $\cup \{\langle H \leftarrow \Omega,\ \langle D0, D1\rangle\rangle\}$
  - index-set := index-set $\cup \{\langle\langle D0, D1\rangle, A, Idx\rangle\}$
  - dependency-set := dependency-set $\cup$ $\{\langle D0, \langle D0, D1\rangle\rangle\}$
  - dependency-set := dependency-set $\cup$ $\{\langle D1, \langle D0, D1\rangle\rangle\}$
  - add-item($\langle$C,$\langle$D0,D1$\rangle\rangle$, Idx)

procedure remove-item($\langle C, D\rangle, Idx0$)
- index-set := index-set / $\{\langle D1, A, Idx\rangle \mid$ depends-on$(D1, D) \wedge$
  subindex$(Idx0, Idx)\}$

Figure 3.10: Algorithm for fully incremental bottom-up Earley deduction

procedure update(Query)
- if L exists then L_{old} := L else $L_{old} := \emptyset$
- create lookup set L for Query
- for every element ⟨C,Deriv,Index⟩ in L/L_{old}
  - add-item(⟨C,Deriv⟩,Index)
- for every element ⟨C,Deriv,Index⟩ in L_{old}/L
  - remove-item(⟨C,Deriv⟩,Index)

Figure 3.11: Procedure for updating the chart after a changed query

index associated with it is manipulated. This allows the optimal reuse of items that have been produced and removed again by an update operation.

Removing an item leaves the representation of the item intact, but the procedure remove-item(Item,Index) only removes the index associated with the item. Since the index is used to determine which items can combine with each other, an item whose index is removed cannot enter into further combinations. The fact that the item is not removed allows its reuse with a different indexing.

The indexing of all items I that logically depend on a given item $I0$ is removed, but only if the index of I is a subindex of the index of $I0$. The subindex relation is the reflexive, transitive closure of the following definition, which states that an index $I0$ is a subindex of any index I that can be constructed by making use of $I0$.

$$\begin{array}{l} subindex(I1, I) \text{ if } I = I0 \star I1 \\ subindex(I0, I) \text{ if } I = I0 \star I1 \end{array} \tag{3.19}$$

The use of the subindex relation is necessary if the same derivation is used at different places in a proof to make sure that only the indices of those occurrences are removed that directly involve the removed item. This situation is the case in parsing when the same word or the same constituents occur more than once in the input string and therefore share items, but not indices in the fully incremental algorithm.

The procedure update(Query) in figure 3.11 takes a new or changed query, and initiates the appropriate actions of adding and removing items in order to update the chart for the new query. The procedure works by creating a *lookup set*, the set of all items which are the output of the lookup relation for the changed query, and comparing it to the lookup set of the previous query. If an item is present in the new lookup set, but not in the old one, then it is added to the chart; and if an item from the old lookup set is not present in the new one, then it is removed. All items which are in the intersection of the old and new lookup sets remain unchanged.

In the concrete implementation, it is advantageous to implement the algorithm

in such a way that it is not necessary to enumerate the entire lookup set and then comparing it to the previous one, but rather to enumerate the differences of the sets directly. In practice this means integration of the lookup and the update procedures into one procedure.

3.5.4 Incremental Addition of Non-Unit Clauses

The preceding sections have shown that unit clauses, typically from the lexicon, can be added incrementally. However, the same is also true of non-unit clauses, i.e., grammar rules, lexical rules etc.

One useful application of this lies in the processing of ill-formed input. Normally, well-formed input is expected, and only the rules which define the well-formed signs of the language are used. If the analysis of a string fails, however, a second, less restrictive, set of rules can be added. Within the Earley deduction framework, this second set of rules can be added to the chart of the failed analysis as non-unit clauses.

3.6 Conclusion

It remains to be seen how bottom-up Earley deduction compares with (and can be combined with) the improved top-down Earley deduction of Dörre [Dörre, 1993], Johnson [Johnson, 1993] and the uniform tabular algorithm (UTA) [Neumann, 1994b], and to head-driven methods with well-formed substring tables [Bouma and van Noord, 1993], and which methods are best suited for which kinds of problems (e.g. parsing, generation, noisy input, incremental processing etc.).

On the question of efficiency, it appears that bottom-up Earley deduction is useful for parsing with discontinuous constituents, for which its performance is comparable to a head-corner parser augmented with memoing. In this case, the bottom-up Earley deduction has an advantage over the UTA. For grammars with a context-free backbone, the performance of bottom-up Earley deduction and the UTA is comparable. For generation, however, the UTA and semantic-head driven generation are at an advantage.

As far as incrementality is concerned, bottom-up Earley deduction has advantages both over head-driven methods and over its top-down counterpart because it can process the input incrementally before the head has been instantiated or before it has been predicted.

One disadvantage of the bottom-up approach is that the `lookup/2` relation must be specified separately for each grammar or program, and an appropriate indexing scheme must be selected. Automatic generation of the lookup relation and selection of the indexing scheme for particular grammars is an interesting and non-trivial problem, which will be the subject of future research.

With the exception of the fully incremental algorithm, the bottom-up Earley deduction system described in this chapter has been fully implemented in Prolog (cf. chapter 5).

Under current Prolog implementations, the use of active items is a source of inefficiency because of the amount of copying that must be performed when an item is stored. Since active items must encode information about the consequent of a clause, *and* about the goals that must still be proven, it is generally substantially larger than a passive item. This is a strong efficiency disadvantage because most of the time in Earley deduction is spent with the copying of constrained terms (feature structures) when items are stored in the chart. For this reason, it is advantageous to restrict the number and the size of the items stored in the chart. In practice, it has turned out that the avoiding of copying can contribute more to efficiency than the storing of partial solutions through active items.

In the case where it is known that a grammar makes only use of binary branching, the algorithm can be improved to avoid the use of active items. We formalise this method through the following inference rule which is specialised for clauses which start with two waiting goals (binary branching), followed by a sequence Ξ of goals with other goal types. This rule is equivalent to the double application of the inference rule 3.2. σ is the most general unifier of $G1$ and $G1'$; ϕ is the most general unifier of $G2$ and $G2'$; and τ is the constraint that is the result of proving the sequence of goals Ξ.

$$\frac{\begin{array}{c}\langle X \leftarrow G1 \wedge G2 \wedge \Xi, I1\rangle \\ \langle G1' \leftarrow, I2\rangle \\ \langle G2' \leftarrow, I3\rangle\end{array}}{\langle \tau\phi\sigma(X \leftarrow), I1 \star (I2 \star I3)\rangle} \tag{3.20}$$

We have not yet done any experiments with bottom-up Earley deduction and really large-coverage grammars. However, the experience of the ALE system and the Babel system [Müller, 1995] suggest that bottom-up Earley deduction without active items is an efficient algorithm for parsing large-coverage HPSG grammars. Müller has implemented a large-coverage HPSG for German, and employs a bottom-up chart parser which allows for combination of discontinuous constituents. His system, which is an instance of the bottom-up Earley deduction scheme described here (making use of indexing scheme 2), is among the fastest HPSG parsers available.

Chapter 4

Preference-Driven Linguistic Deduction

This chapter aims to demonstrate that Bottom-Up Earley Deduction is the right choice for realistic NLP systems because it supports best-first processing (preference-driven linguistic deduction) due to its inherent incrementality discussed in section 3.5. Incremental processing is a property of Earley deduction that is a prerequisite for the implementation of best-first processing. Best-first processing is easy to implement on the basis of an incremental algorithm because the incremental algorithm allows new items to be added at any time; consequently the best (most promising) items can be added first, and the addition of the less promising items can be delayed because they can then be added incrementally if they are needed.

We will introduce a notion of preference value which formalises the criteria used to guide best-first search. The preference value of a structure can be regarded as the probability that it is a sentence of the language described by the grammar. We show the application of preference values to disambiguation (especially making use of word order preferences) and to generation.

4.1 Preferences and Best-First Processing

Models of linguistic competence describe all possible meaning-sound relationships in natural languages. This is generally a many-to-many mapping because one string may express several meanings (ambiguity) and one meaning may be expressed by several strings (paraphrases). In human communication, and in applied NLP systems, it is necessary to choose one of the possible meanings for a given phonetic input in parsing, or one of the possible paraphrases for a given meaning in generation.

Competence models, which are an abstraction from actual language behaviour, must be augmented with performance models in order to form a basis of models of human cognitive activities, such as the choice between different readings of paraphrases. Performance models are also needed to perform similar functions in NLP systems.

In a knowledge-based approach, a system would be endowed with enough knowledge about the language, the communicative situation, the world, the knowledge of its interlocutor etc. to make an intelligent choice between different readings or paraphrases. It is questionable whether all of the required knowledge can be formalised at all, but it is certainly not going to be available for the next generation of NLP systems. Even if the knowledge were available, the required amount of inference would make its use unrealistic for practically applied NLP systems, which are required to be usable on current workstations and personal computers, and to have reasonable response times.

A model of preference that is useful for linguistic engineering, i.e., applied NLP systems, ought to be able to make use of any information that is available for deciding on an order of the produced linguistic deduction (LD) results. This is often not the kind of knowledge needed to make an informed and intelligent choice, but rather statistical information gathered from collections of linguistic data (corpora), or other numeric data which is output from processing modules to give a confidence score on their results. Examples of the available data are the following:

- information about the frequencies of lexical items, or about the frequencies of their readings,
- information about the frequencies of grammar rules or clauses of disjunctively specified principles,
- scored word hypotheses from a speech recogniser,
- output of a spelling corrector with associated penalty scores,
- bigram or trigram statistics from part-of-speech tagging,
- output from semantic processing, e.g. quantifier scoping, selectional restrictions etc.

Our working hypothesis is the following:[1]

Performance = Competence + Preference

[1] In this case, we view performance as the performance of an NLP system in executing its tasks, rather than actual human cognitive activity.

Apart from these data from linguistic engineering, linguistics (in particular theoretical, quantitative and psycholinguistics) has produced numerous results about about the relative acceptability of linguistic derivations, but these results have not yet found an adequate formalisation. The following are just some examples of linguistic work concerned with degrees of acceptability.

- The effect of different word order variations on the acceptability of sentences has been studied in theoretical linguistics (e.g. [Uszkoreit, 1987b], [Hawkins, 1990; Hawkins, 1994]) and investigated in psycholinguistic experiments [Pechmann *et al.*, 1994].

- Attachment preferences have been studied extensively in psycholinguistics (for an overview see [Konieczny *et al.*, 1991]), and have given rise to various parsing models (e.g., [Fodor and Frazier, 1970; Shieber, 1983]).

- Lexical choice, depending on register and context, and the choice of readings of ambiguous lexical entries [Wanner, 1992].

- Word frequencies and collocational properties of words. For example, the conventionality principle for idiomatic expressions ensures that the figurative reading of an idiomatic expression like "kick the bucket" is in general preferred over its literal reading.

- Preferences for quantifier scope assignment. For example, one observation is that quantifiers tend to be scoped in the same order as they appear in the sentence.

- Restrictions on center-embedding have been studied in psycholinguistics, and explained by models of human sentence processing.

- Selectional restrictions and other semantic constraints have been studied extensively.

All of this is very diverse information which cannot readily be integrated. However, it can be noted that a lot of this information is of a statistical nature, and can be interpreted as probabilities. In some cases, it may be possible to re-interpret a numerical value produced by a component (e.g. a speech recogniser or a spelling corrector) as the probability that the item with which this value is associated is the correct one. In the case of spelling correction, the basis for probabilistic reasoning is a model of error production in which certain errors (insertions, deletions, transpositions) occur with a certain probability. Even components which make only binary decisions can be integrated in such a probabilistic model by taking into account the probability that the component's acceptance or rejection of a structure is correct.

Treating preferences as probabilities is a reasonable option because probability theory is mathematically well-understood and makes it possible in principle to relate probabilities of larger structures to the probabilities of their constituents.

It must be noted that multiplication of the probability values assigned by the various components would be appropriate if the different probability values were independent of each other and all components produced equally reliable probabilities. In practice this is not the case, and some fine-tuning will be required in order to arrive at a combination of different probability values which is useful for a particular linguistic engineering task.[2] With these caveats in mind, we interpret the preference of an LD result as its probability.

In a simple processing model, all the LD results would be ordered according to their probabilities after they have been enumerated as solutions to a query. For practical applications, performing a complete search can be too expensive, and the preferences must be used at runtime to follow the most probable paths in the search space. In order for this to happen, the program's clauses (lexical entries, grammar rules, principles) and intermediate LD results (constituents) are also associated with probabilistic preference values. The preference value of a phrase is derived compositionally from the preference values of its constituents.

Before we present a definite clause language augmented with preferences in section 4.1.2, we review some other approaches to preference which have been proposed in the literature.

4.1.1 Models of Preference for Constraint-Based Grammars

We classify the preference models into three categories:

1. **Probabilistic Approaches.** In probabilistic models, preference is expressed through probability values. Structures with a higher probability are preferred.

2. **Processing-Based Approaches.** Processing-based models take processing as the basis for modelling of preference. That LD result which is enumerated first by a particular processing model is the preferred one. Processing-based models are common in psycholinguistic theories.

3. **Other Declarative Approaches.** These models make use of an ordering relation, which is sometimes, but not necessarily based on numerical values.

In the following, we will give an overview of models that have been proposed in each of these approaches.

[2] The problem is similar to that of the use of confidence factors and probabilities in expert systems.

4.1.1.1 Probabilistic Models

A simple example of probabilistic models are the probabilistic context-free grammars [Garside and Leech, 1987; Fujisaki *et al.*, 1991], which associate a probability with each grammar rule (such that the sum of the probabilities for all expansions of one non-terminal symbol is 1), and in which the probability of a derivation is the product of the probabilities of the lexical entries and rules used in the derivation.

A probabilistic model for principle-based grammars, based on logic programming, has been proposed by Eisele [Eisele, 1994]. This model is the most basic probabilistic model that can be assumed for such kinds of grammars. Every goal of a clause (e.g. principles and constituents) is associated with a probability, and the product of these probabilities is the probability of the consequent of the clause.[3]

This model has the advantage that it allows probabilities for rules, lexical entries and different clauses of principles to be estimated from a corpus by counting their frequency. It can therefore be seen as a first step towards a probabilistic model of principle-based grammars that can be applied by making use of data from obtained from corpora. Eisele's approach can account for necessary conditions for the proof of a clause (through goals which have only two solutions: one which has the probability 0 (= failure) and another one which has the probability 1).

While multiplication of probabilities may make sense for constituents of a phrase, whose preference values could be seen as independent, it is a questionable model for the different principles of a principle-based linguistic theory, since they cannot be considered as independent.

Moreover, a model which simply multiplies probabilities cannot account for the fact that the satisfaction or violation of some constraints has a stronger influence on the overall result than other constraints. An example is the Subcat Principle as a really strong constraint, opposed to weaker constraints, such as stylistic factors. What is therefore needed is a notion of weights to model the different strength of influence of different constraints.

Brew proposed another probabilistic model which combines part-of-speech tagging and parsing [Brew, 1993]. Normally tagging is regarded as a kind of preprocessing for parsing, which reduces the search by reducing the number of part-of-speech tags for each word. The problem is that errors of the tagging algorithm may lead to failure of the parser if the tagger fails to assign the tags that the parser needs. In Brew's model, the relationship between parser and tagger is therefore reversed. First the parser enumerates the analyses (on the basis of a pure competence grammar without probabilities), and then the tagger is used to estimate the probabilities of the different sequences of tags that occur as the terminal yield of the parse results, in order to select the analysis which maximises the probability for the tag sequence. On the basis of the knowledge sources which are available today (probabilistic data for part-of-speech tagging, and pure competence models

[3] The same kind of model has been proposed in [Erbach, 1993a], but is superseded by the model proposed in this thesis.

for syntactic parsing), this model provides a way of choosing between different parse results. If a part-of-speech tagger is available which assigns *all possible tags* annotated with probabilities to each word, then the tagger can be used for preprocessing, and the probabilities associated with the tags can be used to guide a best-first search.

In later work, Brew proposes a stochastic HPSG [Brew, 1995]. However, the version of HPSG which Brew considers is far removed from the principle-based HPSG which we have considered so far. His model generalises probabilistic context-free grammars by treating the important problem of re-entrancies (or variable sharing) in grammatical descriptions. In Brew's model, the sorts of an HPSG sort hierarchy play roughly the same role as atomic category labels in context-free grammars.

Attempts to assign probabilities or preference values to feature structures instead of constraints (cf. [Erbach, 1993b]) run into problems because feature structures are simply descriptions of objects, and it is not clear how the probabilities associated with feature values should be combined, especially since they can occur more than once in the same structures due to coreferences.

For building up statistical models, it is necessary to count the frequencies of disjuncts in their context. The open question is the classification of the objects whose probabilities are counted, and how fine-grained the context should be. These choices are important to avoid the sparse data problem.

4.1.1.2 Processing-Based Models

This class of models is intended to explore the search space in such a way that preferred readings are found before the less preferred ones. This is achieved by defining decision criteria for handling non-deterministic processing steps. These decision criteria can be based on statistical probability or structural considerations.

In Prolog-based grammar formalisms like DCG [Pereira and Warren, 1980], the search can be controlled by ordering the clauses in such a way that the preferred clauses are tried first. [Haugeneder and Gehrke, 1986] and [Erbach, 1991a] define arbitrary parsing strategies by assigning priorities to parsing tasks for a chart parser, based on statistics about previous parse results.

An example for the exploitation of structural properties for resolving non-determinism is deterministic parsing [Marcus, 1980], where every choice is deterministic in the sense that no alternatives are ever considered. The choice depends on the phrase structure that has already been built, and on the contents of a buffer of constituents that need to be integrated. Another example is Shieber's use of a shift-reduce parser to model attachment preferences by shifting in case of shift-reduce conflicts, and performing the "longer reduction"[4] in case of reduce-reduce conflicts [Shieber, 1983].

[4]The "longer reduction" is the reduction that involves a grammar rule with more elements on its right-hand side.

The approaches mentioned have been applied in more traditional parsing frameworks, where the effect was mostly to order the rules of the grammar. With the increasing trend towards principle-based grammars, the ordering of rules does not make very much sense, because only very few rule schemata are used.

Controlled Linguistic Deduction [Uszkoreit, 1991] adds control information to declarative grammars in typed feature formalisms, and allows the mixing of depth-first and breadth-first search by assigning preferences to disjuncts. The effect is to derive a set of preferred readings first, and to cut off unlikely paths in the search space. The preferences are based on the success potential of a disjunct, i.e., "the disjuncts that have the highest probability of success are processed first." However, the approach suffers from the problem that it uses only local optimisation for the processing of disjuncts, but often unifications that are successful locally are not successful in later structure building. For lexical ambiguity, preferences are assigned dynamically to disjuncts by means of a spreading activation net, based on "a combination of factors such as the topic of the text or discourse, previous occurrence of priming words, register, style, and many more."

Other processing-based approaches are parsers based on simulated annealing [Kempen and Vosse, 1989], connectionistic models [Schnelle, 1990; Cottrell, 1987], and psycholinguistic models of human sentence processing [Hawkins, 1990].

The use of specialised processing strategies for disambiguation is appealing because the search space is reduced and the efficiency of the system increased.

The drawbacks of ordering readings by means of a processing strategy are the following:

- The criteria for ordering are hidden in the processing model and not specified in a declarative way. A change of the processing model may change the ordering of the analyses.

- Additional knowledge (like word order constraints or selectional restrictions) is not exploited for ordering.

Procedural models make very good sense as explicit psycholinguistic models of human sentence processing, which can account for *emergent behaviour* from the interaction of simple processing principles and also as a target for the compilation of declarative grammars into performance grammars.

4.1.1.3 Other Declarative Approaches

Barnett and Mani propose a model of bidirectional preferences, which can be used for both parsing and generation [Barnett and Mani, 1991; Barnett, 1994]. Their model is simpler than the other models discussed since they do not make use of numerical values. In their approach, preference is just on order relation between readings. While the simplicity of this model is appealing, it cannot handle

cases where the preference of a phrase depends on different factors since it has no mechanism for combining different preferences.

In the PLNLP system, dynamic relaxation techniques are used to handle ill-formed input in a multiple-pass parsing model [Jensen *et al.*, 1992; Chanod *et al.*, 1994]. This model contains conditions which can be relaxed with a certain cost. During the first parsing pass, all constraints are enforced. If the first-pass analysis fails, then further analyses are performed in which constraints can be relaxed dynamically.

The opposite approach is pursued by Manaster-Ramer, who — following Chomsky — proposes a model called *transductive linguistics,* in which every sentence, whether ill-formed or well-formed, is given a structural description [Manaster-Ramer, 1992]. In the case of well-formed sentences, further constraints can be applied to the structural description.

Menzel takes a similar approach, in which parsing is understood as a disambiguation procedure [Menzel, 1995]. In his model, syntax and semantics are regarded as autonomous modules which can independently produce analyses of strings at different layers of representation. If the analysis at one level fails, the other level can still produce an analysis. Constraints are associated with numerical penalty factors (with a range of values between 0 and 1), which are combined by multiplication. If the penalty factor for constraint violation is 0, then it is a strict constraint in the classical sense, and if it is a value greater than 0, then it is a soft constraint which can be violated. Parsing proceeds by proposing a set of analyses, and applying constraints to prune this set to find the least disfavoured analysis.

A related approach is followed by Huckle, who uses a greatly impoverished phrase structure component (finite automata) which is combined with a very complex probabilistic model [Huckle, 1995]. All the constraints of the grammar that go beyond regular languages are handled by the statistical model. It is, however, not obvious that reliable statistical data can be extracted from corpora to model all the complex constraints of the grammar, as would be required in Huckle's model.

Kim [Kim, 1994] proposes a variant of unification called *graded unification* which allows unification failure with a certain penalty score.

Kim's method for combining scores is taking the average of two scores. However, averaging leads to the loss of commutativity, as the following examples show. In both examples, the feature structure X_1 has the score 0.8, X_2 has the score 0.4, and X_3 has the score 0.2. In (4.1), X_1 and X_2 are unified first, and then the result is unified with X_3 with, and in (4.2), X_2 and X_3 are unified first, and then the result is unified with X_1, and in (4.3), X_1 and X_3 are unified first. The resulting scores are different in each case.

$$(X_1 \wedge X_2) \wedge X_3 \text{ has the score } \frac{\left(\frac{(0.8+0.4)}{2}\right) + 0.2}{2} = 0.4 \tag{4.1}$$

$$X_1 \wedge (X_2 \wedge X_3) \text{ has the score } \frac{0.8 + \left(\frac{(0.4+0.2)}{2}\right)}{2} = 0.55 \tag{4.2}$$

$$(X_1 \wedge X_3) \wedge X_2 \text{ has the score } \frac{\left(\frac{(0.8+0.2)}{2}\right) + 0.4}{2} = 0.45 \tag{4.3}$$

Given that Kim's paper addresses the question of psycholinguistic processing, it might be argued that commutativity in the calculation of preferences does not hold in incremental sentence processing because the elements that are processed first have a larger influence. However, Kim does not attempt to give any evidence which could support such a view.

Abductive models of linguistic interpretation associate a cost with every fact that must be assumed in order to explain a reading [Hobbs *et al.*, 1993; Den, 1994].

There are no approaches that explicitly make use of fuzzy logic. This makes sense insofar as the properties assigned to NL structures are not gradable quantities, e.g. being a sign or satisfying the subcat principle can be true with a certain probability, but it is hard to conceive of a structure being a sign to a certain degree.

The question of preferences, probabilities, grammaticality etc. is a notoriously difficult one that will require much more research and touches the foundations of linguistic theory. At present, probabilistic models operate at a very different level (e.g. bi/trigrams) than full-fledged grammatical theories. The formalism and processing model presented here do not attempt to answer these questions, but hope to provide a formal framework in which such models can be implemented and evaluated.

4.1.2 A Model of Preference

In order to model the choice between different linguistic deduction results (i.e., different readings for parsing and different paraphrases in generation), we impose an ordering relation on the different LD results. The relation expresses which results are preferred over each other. The relation is transitive, but not required to be antisymmetric, so that two LD results can be equally preferred.

The ordering relation is based on numerical preference values, which can be interpreted as probabilities. Each clause of a program is associated with a preference value. In case of unit clauses, this is a probability value, a real number between 0 and 1. In case of non-unit clauses, some goals in the antecedent are associated with a preference value, while others have no preference value because they express necessary conditions for the clause to be true. The consequent is associated with a formula which denotes a function for computing the preference value of the whole clause from the preference values of those goals in the antecedent which have a preference value.

The function for computing the preference value of a structure from the preference values of its parts is required to be monotonic. If the preference value of one part is increased, and the preference values of all other parts remain unchanged, then the preference value of the whole structure must also increase. This monotonicity requirement is crucial for the best-first search algorithm presented in section 4.1.3 because it makes sure that items with low preference value can be ignored or delayed because they cannot increase the preference value of the constituents which they are a part of.

The monotonicity requirement might seem to suggest a kind of "context-freeness" of the combination of items in the sense that the combination of two constituents with high preference value would also receive a higher preference value than the combination of two constituents with lower preference value. However, this is not necessarily the case since the principles of the grammar can express conditions on the relation between two constituents that are combined. Since the evaluation of the principles also results in a preference value which enters into the preference value of the whole, it is possible that violation of a relational constraint expressed by a principle leads to the degradation of the preference value of the whole structure.

We propose a definite clause language augmented with preference values. A clause of a logic program has the form in (4.4), where C is the consequent of the clause, and the A_i are the goals in the antecedent.

$$C \leftarrow A_1 \wedge \ldots \wedge A_n \tag{4.4}$$

In the following, we give two schemes for calculating the preference value of a clause. $\mathcal{P}(x)$ stands for the probability of the consequent or one of the goals in the antecedent of the clause, and $\mathcal{P}(Clause)$ stands for the probability of the entire clause. The probabilities of different clauses that can be chosen as alternatives must add up to 1. The question which clauses can be chosen as alternatives to each other depends on the instantiation of the 'input' arguments of these clauses. For example, if the instantiation of an input argument leads to the deterministic choice of a clause, then the probability of choosing that clause, given the input argument, is always 1. Since the same holds for the other clauses, for different input arguments, the sum of the probabilities for the defining clauses of a predicate can be larger than one. In the special case where all clauses that make up a predicate definition can be chosen, their probabilities must add up to 1.

In (4.5), the scheme of for the calculation of probabilities is the same as in Eisele's approach: all the goals are treated as having independent probabilities, whose product multiplied with the probability of the clause is the probability of the consequent of the clause.

$$\mathcal{P}(C) = \mathcal{P}(Clause) * \mathcal{P}(A_1) * \ldots * \mathcal{P}(A_n) \tag{4.5}$$

In (4.6), the probability of the consequent is the product of the probability of the clause and the weighted sum of the probabilities of the goals in the antecedent. In this case, the goals are not treated as independent, but rather the satisfaction of each goal constitutes additional evidence for the truth of the consequent. All weights W_i in the formula add up to 1 ($W_1 + \ldots + W_n = 1$), and the preference values (probabilities) of all goals in the antecedent of a clause can have a value between 0 and 1.

$$\mathcal{P}(C) = \mathcal{P}(Clause) * [W_1 * \mathcal{P}(A_1) + \ldots + W_n * \mathcal{P}(A_n)] \tag{4.6}$$

In practice, it may be necessary to mix the two kinds of approaches and treat some goals as independent, while others just supply supporting evidence. We will see an example below for the case of HPSG where the probabilities of the daughters are treated as independent and multiplied, whereas the principles are seen as providing supporting evidence, and their weighted sum enters in the probability calculation.

In the concrete notation, we combine the consequent C and its preference formula with the operator `#` and likewise the goals in the antecedent and with their preference variables, so that the clause and the probability assignment in (4.6) are combined in the following notation:

$$\begin{array}{l} C\#(P_{Clause} * (W_1 * P_1 + \ldots + W_n * P_n)) \leftarrow \\ \qquad A_1\#P_1 \wedge \\ \qquad \vdots \\ \qquad A_n\#P_n. \end{array} \tag{4.7}$$

As a simple illustration, figure 4.1 shows a preference-based definition of the member relation that gives preference to the first members of the list. The solutions of a query and their preference values are also shown.

(4.8) shows the definition of a sign in HPSG as a clause in a logic program (cf. section 2.3). The clause given is the one for head-complement structures.

$$\begin{array}{l} \text{sign}\left(X\& \begin{bmatrix} \text{head_dtr:} & \text{HD} \\ \text{comp_dtr:} & \text{CD} \end{bmatrix} \right) \leftarrow \\ \qquad \text{sign(HD)} \wedge \\ \qquad \text{sign(CD)} \wedge \\ \qquad \text{principles(X)}. \end{array} \tag{4.8}$$

The probabilities for the clause are calculated according to schema (4.5), as indicated below in semi-formal notation.

```
member(X,⟨X|R⟩)#0.5.

member(X,⟨H|R⟩)#0.5*P ←
        member(X,R)#P.
```

```
?- member(X,⟨a,b,c⟩)#Pref.
```

$X = a \quad Pref = 0.5$
$X = b \quad Pref = 0.5 * 0.5 = 0.25$
$X = c \quad Pref = 0.5 * 0.5 * 0.5 = 0.125$

Figure 4.1: A definition of the member relation with preference values

$$\mathcal{P}(sign(X)) = \mathcal{P}(sign(HD)) * \mathcal{P}(sign(CD)) * \mathcal{P}(principles) * \mathcal{P}(Clause) \quad (4.9)$$

$$\begin{array}{l} \text{sign}\left(X \& \begin{bmatrix} \text{head_dtr: HD} \\ \text{comp_dtr: CD} \end{bmatrix} \right) \#P_{HD} * P_{CD} * P_{principles} * P_{clause} \leftarrow \\ \quad sign(HD)\#P_{HD} \wedge \\ \quad sign(CD)\#P_{CD} \wedge \\ \quad principles(X)\#P_{principles}. \end{array} \quad (4.10)$$

For the calculation of the preference value of the principles, we use schema (4.6), the weighted sum of preferences, since the principles cannot really be considered as independent. The probability of the clause does not appear in this example since there is only one clause defining the principles, so that the probability of this clause is 1.

$$\text{principles(X)} \leftarrow \text{principle}_1\text{(X)} \wedge \ldots \wedge \text{principle}_n\text{(X)}. \quad (4.11)$$

$$\mathcal{P}(principles) = W_1 * \mathcal{P}(principle_1) + \ldots + W_n * \mathcal{P}(principle_n) \quad (4.12)$$

$$\begin{array}{l} \text{principles(X)}\#(W_1 * P_1 + \ldots + W_n * P_n) \leftarrow \\ \quad \text{principle}_1\text{(X)}\#\text{P}_1 \wedge \ldots \wedge \text{principle}_n\text{(X)}\#\text{P}_n. \end{array} \quad (4.13)$$

It must be noted that this scheme for calculating preferences is an educated guess at the appropriate method that appears to fulfill the requirements given above. Clearly, the exact statistical dependencies between the choices of different clauses for a principle and the resulting influence on the probability of a reading or paraphrase are too hard to work out exactly at the present state of our knowledge, so that we propose to approximate this relationship by the use of weights which correspond to the different strength of the influence of each principle.

In particular, it appears appropriate that

- the probabilities of the constituents are treated as independent and therefore multiplied, as any relationships between the signs are stated in the principles,
- the use of addition for the combination of weighted evidence can be used to limit the influence of really peripheral (e.g., stylistic) principles, and
- the probability of the clause enters as a multiplicand in the formula, as this ensures that more frequent structures are indeed preferred.

4.1.3 Preferences and best-first processing

In this section, we will discuss how preference values can be used to control the agenda-based best-first algorithm presented in chapter 3. There are a number of methods for assigning a priority. A good strategy might try to compensate for the effect that the preference value is generally lower with more complex derivations. For example, in the case of parsing, an items that covers more of the input string than another item will, ceteris paribus, have a lower preference value than an item which covers less of the input string. Since the longer item will need fewer combinations with other items to cover the entire input string (i.e., a proof of the goal), it appears reasonable to give it a higher priority than a shorter item with the same preference value. Another strategy could make use of the indices of the items to ensure that combination of the items at the top of the agenda with the items already present in the chart can completely cover the entire input. Since it can be very complex to tune such strategies, we will stick to a simpler model in this section: we will equate the preference value of an item with its priority on the agenda. This model has the nice property that it guarantees that the derivation with the highest preference value is found first. This best-first search algorithm implements the $A*$ search algorithm [Nilsson, 1980].

The same model has been used for speech parsing by Thompson, who applies chart parsing techniques to find an optimal path through a lattice of word hypotheses with probabilities for the word hypotheses given the acoustic evidence, and tag transition probabilities [Thompson, 1990].

In the case of passive items, the equation of preference values with priorities poses no problem. A passive item (as a completely proved goal) always has a definite preference value. In the case of active items, on the other hand, the preference

formula associated with its consequent may contain uninstantiated variables for the preference values of the goals that are still needed for the proof of the item. Since these variables can only take on a value between 1 and 0, the preference value of an active item will be in an interval whose lower bound is determined if all free preference variables in a formula are set to 0, and whose upper bound is determined by setting all free variables to 1. The formula for calculating the preference interval of a formula is given in rule (4.14) and makes use of the formulae for the lower bound in (4.15) and for the upper bound in (4.16).

$$\text{pref-interval}(F) = [\text{lower-bound}(F), \text{upper-bound}(F)] \tag{4.14}$$

$$\begin{array}{ll} \text{lower-bound(x)} = 0 & \text{where x is a free variable} \\ \text{lower-bound(n)} = \text{n} & \text{where n is a real number} \\ \multicolumn{2}{l}{\text{lower-bound(x + y)} = \text{lower-bound(x)} + \text{lower-bound(y)}} \\ \multicolumn{2}{l}{\text{lower-bound(x} * \text{y)} = \text{lower-bound(x)} * \text{lower-bound(y)}} \end{array} \tag{4.15}$$

$$\begin{array}{ll} \text{upper-bound(x)} = 1 & \text{where x is a free variable} \\ \text{upper-bound(n)} = \text{n} & \text{where n is a real number} \\ \multicolumn{2}{l}{\text{upper-bound(x + y)} = \text{upper-bound(x)} + \text{upper-bound(y)}} \\ \multicolumn{2}{l}{\text{upper-bound(x} * \text{y)} = \text{upper-bound(x)} * \text{upper-bound(y)}} \end{array} \tag{4.16}$$

We assign the upper bound of its preference formula as the priority of an active item. Under this assumption, we can state the following proposition that holds if preferences values of goals are multiplied with each other, i.e., if they are treated as independent probability values as in Eisele's model.

> If the priority of each passive item is its preference value and the priority of each active item is the upper bound of its preference value, then a best-first algorithm will enumerate all solutions in order of decreasing preference (i.e., it will find the optimal solution first).

The proof of this proposition proceeds by contradiction: Assume that a non-optimal solution S with preference value P has been found, but there is still an item I with preference value $P1$ on the agenda whose combination with the chart can produce a better solution. Since $P1$ is after P on the agenda, $P1 \leq P$. Since preference values are multiplied, a new item which is constructed by making use of item I must have a preference value lower then $P1$. Hence, no item can be

constructed which has a preference value higher than $P1$, and by transitivity of the $\leq$ relation no item which has a higher preference value than P. Therefore, the solution S with preference value P is optimal, and the assumption is false.

Unfortunately, this nice property does not hold in the case where preferences are not treated as independent probabilities. In the special case of a formula (4.17), however, it can be assumed to hold, since the only things that are stored as items (the daughters, but not the individual instances of principles), are in fact combined by multiplication.

$$\mathcal{P}(X) = \mathcal{P}(HD) * \mathcal{P}(CD) * \mathcal{P}(principles) * \mathcal{P}(clause) \tag{4.17}$$

However, it must be noted that such a strategy which guarantees to find optimal solutions can lead to a fairly complete exploration of the search space because all items which still have the chance of contributing to a better solution are combined before the best solution found so far is further processed. In order to avoid this large search space one may give higher priority to longer strings or more complex derivations (by normalising by the string length or number of nodes), or consider beam search as an alternative which cuts off some parts of the search space with low preference values, but may occasionally miss an optimal solution.

4.1.4 Word Order

The original motivation for the proposal made in this section stems from our work on the encoding and processing of linear precedence (LP) constraints for German word order. LP constraints are not absolute constraints, but their violation only makes a sentence less acceptable. The same is true of other kinds of linguistic information, e.g., selectional restrictions, collocational information, or attachment preferences. On the one hand, information about word order regularities etc. can easily be expressed in typed feature formalisms (see [Engelkamp *et al.*, 1992]), and used for disambiguation. On the other hand, if these kinds of knowledge are added as additional constraints to natural language grammars, some unambiguous sentences will also be excluded. Sentence (5) illustrates this problem with a simple example from German word order.

(5) Die Mutter (nom/acc) küßte die Tocher (nom/acc)
The mother kissed the daughter
The mother kissed the daughter / The daughter kissed the mother

In English, subject and object are distinguished by their position, whereas in German they are distinguished by case marking. The subject is in the nominative case, and the object in the accusative. Sentence (5) is ambiguous because the NPs *die Mutter* and *die Tochter* can morphologically both be in the nominative and in the accusative case. So, both NPs can be either the subject or the object of the sentence.

However, speakers of German are in general not aware of the ambiguity of example sentence (5) because there is a strong tendency for the subject to precede the object in linear order. Example sentence (5) can be disambiguated by adding an LP constraint (*nom* < *acc*) to the grammar. Unfortunately, a grammar which contains this LP constraint would also exclude the perfectly grammatical German sentence (6).

(6) Den Knaben schlägt der Lehrer
the boy (acc) hits the teacher (nom)
The teacher hits the boy

The problem is that the grammar becomes too restrictive when the knowledge needed for disambiguation is added.

The adequate treatment of word order is still an open problem in linguistics. For the purposes of this section, we consider two approaches: first, a model which assumes an unmarked (default) argument order, and gives deviations from this order a lower preference value, and secondly a model which assumes several competing ordering principles.

For the first model, the unmarked word order is the order of the elements of the subcategorisation (SUBCAT) list[5], as in the simplified lexical entry for the ditransitive verb *gibt* (give).

$$\text{lexicon}\left(\text{gibt}, \begin{bmatrix}\text{synsem:loc:} & \begin{bmatrix}\text{cat:head:v} \\ \text{subcat:}\langle\,\text{np(acc), np(dat), np(nom)}\,\rangle\end{bmatrix}\end{bmatrix}\right). \tag{4.18}$$

Elements of the subcat list are discharged by the Subcat principle. We assume that only one element of the Subcat list is taken at a time, so that binary branching trees result.

$$\text{subcat_principle}\left(\begin{bmatrix}\text{synsem:loc:cat:subcat:Subcat} \\ \text{dtrs} \begin{bmatrix}\text{head_dtr:synsem:loc:cat:subcat:HSC} \\ \text{comp_dtr:synsem:CD_Synsem}\end{bmatrix}\end{bmatrix}\right) \leftarrow \\ \text{insert(Subcat,CD_Synsem,HSC).} \tag{4.19}$$

[5]We use Prolog notation for lists. The elements of the SUBCAT list are given in reverse surface order to facilitate processing in head-final structures, where the head takes arguments from right to left.

The third argument of the predicate `insert/3` is a list, in which the second argument is inserted at any place into the first argument, which is a list.

$$\begin{array}{l} \text{insert(L,X,}\langle \text{X} \mid \text{L} \rangle\text{).} \\ \text{insert(}\langle \text{H} \mid \text{L1} \rangle\text{,X,}\langle \text{H} \mid \text{L} \rangle\text{)} \leftarrow \\ \qquad \text{insert(L1,X,L).} \end{array} \tag{4.20}$$

These definitions allow any word order that is a permutation of the subcat list of the head. For this example, we make the simple assumption that the "best" word order is the unmarked word order, and that any deviation from the unmarked order results in a decrease in preference.

We write pairs of feature terms (FT) and preference values (PV) as `FT#PV`. Each consequent of a clause is followed by a formula which specifies how the preference for that clause is calculated. The definitions of `insert/3`, `subcat_principle` and `phrasal_sign` annotated with formulae for preference calculation are shown in figure 4.2. The preference value of a phrasal sign is defined by the multiplication of the preference values of the daughters, and the preference value that results from application of the subcat principle for that sign. The subcat principle can unify any element of the subcat list with the complement daughter, by means of the relation `insert/3`. The subcat principle gets its preference value from `insert/3`. The idea is that the preference value is highest if the first element is taken from the SUBCAT list of the head daughter. The further away the element is from the beginning of the list, the lower the preference.

The preference calculation function in the definition of `phrasal_sign` is responsible for percolating preference values in phrase structures, which results in the preference assignments shown in figure 4.3 for the constituents of permutations of sentence (7), which is given in the unmarked word order.

(7) (weil) der Mann dem Mädchen das Buch gibt
(because) the man (nom) the girl (dat) the book (acc) gives

Under the view presented here, word order constraints are not really constraints that are violated, but rather preferences for choosing one alternative clause in the definition of `insert/3`[6].

Equivalently, the above treatment of unmarked word order can be handled in the lexical entry of the verb. In this case we assume that the SUBCAT principle always takes the first element of the SUBCAT list. The lexical entry for the verb differs from the one above in that its SUBCAT value is the permutational closure of the SUBCAT list.

[6] There may be other uses of `insert/3`, in which there is no preferred order. For these cases, an alternative definition of `insert/3` must be used, in which all solutions have the same preference value.

insert(L,X,⟨X|L⟩)#0.5.
insert(⟨H|L1⟩,X,⟨H|L⟩)#0.5*Pref ←
    insert(L1,X,L)#Pref.

$$\text{subcat_principle}\left(\begin{bmatrix}\text{synsem:loc:cat:subcat: Subcat} \\ \\ \text{dtrs: } \begin{bmatrix}\text{head_dtr:synsem:loc:cat:subcat: HD_SC} \\ \text{comp_dtr:synsem: CD_Synsem}\end{bmatrix}\end{bmatrix}\right)\#\text{Pref} \leftarrow$$

    insert(Subcat,CD_Synsem,HD_SC)#Pref.

$$\text{phrasal_sign}\left(X\& \begin{bmatrix}\text{hd: HD} \\ \text{cd: CD}\end{bmatrix}\right)\#P1 * P2 * P3 \leftarrow$$

    sign(HD)#P1,
    sign(CD)#P2,
    subcat_principle(X)#P3.

Figure 4.2: Principles with preferences

String	Pref.	Comment
gibt	1	
das Buch gibt	1	first element of subcat list
dem Mädchen das Buch gibt	0.5	first element of remaining SL
der Mann dem Mädchen das Buch gibt	0.5	
der Mann das Buch gibt	0.25	2nd element of remaining SL
dem Mädchen der Mann das Buch gibt	0.25	
dem Mädchen gibt	0.25	2nd element of subcat list
das Buch dem Mädchen gibt	0.25	first element of remaining SL
der Mann das Buch dem Mädchen gibt	0.25	
der Mann dem Mädchen gibt	0.125	2nd element of remaining SL
das Buch der Mann dem Mädchen gibt	0.125	
der Mann gibt	0.125	3rd element of subcat list
dem Mädchen der Mann gibt	0.125	first element of remaining SL
das Buch dem Mädchen der Mann gibt	0.125	
das Buch der Mann gibt	0.0625	2nd element of remaining SL
dem Mädchen das Buch der Mann gibt	0.0625	

Figure 4.3: Preference values for the permutations of a sentence

$$\text{lexicon}\left(\text{gibt}, \left[\text{synsem:loc:} \begin{bmatrix}\text{cat:head:v} \\ \text{subcat:SUBCAT}\end{bmatrix}\right]\right)\#Pref \leftarrow$$
$$\text{permute}(\langle \text{np(acc),np(dat),np(nom)} \rangle\text{,SUBCAT)\#Pref.} \qquad (4.21)$$

The closer the permuted list is to the original list, the better the preference value that the relation `permute/2` assigns.

```
permute(⟨ ⟩,⟨ ⟩)#1.0.
permute(⟨H|T⟩,Perm)#Pref0*Pref1 ←
        permute(T,Perm1)#Pref0,
        insert(Perm1,H,Perm)#Pref1.
```
(4.22)

As a result, we get six lexical entries for the word *gibt* with all permutations of the standard order of the subcat list. In figure 4.4, the permuted subcat lists and

their preference values are given.

⟨np(dat),np(acc),np(nom)⟩	0.25
⟨np(acc),np(nom),np(dat)⟩	0.25
⟨np(nom),np(acc),np(dat)⟩	0.125
⟨np(dat),np(nom),np(acc)⟩	0.125
⟨np(nom),np(dat),np(acc)⟩	0.0625

Figure 4.4: Permuted SUBCAT lists and their preference values

This model of word order preferences as deviation from a standard order is too simple to account for real word order data because other factors also play a role. We will take a look at the work on Linear Precedence constraints for German [Uszkoreit, 1987b]. Uszkoreit [p. 24] lists the following most relevant principles that govern the order of complements and adjuncts in German:[7]

1. Focus follows nonfocus.

2. The unmarked order is SUBJ, IOBJ, DOBJ.

3. Personal pronouns precede other NPs.

4. Definite NPs precede nondefinite NPs.

5. Light constituents precede heavy constituents.

6. If a focussed constituent precedes a nonfocussed it will carry the focus accent.

However, not all of these principles must always be observed, and it is often impossible to observe all these principles simultaneously, for example in a sentence with a pronominal object NP and a non-pronominal subject NP, one principle will always be violated, as Uszkoreit illustrated with the sentences (8) and (9).

(8) Dann wird der Doktor ihn sehen
Then will the doctor him see
Then the doctor will see him

(9) Dann wird ihn der Doktor sehen
Then will him the doctor see
Then the doctor will see him

[7] These principles were first proposed in [Lenerz, 1977], and have been tested and confirmed in recent psycholinguistic experiments [Pechmann *et al.*, 1994].

In sentence (8), principle 3 is violated since a pronoun follows a full NP, and the order in sentence (9) violates principle 2 since it deviates from the standard order. Uszkoreit proposes the following solution for the problem of ordering conflicts. The set of ordering principles above is divided into those that directly influence the grammaticality of a sentence, and those that are stylistic factors that influence the acceptability, but do not make a sentence ungrammatical. We shall call the former set *ordering principles* and the latter set *ordering preferences*. According to Uszkoreit, this division is a separation of syntactic and stylistic rules. The set of ordering principles are the following:

1. Focus follows nonfocus.
2. The unmarked order is SUBJ, IOBJ, DOBJ.
3. Personal pronouns precede other NPs.

Only violation of ordering principles can make a sentence entirely ungrammatical. However, for a sentence to become ungrammatical, all ordering principles must be violated. In [Uszkoreit, 1987b], this is formalised by taking these word order constraints as a disjunction. At least one of the disjuncts must be true, not just in the sense that the order specified by the disjunct is not violated by a pair of constituents, but in the stronger sense that the reverse ordering would lead to a violation of the constraint.

Moreover, "the degree of markedness increases with the number or total weight of violated principles" [Uszkoreit, 1987b, p. 123]. We will use our notion of preferences to formalise this observation (where a lower preference value corresponds to an increased degree of markedness).

The constraints from the set of ordering preferences (relating to definiteness, heaviness and focus accent) also influence the relative acceptability, but can never make a sentence completely ungrammatical.

We will now show how to formalise this theory with our notion of preferences. In contrast to the GPSG theory in [Uszkoreit, 1987b], we can not only formalise the fact that violation of a combination of ordering constraints makes a sentence ungrammatical, but also the varying degrees of acceptability.

The definition of a sign in figure 4.5 makes use of the notion of an ordering domain (a sequence of constituents), and uses the procedures `ordering-principles/1` and `ordering-preferences/1` to check the ordering principles and the ordering preferences for this domain. The resulting probabilities are multiplied, based on the assumption of independence between the principles and the preferences. The procedure `ordering-principles` is defined by a double recursion over the word order domain in order to check the three ordering principles for every pair of elements in the domain.

The procedures `check-focus` and `check-pronominal` (figure 4.6) define the checking of the ordering principles relating to focus and pronominality. In this

```
sign(X)#P1 * P2 * ... * Pwo1 * Pwo2 ←
        sign(Dtr1)#P1 ∧
        sign(Dtr2)#P2 ∧
        ...
        ordering-domain(X,OD) ∧
        ordering-principles(OD)#Pwo1 ∧
        ordering-preferences(OD)#Pwo2.

ordering-principles(⟨F|R⟩)#P1 * P2 ←
        ordering-principles2(F,R)#P1 ∧
        ordering-principles(X)#P2.

ordering-principles(⟨ ⟩)#1.

ordering-principles2(X,⟨F|R⟩)#(W1 * P1 + W2 * P2 + W3 * P3) * P4 ←
        check-unmarked-order(X,F)#P1 ∧
        check-focus(X,F)#P2 ∧
        check-pronominal(X,F)#P3 ∧
        ordering-principles2(X,R)#P4.

ordering-principles2(X,⟨ ⟩)#1.
```

Figure 4.5: Ordering principles

```
check-focus(+focus,−focus)#0.
check-focus(−focus,−focus)#1.
check-focus(+focus,+focus)#1.
check-focus(−focus,+focus)#1.

check-pronominal(+pro,−pro)#1.
check-pronominal(−pro,−pro)#1.
check-pronominal(+pro,+pro)#1.
check-pronominal(−pro,+pro)#0.

check-unmarked-order(dat,nom)#0.
check-unmarked-order(acc,nom)#0.
check-unmarked-order(acc,dat)#0.
check-unmarked-order(acc,acc)#1.
check-unmarked-order(nom,acc)#1.
check-unmarked-order(nom,dat)#1.
check-unmarked-order(dat,acc)#1.
check-unmarked-order(X,¬ case)#1.
check-unmarked-order(¬ case,X)#1.
```

Figure 4.6: Procedures for checking the ordering principles

case, the preference values 1 and 0 are simply interpreted as truth and falsehood, respectively. Since both variables in the goal are expected to be instantiated (input variables), the preference values assigned to the clauses can add up to more than 1. It is assumed that the features FOCUS and PRO are appropriate for all complements and adjuncts. The notations `+focus, -focus, +pro, -pro` are abbreviations for the corresponding feature structures.

The procedure `check-unmarked-order` assigns a preference value of 0 if the unmarked order is violated, and a preference value of 1 otherwise. There are two clauses for the case where one of the elements of the ordering domain is not specified for case.[8]

If all word ordering principles are violated for a pair of constituents in an ordering domain, the preference value for each principle is 0, and hence the weighted sum of the preference values is also 0. Since this value enters into the preference calculation of the whole sentence as a factor of a product, the preference value for

[8]The notation `nom, dat, acc` are abbreviations for feature structures in which the case value of specified as nominative, dative, or accusative, respectively. The notation `¬case` is an abbreviation for the sorts for which the feature CASE is not appropriate. There are no clauses for the pairs ⟨nom,nom⟩ and ⟨dat,dat⟩ since these will not occur together in an ordering domain.

the sentence will also be 0, i.e., the sentence is ungrammatical.

If one of the ordering principles is satisfied and has preference value 1, the weighted sum of the preferences is greater than 0, and therefore the ordering principles do not lead to the ungrammaticality of the structure. This has exactly the same effect as the formulation of the ordering principles as a disjunction in [Uszkoreit, 1987b], but in addition, it can also account for the differences in acceptability among the sentences which satisfy some of the ordering principles.

The exact determination of the weights will have to make use of corpus-based statistics, and of the results of the psycholinguistic experiments on word order preferences described in [Pechmann *et al.*, 1994].

In the case where ordering preferences are violated, the preference value is always greater than 0, so that these principles will never predict the ungrammaticality of a sentence, but only model its loss of acceptability. This is shown in figure 4.7, where the procedure for checking ordering preferences is given, with preference values for the ordering preference “definites precede indefinites”.[9] Like the procedure `ordering-principles`, the procedure `ordering-preferences` is defined by a double recursion over the list representing the word order domain (figure 4.7).

We will not discuss optimal processing strategies for checking the ordering constraints here, but it is clear that they should be based on guarded constraints to avoid the instantiation of features through the checking of ordering constraints, which would lead to an increased amount of search. There must also be a numerical constraint that the preference value of a phrase be greater than 0, to ensure the instantiation of the feature FOCUS with the values + or − in cases where this is the only possibility to make a sentence grammatical.

In a more recent model word order constraints are modelled by finite state transducers, which return a preference value for each order [Erbach and Uszkoreit, 1995]. The output of such a transducer can also enter as a factor in the preference calculation of a sign.

4.1.5 Application to Disambiguation

A typical natural language *understanding* (NLU) system will have to decide which of several analyses of a given string is the appropriate one, i.e. the one intended by the (human or artificial) system that generated the string. The question is which parts of an NLU system are responsible for this step.[10]

[9] `+def`, `-def`, `¬def` are the usual abbreviatory conventions. The preference value 0.5 for the case where the principle is violated is randomly chosen, and has not been determined by a statistical analysis. Since 1 is interpreted as truth and 0 as falsity, 0.5 can be interpreted as the probability the the clause being true given the input arguments.

[10] In the construction of the LILOG system, there was at one time a situation where the group responsible for syntactic analysis delivered all possible readings of an input string and counted on the knowledge processing group to take care of the disambiguation, whereas the knowledge

ordering-preferences(⟨F|R⟩)#$P_1 * P_2 \leftarrow$
    ordering-preferences2(F,R)#P_1 ∧
    ordering-preferences(X)#P_2.

ordering-preferences(⟨ ⟩)#1.

ordering-preferences2(X,⟨F|R⟩)#$(W_1 * P_1 + W_2 * P_2 + W_3 * P_3) * P4 \leftarrow$
    check-heaviness(X,F)#P_1 ∧
    check-definiteness(X,F)#P_2 ∧
    check-focusaccent(X,F)#P_3 ∧
    ordering-preferences2(X,R)#P_4.

ordering-preferences2(X,⟨ ⟩)#1.

check-definiteness(+def,−def)#1.
check-definiteness(−def,−def)#1.
check-definiteness(+def,+def)#1.
check-definiteness(X,¬ def)#1.
check-definiteness(¬ def,X)#1.
check-definiteness(−def,+def)#0.5.

Figure 4.7: Procedures for checking ordering preferences

We can roughly characterise the architecture of an NLU system as consisting of three major parts.

1. Syntactic (and morphological) analysis
2. Contextual interpretation (anaphora resolution etc.)
3. Knowledge representation and inference

We claim that the task of disambiguation cannot be solved by any of these components alone, but all of these three parts can play a role in disambiguation, as sentences 10 to 17 illustrate.

The German example sentence (10) has four readings, since *Schauspielerin* can fill the roles of subject and direct object, and *Moritz* and *Lisa* can fill the roles of subject, indirect object and direct object. All four readings are equally plausible, but word order preferences provide strong evidence for the reading in which *Moritz* is the subject, *Lisa* the indirect (dative) object and *Schauspielerin* the direct (accusative) object. This is a case where syntactic preferences determine the choice of a reading, unless there is evidence to the contrary.

(10) Moritz zeigt Lisa die Schauspielerin
Moritz shows Lisa the actress

Sentence (11) shows a lexical ambiguity of the word *mouse*, which is ambiguous between rodent and computer input device. In the first sentence, both readings are equally possible. Coindexation of the pronoun *it* in the second sentence with *mouse*, together with selectional restrictions of the predicative adjective *dead*, lead to a preference for the rodent reading. The same effect occurs in sentence (12), and the device reading is preferred in sentence (13). In this case, anaphora resolution and selectional restrictions determine the choice of a reading.

(11) On the table there was a mouse. It was dead.

(12) Bill found a mouse. The animal was half-starved.

(13) Bill found a mouse. The device was in need of repair.

The following sentences can only be disambiguated by inference and world knowledge. In sentence (14), the locative adjunct *in the river* suggests the auxiliary verb reading of *can*, since rivers are locations where fishing is possible, and in sentence (15), the main verb reading of *can* is suggested by the locative adjunct since factories are locations where goods are canned.

(14) We can fish in this river.

---

processing group assumed that they would get only one reading because disambiguation i takes place during syntactic processing.

(15) We can fish in this factory.

In the sentences (16) and (17) knowledge processing is required to resolve the lexical ambiguity of *window* between an opening in a wall and a human-computer interface widget. The process of fitting the definite descriptions *the pane* and *the title bar* into the discourse representation will assist in the disambiguation of *window* if knowledge is available that a pane is part of a framed window in the wall, and that a title bar is part of a computer window.

(16) John stared at the window. The pane was broken.

(17) John stared at the window. The title bar showed weird characters.

By viewing syntactic analysis as a deductive process, it can be integrated with the other two processes. The same preference scheme, and even the same deductive techniques can be applied to all three stages, so that the final preference ordering among different readings is made up from results of all three processes.

4.1.6 Application to Generation

In an ideal situation, one can assume that the input to a generator is always fully specified so that only one unique result is generated that is optimally suited to a achieve the desired communicative effect in a given communicative situation. However, in realistic NLP applications, there is often not enough information available and decisions must be made in the face of this incomplete knowledge. Such situations arise for example in systems that must deal deal with a wide variety of input, such as the generation component of large-coverage machine translation systems, which normally do not make use of a discourse model. In order to arrive at a reasonably acceptable output of the generator when no information is available to make an informed decision for a non-deterministic choice, one can use statistical information and simply take the most probable choice. For example in the case of lexical selection or the selection of syntactic variants, the most frequent one (either in terms of absolute probabilities or of conditional probabilities such as n-gram models) is chosen. Whenever the information for making a choice is available, it is used, and preferences (probabilities) are only used to resolve those choices which are left underspecified in the input.

4.1.6.1 Preferences and Self-Monitoring

Neumann and van Noord have developed an algorithm for self-monitoring of syntactic generation [Neumann and van Noord, 1992]. The purpose of the algorithm is to avoid the generation of ambiguous utterances. The algorithm starts out by generating a sentence which is then parsed in order to determine whether it is ambiguous. If a sentence is found to be ambiguous, its analysis trees are traversed to

find the place in which the derivations differ. At this choice point, another choice is taken, and the result is then again checked for ambiguity. This process is repeated until a non-ambiguous paraphrase of the utterance is generated. The algorithm is quite efficient because it reduces the search to the parts of the derivation which are responsible for the ambiguity of the utterance.

The classic example is the ambiguous sentence (18), which can be paraphrased by the non-ambiguous sentences (19) and (20).

(18) Remove the folder with the system tools

(19) Remove the folder by means of the system tools

(20) Remove the folder containing the system tools

In realistic language models, there are often situations where it is impossible to generate non-ambiguous utterances. This is due to lexical ambiguities (most lexical entries have a number of different readings), structural ambiguities (which are often not even noticed), and scope ambiguities (it is hardly possible to avoid ambiguities in quantifier scope). In such "deadlock" situations, it is desirable to generate a phrase that is *unlikely to be misinterpreted.* In terms of preference values this means that the preference value of the most preferred reading must be *significantly* higher than that of the second most preferred reading.[11]

An example would be the German sentence *Kohl kritisierte Chirac* (Kohl criticised Chirac), which has one reading in which *Kohl* is the subject and *Chirac* the object, and another reading in which the roles are reversed. However, the second reading is so unlikely that there is hardly a chance of a misunderstanding. In this case the word order preferences are so strong that there will be a large distance in the preference values.

An example in which there is not enough difference is the sentence *Florian löst sein Problem mit exzessivem Drogenkonsum* (Florian solves his problems with excessive drug-consumption) which has one reading the the drug consumption is the problem that is solved and another — equally plausible — one where the problem is solved by means of drug consumption. In this case, unambiguous paraphrases must be generated.

Of course the preference values (viewed as probabilities) can be different for different sublanguages. In the computer domain, the word *mouse* is more likely to refer to an input device than to a rodent, and the word *window* is more likely to refer to user interface widget than an opening in a wall, while the probabilities will be reversed in texts from other domains (e.g., pest control, construction).

[11] We won't attempt to formalise what "significantly higher" means since it depends strongly on the given purpose of the generation system. The higher the cost of misunderstanding (the potential damage), the greater the difference in preference values should be.

4.1.7 Determination of Preference Values

We still need to answer the question how the preference values can be determined from a given corpus. What is needed is a corpus annotated with analyses (e.g. HPSG signs). Since such of corpus is currently not available, the ideas in this section cannot be backed up with empirical evidence and rates of accuracy.
There are two kinds of values that must be determined:

- the probabilities of clauses (lexical entries, grammar rules, principles)
- the relative weights of different goals in a clause

For the probabilities of the rules and lexical entries, we would follow the suggestion of Eisele to count the frequency of occurrence in a corpus [Eisele, 1994]. Of course, this raises a number of problems, especially whether just absolute frequencies are counted or frequencies relative to a given context, and about the choice of the context.

The assignment of weights is an even harder problem, since it is not obvious from a corpus how strong the influence of the different factors is. This problem is very complex because it involves a large number of variables that can be varied, and for which an optimal assignment of values must be found. Since this is an optimisation problem with a vast search space, appropriate search techniques must be used, such as evolutionary strategies [Bäck *et al.*, 1991] or genetic algorithms [Holland, 1975; Goldberg, 1989]. These methods work by choosing several initial sets of weights and applying them to a corpus. The resulting performance of each set of weights is used as its *fitness measure.* The sets with the highest fitness value are selected, and further variations (mutation and crossover) are used to produce variations of the sets. This new set of weights is the input for the next step of the algorithm, which will settle on an optimal solution after some number of steps.

4.2 Conclusion

We have presented a formalisation of the notion of degrees of grammaticality by augmenting definite clauses with preference values. Best-first search can be applied in a straightforward manner to obtain solutions with the highest preference values first.

We have shown by giving some examples that a notion of numerical preference value can indeed have beneficial effects for language engineering purposes because it provides criteria for making decisions in non-deterministic situations.

More theoretical and empirical work is required in order to arrive at a satisfactory foundation of preference values. The exact relationship between preference values and probabilities must be clarified, and methods for obtaining preference values from observable data (such as corpora) must be developed.

The methods discussed in this chapter are more speculative than those presented in the previous chapters, and not yet supported by empirical evidence. Since such empirical work involves a lot of effort (e.g. annotation of corpora with HPSG signs), we find it first necessary to argue that the expected results may be worth the effort.

Chapter 5

Implementation

The methods discussed in the previous chapters are implemented in an experimental NLP system called GeLD (Generalised Linguistic Deduction). Considerable attention has been paid to efficiency issues in the implementation.

The aim of the implementation is to combine the useful linguistic deduction algorithms described in the preceding chapters into a logic programming system in order to make them applicable to constraint-based grammars that do not rely on a particular rule format. Grammars can be written in a definite clause language that is augmented with sorted feature terms and the possibility to add control information which determines how grammars are processed.

The emphasis of the system is to provide a framework for easy experimentation with different kinds of processing strategies for different kinds of grammars by adding control information. As such, it is a tool for the developer of processing algorithms, rather than a tool for the development of declarative grammars.

The realization of the linguistic deduction methods in a logic programming framework is achieved by manipulating the *Control* part in the famous equation that defines logic programming [Kowalski, 1979].

Algorithm = Logic + Control

We can characterise our Generalised Linguistic Deduction (GeLD) system by instantiating Logic, Control, and Algorithm as follows:

Logic. As our logic, we have a declarative theory of grammar, stated as definite clauses.

Control. Instead of a fixed control strategy (e.g. Prolog's depth-first top-down strategy, or more specialised schemes such as chart parsers, etc.) control information is added to the clauses of the grammar, in order to permit experimentation with different processing algorithms.

Algorithm. Depending on the control information, different algorithms (and mixtures of algorithms) result.

Like Prolog, GeLD is a proof procedure for definite clause programs. Unlike Prolog, however, our interest is not in providing a universal procedural programming language. Therefore, we do not support control constructs in programs that eliminate some of the solutions (e.g. the cut, negation as failure, or tests of the instantiation state of variables such as `var/1`, `nonvar/1` etc.). We find the control information presented here more appropriate for linguistic applications than the control facilities offered by Prolog. The control constructs we provide instead concern choosing different proof procedures for different goals, and preferred choices in case of non-determinism.

The GeLD system consists of three parts:

1. A sorted feature term language including set descriptions and set constraints, guarded constraints, and linear order constraints.

2. A partial deduction system for grammar transformations

3. A linguistic deduction system which allows the combination of Earley deduction with head-driven processing, top-down processing and direct Prolog execution.

The processing of a grammar proceeds in three stages:

1. Sorted Feature Terms are translated into a Prolog term representation by the ProFIT system (cf. section 5.1).

2. Various grammar transformations are carried out by the partial deduction system according to the specified control information, and the grammar is converted into an internal format (cf. section 5.3.5).

3. The linguistic deduction system is used to parse and generate strings.

The overall architecture of the system is shown in figure 5.1.

The system is implemented in Sicstus Prolog. The sizes of the system's components given in figure 5.2 give a rough idea of the size of the system.

The following sections give an overview of the implementation. Full user documentation for ProFIT and CL-ONE is available.[1]

[1] [Erbach, 1995; Ruessink, 1994; Erbach *et al.*, 1995c]

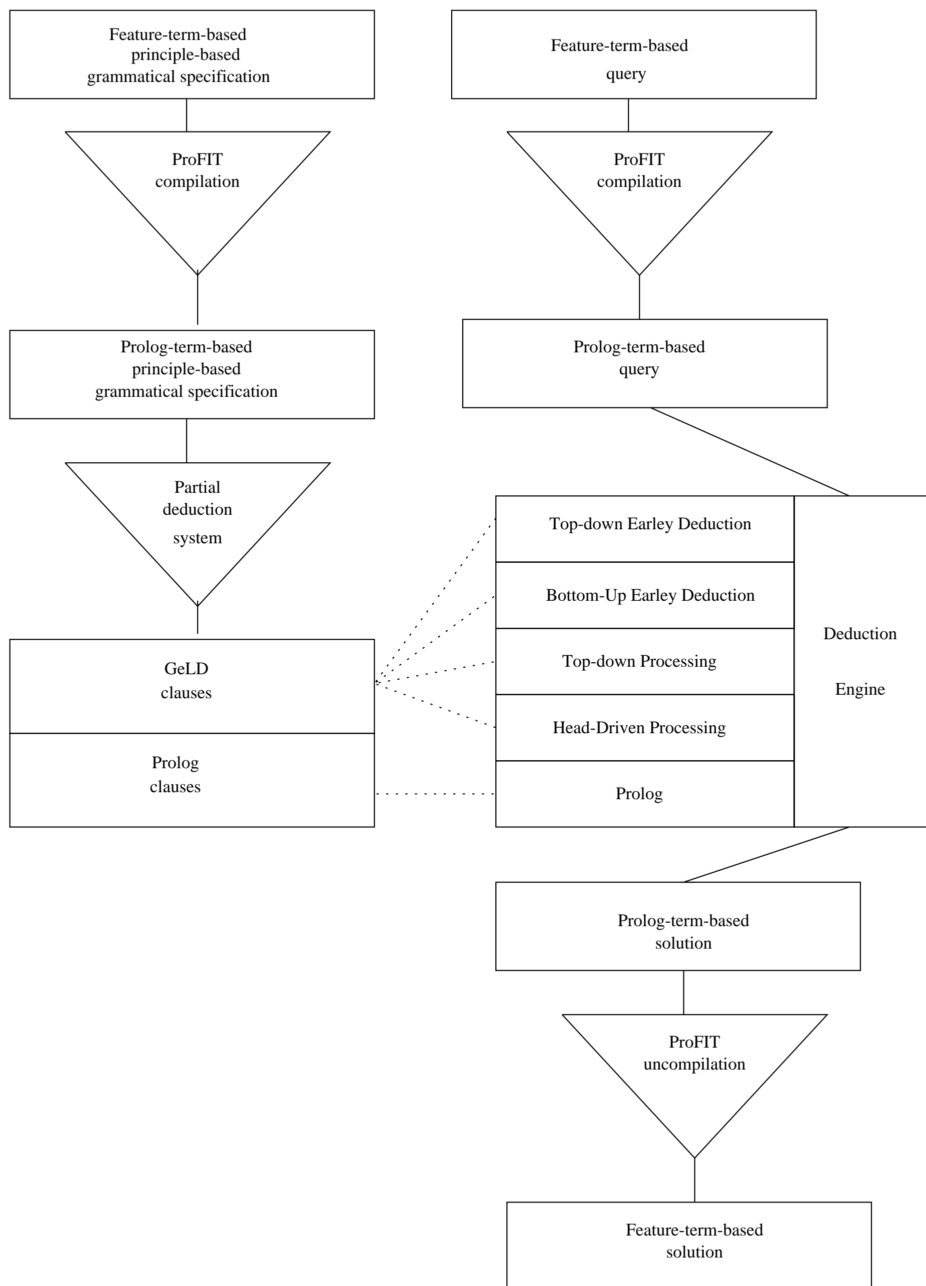

Figure 5.1: Architecture of the linguistic deduction system

System Component	Lines of Prolog Code	Size in Kilobytes
Sorted Feature Terms (ProFIT)	2850	77
Extended Constraint Language (CL-ONE)	3250	67
Deduction System (GeLD)	1320	36
Total	7420	180

Figure 5.2: Components of the GeLD system

5.1 Sorted Feature Terms: ProFIT

It has been noted that first-order Prolog terms provide expressive power equivalent to sorted feature terms [Mellish, 1992]. For example, Carpenter's typed feature structures [Carpenter, 1992] can easily be represented as Prolog terms, if the restriction is given up that the sort hierarchy be a bounded complete partial order.

Such compilation of sorted feature terms into Prolog terms has been successfully used in the Core Language Engine (CLE) [Alshawi, 1991] and in the Advanced Linguistic Engineering Platform (ALEP), [Alshawi *et al.*, 1991].[2] ProFIT extends the compilation techniques of these systems through the handling of multiple inheritance, (cf. [Erbach, 1994b]) and makes them generally available for a wide range of applications by translating programs (or grammars) with sorted feature terms into Prolog programs.

ProFIT is not a grammar formalism, but rather extends any grammar formalism in the logic grammar tradition with the expressive power of sorted feature terms.

5.1.1 The ProFIT Language

The set of ProFIT programs is a superset of Prolog programs. While a Prolog program consists only of definite clauses (Prolog is an untyped language), a ProFIT program consists of datatype declarations and definite clauses. The clauses of a ProFIT program can make use of the data types (sorts, features, templates and finite domains) that are introduced in the declarations. A ProFIT program consists of:

- Declarations for sorts
- Declarations for features
- Declarations for templates

[2]Similar, but less efficient compilation schemes are used in Hirsh's P-PATR [Hirsh, 1986] and Covington's GULP system [Covington, 1989].

- Declarations for finite domains
- Definite clauses

5.1.1.1 Sort Declarations

In addition to unsorted Prolog terms, ProFIT allows sorted feature terms, for which the sorts and features must be declared in advance.

The most general sort is `top`, and all other sorts must be subsorts of `top`. Subsort declarations have the syntax given in (5.1). The declaration states that all Sub_i are subsorts of $Super$, and that all Sub_i are mutually exclusive.

$$Super > [Sub_1, \ldots, Sub_n]. \tag{5.1}$$

It is also possible to provide subsorts that are not mutually exclusive, as in (5.2), where one subsort may be chosen from each of the "dimensions" connected by the $*$ operator. This kind of multi-dimensional inheritance is described in more detail in [Erbach, 1994b].

$$Super > [Sub_{1.1}, \ldots, Sub_{1.n}] * \ldots * [Sub_{k.1}, \ldots, Sub_{k.m}] \tag{5.2}$$

Every sort must only be defined once, i.e. it can appear only once on the left-hand side of the connective $>$.

The sort hierarchy must not contain any cycles, i.e. no sort may be (directly or indirectly) a supersort of itself.

The immediate subsorts of `top` can be declared to be extensional. Two terms which are of an extensional sort are only identical if they have the same most specific sort (terminal node in the sort hierarchy), and if all features are instantiated to ground terms. If a sort is not declared as extensional, it is intensional. Two intensional terms are identical only if they have been unified.

5.1.1.2 Feature Declarations

Unlike unsorted feature formalisms (such as PATR-II), where any feature can be added to any structure, ProFIT follows the notion of appropriateness in Carpenter's logic of typed feature structures [Carpenter, 1992], and introduces features for particular sorts. For each sort, one must declare which features are introduced by it. The features introduced by a sort are inherited by all its subsorts, which may also introduce additional features. A feature must be introduced only at one most general sort. This makes it possible to provide a notation in which the sort name can be omitted since it can be inferred from the use of a feature that is appropriate for that sort.

This notion of appropriateness is desirable for structuring linguistic knowledge, as it prevents the ad-hoc introduction of features, and requires a careful design of

the sort and feature hierarchy. Appropriateness is also a prerequisite for compilation of feature terms into fixed-arity Prolog terms.

Each feature has a sortal restriction for its value. If a feature's value is only restricted to be of sort `top`, then the sortal restriction can be omitted. The syntax of feature declarations is given in (5.3).

$$Sort\ \texttt{intro}\ [Feature_1 : Restr_1, \ldots, Feature_n : Restr_n]. \tag{5.3}$$

The following declaration defines a sort *binary_tree* with subsorts *leaf* and *internal_node*. The sort *binary tree* introduces the feature *label* and its subsort adds the features *left_daughter* and *right_daughter*. If a sort has subsorts and introduces features, these are combined in one declaration.

```
binary_tree >  [leaf,internal_node]
   intro [label].

internal_node intro
   [left_daughter:binary_tree,
    right_daughter:binary_tree].
```

5.1.1.3 Sorted Feature Terms

On the basis of the declarations, sorted feature terms can be used in definite clauses in addition to and in combination with Prolog terms. A Prolog term can have a feature term as its argument, and a feature can have a Prolog term as its value. This avoids potential interface problems between different representations, since terms do not have to be translated between different languages. As an example, semantic representations in first-order terms can be used as feature values, but do not need to be encoded as feature terms.

Sorted feature terms consist of a specification of the sort of the term (5.4), or the specification of a feature value (5.5), or a conjunction of terms (5.6). A complete BNF of all ProFIT terms is given in the appendix.

$$< Sort \tag{5.4}$$

$$Feature\ !\ Value \tag{5.5}$$

$$Term\ \&\ Term \tag{5.6}$$

The following clauses (based on HPSG) state that a structure is saturated if its subcat value is the empty list, and that a structure satisfies the Head Feature Principle if its head features are identical with the head features of its head daughter.[3]

[3]These clauses assume appropriate declarations for the sort `elist`, and for the features `synsem`, `local`, `cat`, `subcat`, `head`, `dtrs` and `head_dtr`.

Note that these clauses provide a concise notation because uninstantiated features can be omitted, and the sorts of structures do not have to be mentioned because they can be inferred from use of the features.

```
saturated( synsem!local!cat!subcat!<elist ).

head_feature_principle( synsem!local!cat!head!X &
                        dtrs!head_dtr!synsem!local!cat!head!X ).
```

Note that conjunction also provides the possibility to tag a Prolog term or feature term with a variable (`Var` & `Term`).

5.1.1.4 Feature Search

In the organisation of linguistic knowledge, feature structures are often deeply embedded, due to the need to group together sets of features whose value can be structure-shared. In the course of grammar development, it is often necessary to change the "location" of a feature in order to get the right structuring of information.

Such a change of the "feature geometry" makes it necessary to change the path in all references to a feature. This is often done by introducing templates whose sole purpose is the abbreviation of a path to a feature.

ProFIT provides a mechanism to search for paths to features automatically provided that the sortal restrictions for the feature values are strong enough to ensure that there is a unique minimal path. A path is minimal if it does not contain any repeated features or sorts.

The sort from which to start the feature search must either be specified explicitly (5.7) or implicitly given through the sortal restriction of a feature value, in which case the sort can be omitted and the expression (5.8) can be used.

$$\mathit{Sort} >>> \mathit{Feature}\ !\ \mathit{Term} \tag{5.7}$$

$$>>> \mathit{Feature}\ !\ \mathit{Term} \tag{5.8}$$

The following clause makes use of feature search to express the Head Feature Principle.

```
head_feature_principle( sign>>>head!X &
                        dtrs!head_dtr! >>>head!X ).
```

While this abbreviation for feature paths is new for formal description languages, similar abbreviatory conventions are often used in linguistic publications. They are easily and unambiguously understood if there is only one unique path to the feature which is not embedded in another structure of the same sort.

5.1.1.5 Templates

The purpose of templates is to give names to frequently used structures. In addition to being an abbreviatory device, the template mechanism serves three other purposes.

- Abstraction and interfacing by providing a fixed name for a value that may change,
- Partial evaluation,
- Functional notation that can make specifications easier to understand.

Templates are defined by expressions of the form (5.9), where *Name* and *Value* can be arbitrary ProFIT terms, including variables, and template calls. There can be several template definitions with the same name on the left-hand side (relational templates). Since templates are expanded at compile time, template definitions must not be recursive.

$$\textit{Name} := \textit{Value}. \tag{5.9}$$

Templates are called by using the template name prefixed with `@` in a ProFIT term.

Abstraction makes it possible to change data structures by changing their definition only at one point. Abstraction also ensures that databases (e.g. lexicons) which make use of these abstractions can be re-used in different kinds of applications where different data structures represent these abstractions.

Abstraction through templates is also useful for defining interfaces between grammars and processing modules. If semantic processing must access the semantic representations of different grammars, this can be done if the semantic module makes use of a template defined for each grammar that indicates where in the feature structure the semantic information is located, as in the following example for HPSG.

```
semantics(synsem!local!cont!Sem) := Sem.
```

Partial evaluation is achieved when a structure (say a principle of a grammar) is represented by a template that gets expanded at compile time, and does not have to be called as a goal during processing.

We show the use of templates for providing functional notation by a simple example, in which the expression `@first(X)` stands for the first element of list X, and `@rest(X)` stands for the tail of list X, as defined by the following template definition.

```
first([First|Rest]) := First.
rest([First|Rest])  := Rest.
```

The member relation can be defined with the following clauses, which correspond very closely to the natural-language statement of the member relation given as comments. Note that expansion of the templates yields the usual definition of the member relation in Prolog.

```
% The first element of a list is a member of the list.
member(@first(List),List).

% Element is a member of a list if it is a member of the rest of the list
member(Element,List) :-
        member(Element,@rest(List)).
```

The expressive power of an n-place template is the same as that of an n+1 place fact.

5.1.1.6 Disjunction

Disjunction in the general case cannot be encoded in a Prolog term representation.[4] Since a general treatment of disjunction would involve too much computational overhead, we provide disjunctive terms only as syntactic sugar. Clauses containing disjunctive terms are compiled to several clauses, one for each consistent combination of disjuncts. Disjunctive terms make it possible to state facts that belong together in one clause, as the following formulation of the Semantics Principle of HPSG, which states that the content value of a head-adjunct structure is the content value of the adjunct daughter, and the content value of the other headed structures (head-complement, head-marker, and head-filler structure) is the content value of the head daughter.

```
semantics_principle( (<head_adj &
                       >>>cont!S & >>>adj_dtr!>>>cont!S )
                   or
                     ( (<head_comp or <head_marker or <head_filler) &
                       >>>cont!S & >>>head_dtr!>>>cont!S )
                   ).
```

For disjunctions of atoms, there exists a Prolog term representation, which is described below.

5.1.1.7 Finite Domains

For domains involving only a finite set of atoms as possible values, it is possible to provide a Prolog term representation (due to Colmerauer, and described by Mellish [Mellish, 1988]) to encode any subset of the possible values in one term. For reasons of space, we do not give a description of the encoding here.

---

[4] see the complexity analysis by Brew [Brew, 1991].

```
agr fin_dom [1,2,3] * [sg,pl].

verb(sleeps,3&sg).
verb(sleep, ~(3&sg)).
verb(am, 1&sg).
verb(is, 3&sg).
verb(are, 2 or pl).

np('I', 1&sg).
np(you, 2@agr).
```

Figure 5.3: An example of finite domains

Consider the agreement features *person* (with values `1, 2` and `3`) and *number* (with values `sg` and `pl`). For the two features together there are six possible combinations of values (1&sg, 2&sg, 3&sg, 1&pl, 2&pl, 3&pl). Any subset of this set of possible values can be encoded as one Prolog term. Figure 5.3 shows the declaration needed for this finite domain, and some clauses that refer to subsets of the possible agreement values by making use of the logical connectives `~` (negation), & (conjunction), `or` (disjunction).[5]

This kind of encoding is only applicable to domains which have no coreferences reaching into them, in the example only the agreement features as a whole can be coreferent with other agreement features, but not the values of person or number in isolation. This kind of encoding is useful to avoid the creation of choice points for the lexicon of languages where one inflectional form may correspond to different feature values.

5.1.1.8 Cyclic Terms

Unlike Prolog, the concrete syntax of ProFIT allows one to write down cyclic terms by making use of conjunction:

```
X & f(X).
```

Cyclic terms no longer constitute a theoretical or practical problem in logic programming, and almost all modern Prolog implementations can perform their unification (although they can't print them out). Cyclic terms arise naturally in

[5] The syntax for finite domain terms is `Term@Domain`. However, when atoms from a finite domains are combined by the conjunction, disjunction and negation connectives, the specification of the domain can be omitted. In the example, the domain must only be specified for the value 2, which could otherwise be confused with the integer 2.

NLP through unification of non-cyclic terms, e.g. the Subcategorisation Principle and the Spec Principle of HPSG.

ProFIT supports cyclic terms by being able to print them out as solutions. In order to do this, a costly 'occurs check' must be performed. Since this must be done only when results are printed out as ProFIT terms, it does not affect the runtime performance.

5.1.2 From ProFIT Terms to Prolog Terms

5.1.2.1 Compilation of Sorted Feature Terms

The compilation of sorted feature terms into a Prolog term representation is based on the following principles, which are explained in more detail in [Mellish, 1988; Mellish, 1992; Schöter, 1993; Erbach, 1994b].

- The Prolog representation of a sort is an instance of the Prolog representation of its supersorts.
- Features are represented by arguments. If a feature is introduced by a subsort, then the argument is added to the term that further instantiates its supersort.
- Mutually exclusive sorts have different functors at the same argument position, so that their unification fails.

We illustrate these principles for compiling sorted feature terms into Prolog terms with an example from HPSG. The following declaration states that the sort `sign` has two mutually exclusive subsorts `lexical` and `phrasal` and introduces four features.

```
sign > [lexical,phrasal] intro [phon,synsem,qstore,retrieved]
```

In the corresponding Prolog term representation below, the first argument is a variable whose only purpose is being able to test whether two terms are coreferent or whether they just happen to have the same sort and the same values for all features. In case of extensional sorts (see section 5.1.1.1), this variable is omitted. The second argument can be further instantiated for the subsorts, and the remaining four arguments correspond to the four features.

```
$sign(Var,LexPhras,Phon,Synsem,Qstore,Retrieved)
```

The following declaration introduces two sort hierarchy "dimensions" for subsorts of `phrasal`, and one new feature. The corresponding Prolog term representation instantiates the representation for the sort `sign` further, and leaves argument positions that can be instantiated further by the subsorts of `phrasal`, and for the newly introduced feature `daughters`.

```
phrasal > [headed,non_headed] * [decl,int,rel]
     intro [daughters].
```

```
$sign(Var,$phrasal(Phrasesort,Clausesort,Daughters),Phon,Synsem,Qstore,Retrieved)
```

5.1.2.2 Compilation of Finite Domains

The compilation of finite domains into Prolog terms is performed by the "brute-force" method described in [Mellish, 1988]. A finite domain with n possible domain elements is represented by a Prolog term with $n+1$ arguments. Each domain element is associated with a pair of adjacent arguments. For example, the agreement domain `agr` from section 5.1.1.7 with its six elements (1&sg, 2&sg, 3&sg, 1&pl, 2&pl, 3&pl) is represented by a Prolog term with seven arguments.

```
$agr(1,A,B,C,D,E,0)
```

Note that the first and last argument *must* be different. In the example, this is achieved by instantiation with different atoms, but an inequality constraint (Prolog II's `dif`) would serve the same purpose. We assume that the domain element 1&sg corresponds to the first and second arguments, 2&sg to the second and third arguments, and so on, as illustrated below.

```
$agr( 1  ,  A  ,  B  ,  C  ,  D  ,  E  ,  0 )
        1sg   2sg   3sg   1pl   2pl   3pl
```

A domain description is translated into a Prolog term by unifying the argument pairs that are *excluded* by the description. For example, the domain description `2 or pl` excludes 1&sg and 3&sg, so that the the first and second argument are unified (1&sg), as well as the third and fourth (3&sg).

```
$agr(1,1,X,X,D,E,0)
```

When two such Prolog terms are unified, the union of their excluded elements is computed by unificatation, or conversely the intersection of the elements which are in the domain description. The unification of two finite domain terms is successful as long as they have at least one element in common. When two terms are unified which have no element in common, i.e., they *exclude* all domain elements, then unification fails because all arguments become unified with each other, including the first and last arguments, which are different.

5.1.3 ProFIT Implementation

ProFIT has been implemented in Quintus and Sicstus Prolog, and should run with any Prolog that conforms to or extends the proposed ISO Prolog standard.

All facilities needed for the development of application programs, for example the module system and declarations (dynamic, multifile etc.) are supported by ProFIT.

Compilation of a ProFIT file generates two kinds of files as output.

1. Declaration files that contain information for compilation, derived from the declarations.

2. A program file (a Prolog program) that contains the clauses, with all ProFIT terms compiled into their Prolog term representation.

The program file is compiled on the basis of the declaration files. If the input and output of the program (the exported predicates of a module) only make use of Prolog terms, and feature terms are only used for internal purposes, then the program file is all that is needed. This is for example the case with a grammar that uses feature terms for grammatical description, but whose input and output (e.g. graphemic form and logical form) are represented as normal Prolog terms.

Declarations and clauses can come in any order in a ProFIT file, so that the declarations can be written next to the clauses that make use of them. Declarations, templates and clauses can be distributed across several files, so that it becomes possible to modify clauses without having to recompile the declarations, or to make changes to parts of the sort hierarchy without having to recompile the entire hierarchy.

Sort checking can be turned off for debugging purposes, and feature search and handling of cyclic terms can be turned off in order to speed up the compilation process if they are not needed.

Error handling is currently being improved to give informative and helpful warnings in case of undefined sorts, features and templates, or cyclic sort hierarchies or template definitions.

For the development of ProFIT programs and grammars, it is necessary to give input and output and debugging information in ProFIT terms, since the Prolog term representation is not very readable. ProFIT provides a user interface which

- accepts queries containing ProFIT terms, and translates them into Prolog queries,

- converts the solutions to the Prolog query back into ProFIT terms before printing them out,

- prints out debugging information as ProFIT terms.

When a solution or debugging information is printed out, uninstantiated features are omitted, and shared structures are printed only once and represented by variables on subsequent occurences.

A pretty-printer is provided that produces a neatly formatted screen output of ProFIT terms, and is configurable by the user. ProFIT terms can also be output in LaTeX format, and an interface to graphical grammar development environments such as Pleuk [Calder and Humphreys, 1993], HDrug [van Noord, 1994] or the graphical feature editor Fegramed (developed at DFKI) is foreseen.

In order to give a rough idea of the efficiency gains of a compilation into Prolog terms instead of using a feature term unification algorithm implemented on top of Prolog, we have compared the runtimes with ALE and the Eisele-Dörre algorithm for unsorted feature unification for the following tasks:

1. unification of (unsorted) feature structures,
2. unification of inconsistent feature structures (unification failure),
3. unification of sorts,
4. lookup of one of 10000 feature structures (e.g. lexical items),
5. parsing with an HPSG grammar to provide a mix of the above tasks.

The timings indicate that ProFIT is 5 to 10 times faster than a system which implements a unification algorithm on top of Prolog, a result which is predicted by the studies of Schöter [Schöter, 1993] and the experience of the Core Language Engine. An overview of the results is given in appendix B.1.

5.2 Extensions of the Constraint Language

5.2.1 Set Descriptions and Set Constraints

For set descriptions and set constraints, we use the set constraint solver developed by Suresh Manandhar [Manandhar, 1994; Erbach *et al.*, 1994a; Erbach *et al.*, 1994b; Erbach *et al.*, 1995b], and integrated with ProFIT by S. Manandhar and the author.

The set descriptions and set constraints shown in figure 5.5 are allowed in definite clauses.

Full details about the implementation can be found in the deliverables of the RGR project [Erbach *et al.*, 1994a; Erbach *et al.*, 1995a].

5.2.2 Guarded Constraints

The implementation of guarded constraints supports the following general purpose syntax:

```
PFT  := <Sort                  Term of a sort Sort
      | Feature!PFT            Feature-Value pair
      | PFT & PFT              Conjunction of terms
      | PROLOGTERM             Any Prolog term
      | FINDOM                 Finite Domain term, BNF see below
      | @Template              Template call
      | ‘ PFT                  Quoted term, is not translated
      | ‘‘ PFT                 Main functor is not translated
      | >>>Feature!PFT         Search for a feature
      | Sort>>>Feature!PFT     short for <Sort & >>>Feature!PFT
      | PFT or PFT             Disjunction; expands to multiple terms


FINDOM  :=  FINDOM@FiniteDomainName
         |   FINDOM
         |  FINDOM & FINDOM
         |  FINDOM or FINDOM
         |  Atom
```

Figure 5.4: BNF for ProFIT terms

Set Constraint	Meaning	Syntax for variable X
empty set	X is the empty set	{ }
element	E is an element of X	exist(E)
set description	X contains the elements $E_1 \ldots E_n$ (but they need not be disjoint)	$\{E_1, \ldots, E_n\}$
fixed cardinality set	X contains the disjoint elements $E_1 \ldots E_n$	fixed_card($\{E_1, \ldots, E_n\}$)
subset	X is a subset of Y	subset(Y)
union	X is the union of Y and Z	union(Y,Z)
intersection	X is the intersection of Y and Z	intersection(Y,Z)
disjointness	the members of X and Y are disjoint	disjoint(Y)
disjoint union	X is the disjoint union of Y and Z	dis_union(Y,Z)

Figure 5.5: Syntax of set constraints

LP Constraint	Meaning	Syntax for variable X
precedence	X precedes Y	precedes(Y)
precedence equals	X precedes or is equal to Y	precedes_equals(Y)
first daughter	X precedes all other elements of domain Y	fst_daughter(Y)
domain precedence	(every element of) domain X precedes (every element of) domain Y	dom_precedes(y)
guard on precedence	if X precedes Y then X is unified with S, otherwise X is unified with T	if precedes(Y) then S else T

Figure 5.6: Syntax of linear precedence

case([$condition_1$ `#>`$action_1$,
...
$condition_n$ `#>`$action_n$
])
else $action_{n+1}$

Each of the $action_i$ can be any term or another guarded constraint. Each of the $condition_i$ (also known as *guard*) is restricted to one of the following forms (the variables $\exists x_1, \ldots, x_n$ stand for existentially quantified variables).

$$condition \longrightarrow \exists x_1, \ldots, x_n \; feature_term \;|\; \exists x_1, \ldots, x_n \; exists(feature_term) \;|\; precedes(x, y)$$

The constraint *if* G *then* S *else* T can then be thought of as syntactic sugar for the statement $case([G \Rightarrow \; S])$ *else* T. The constraint *if* G *then* S *else* T can also be thought as syntactic sugar for the constraint *if* G *then* $S \sqcap$ *if* $\neg G$ *then* T. However, since the test for both G and $\neg G$ can be done simultaneously, the former representation is more efficient.

5.2.3 Linear Precedence Constraints

In figure 5.6 we describe the syntax of the linear precedence constraints supported by our implementation; for the formal semantics, refer to [Manandhar, 1995].

5.2.4 Interaction between Constraint Handling and Tabulation

Unfortunately, there is no representation of set descriptions as Prolog terms that allows unification of set descriptions to be performed by Prolog unification of their representations. Therefore, programs containing set descriptions cannot be compiled to Prolog programs, but need a meta-interpreter that takes care of set unification.

For datatypes such as sets that do not have such a convenient Prolog term representation, we use the following technique: In the Prolog term representation, we leave a variable at the position for an object which has such a datatype, and keep a separate list in which we can look up the object which belongs to that variable. In addition, we use the coroutining mechanism of Sicstus Prolog to ensure that unification of the externally represented data objects is performed whenever two of the variables that replace them in the Prolog term representation are unified. The same coroutining mechanism is used to enforce constraints on sets (e.g. subset constraints), implicational constraints, and linear precedence constraints.

The treatment of set constraints makes heavy use of the coroutining facilities of Sicstus Prolog. Where these "blocked goals" have not yet been evaluated, they belong to the clause containing the set description. Whenever a clause is stored as an item, these "blocked goals" must be copied, which causes a significant slowdown, as experiments have shown.[6] This slowdown is due to the large number of "blocked" goals that are introduced for every pair of set descriptions occurring in a term in order to ensure that the set descriptions get unified in case the variables associated with them get unified. Further "blocked goals" are introduced by the membership, subset, union, and intersection constraints.

Of course this does not mean that there is a fundamental incompatibility between Earley deduction and the use of set descriptions and constraints. What it means is that the efficient implementation of set descriptions is not really feasible on top of today's Prolog systems, a problem which is made worse by the fact that tabulation (memoing) in Prolog is not handled very efficiently. Combining both of these sources of inefficiency does therefore not appear to be a wise move.

Since this thesis is mainly concerned with the study of Earley deduction techniques for efficient NLP, we opt for tabulation at the expense of set descriptions at the current stage of the implementation.

For the future, we expect significant improvements on both fronts. With respect to set descriptions, we expect a significant improvement in the near future

[6] In the implementation of CL-ONE, both a chart parser and a left corner parser for ALE-style grammars are provided. If set descriptions are present in clauses, it is recommended to use the left corner parser because of the performance problems that the chart parser suffers due to the large (2 – 4 times slower according to experiments done with a small grammar) slowdown through the copying of "blocked goals."

from the use of attributed variables [Holzbaur, 1992] in the next version of Sicstus Prolog (3.0), which form a natural basis for the implementation of set constraints since they allow user-defined unification.

In the long run, we expect sets to become a built-in datatype in logic programming languages, which allows a much more efficient implementation.

On the tabulation front, we follow with great interest the development of XSB Prolog, which uses top-down Earley deduction as its basic operation. Any advances in the implementation of tabulation made in this respect are also applicable for a more efficient implementation of the bottom-up variant.

It is these developments that have encouraged the use of the extended constraint language for expository purposes in the preceding chapters, even though the current implementation cannot support their efficient processing in combination with Earley deduction strategies.

5.3 The Generalised Linguistic Deduction (GeLD) System

This section gives an overview of the GeLD system, and in particular of the different kinds of control information that can be added to clauses in order to determine their behaviour with respect to partial deduction, and to the different deduction systems.

The compilation of programs, and the workings of the partial deduction system are described, whereas the description of the deduction engine is the subject of section 5.4.

5.3.1 Control Information

5.3.1.1 Clause Types

The most important kind of control information concerns the question of what a clause should be compiled into. This is indicated by the main connective that links the consequent and the antecedent of the clause (cf. figure 5.7).

Clause Type	Syntax
Prolog clause	C :- A.
GeLD clause	C <- A.
Macro	C <= A.
Fact	C.

Figure 5.7: Clause types

The choice of a clause type determines how the clause is processed. The four classes have the following appropriate uses:

Prolog clause: Prolog clauses are compiled directly into Prolog clauses, although some of their goals may be expanded by partial deduction. They can later only be executed directly by Prolog, which is in certain cases more efficient than use of a deduction system implemented on top of Prolog, but suffers from the usual problems of Prolog's search strategy. Goals in a Prolog clause can be passed to the GeLD system for an alternative deduction strategy. Prolog clauses are primarily intended for calling externally defined procedures, and for side effects.

GeLD clause: GeLD clauses are compiled into a form that can be interpreted by the deduction system described below. If a GeLD clause has an empty body (usually as the result of partial deduction), it is compiled into a Prolog fact in order to gain efficient access to it through Prolog's indexing mechanisms. This is especially important in order to ensure efficienct access to large data bases, e.g. lexicons with thousands of entries.

Macro: A "macro" is a clause that is used during the partial deduction phase, but for which no representation is created in the target file. This not only avoids redundant code in the target file, but also allows the expansion of recursive macros, provided that their relevant variables are properly instantiated.

Fact: Clauses without goals (possibly as the result of partial deduction) are always compiled into Prolog facts, since obviously they need not contain any control information, but are trivially provable. The compilation into Prolog facts is used to make the Prolog compiler perform indexing on these clauses, so that even large databases of facts can be efficiently accessed.

This distinction into several clause types allows a proper division of labour between those clauses which can easily be handled by Prolog (and benefit of efficient compilation) and those that need the deduction system as a meta-interpreter.

5.3.1.2 Goal Types

A goal in the body of a clause can be annotated with control information (goal type) that specifies how it should be executed. A goal type is specified by a one-letter code which is prefixed to the goal. A list of possible goal types is given in figure 5.8.
The significance of each goal type is explained below:

c: chart-based processing. This option refers to the bottom-up Earley deduction developed in chapter 3. A new chart is created for proving the goal.

d: top-down processing. This option is like Prolog's proof strategy, but a meta-interpreter is used instead. This option should be chosen if goals other than Prolog goals can occur in the proof of the goal.

Code	Goal Type
c	**c**hart-based processing
d	top-**d** own goal
e	(top-down) **E**arley deduction
h	**H**ead-driven processing
i	**i**mmediate execution
m	**m**acro
p	**P**rolog goal
w	**w**aiting goal

Figure 5.8: Goal types

e: (top-down) Earley deduction. These goals are executed by the classic top-down Earley deduction algorithm. A new chart is created for the proof of an **e**-goal.

i: immediate execution. **i** goals are executed immediately at compile time by calling them as Prolog goals. This is a form of partial deduction: **i** goals never appear in the output of the partial deduction; instead there will be a number of new clauses, depending on how many solutions the **i**-goal has. **i**-goals are used for calling Prolog built-in predicates or predicates of a program that has previously been consulted.[7]

m: macro. These goals are replaced in a partial deduction step. An **m**-goal is matched against clauses (of the same program) with the same predicate in their consequent, and replaced by the goals in the antecedents of these clauses. If the goals in these clauses contain again **m** or **i**-goals, the process is applied recursively. It is possible that the goal completely disappears in the process by being (successively) replaced by the empty body.

p: Prolog goal. Prolog goals are executed directly by Prolog at runtime. Clauses for goals that are called as Prolog goals should be either Prolog built-in predicates or specified as Prolog clauses.

h: head-driven goal. These goals are executed according to a head-driven strategy.

w: waiting goal. A waiting goal is one that *waits* to be combined with a passive item in the course of (bottom-up or top-down) Earley deduction, or with the result of a bottom-up deduction step in head-driven processing. When a waiting goal is encountered in the proof of a goal, its clause is turned into

[7]Some Prolog systems make a distinction between *consulting* and *compiling*, where the latter is more efficient. When the term *consult* is used in this chapter, we generally mean the loading of Prolog clauses by the most efficient method available.

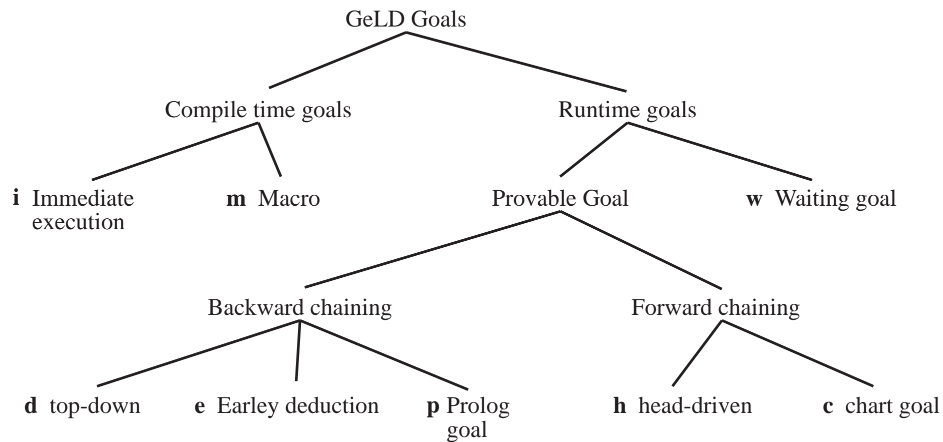

Figure 5.9: Hierarchy of goal types

an active item. Any clauses whose first goal is a waiting goal are turned into items at compile time.

These goal types can be divided into those which are used at compile time (**i** and **m**) and the others which are used at runtime. The runtime goals can be divided into waiting goals (**w**) which are passive and activated in Earley deduction or head-driven processing, and provable goals, for which proof procedures are implemented in the system. The provable goals are further subdivided into backward chaining (top-down) goals (**d**, **e** and **p**) and forward chaining (bottom-up) goals (**c** and **h**). This hierarchy of goal types is shown in figure 5.9.

In addition to goal types for individual goals, it is possible to specify a default goal type for each predicate. This default will be used for every goal which is not given an explicit goal type. The notation for the specification of a default is the following:

```
:- goal_type(Functor/Arity,Type).
```

The following directives specify that the default goal type for `sign/1` is **w** and for `head_feature_principle/1` is **m**.

```
:- goal_type(sign/1,w).
:- goal_type(head_feature_principle/1,m).
```

If there is no goal type and no default for a goal in a clause, then it becomes a **p** goal. This is in accordance with the strategy to use execution by Prolog whenever possible for reasons of efficiency.

There is a restriction for the goal types in the body of a Prolog clause or a macro: they must not contain any waiting goals.

5.3.1.3 Base Case Lookup

In order to prove **c** and **h** goals by means of forward chaining, the deduction system must know which base cases of recursively defined predicates to use for a given goal. For example, for computing the factorial function bottom-up, the base case should always be the factorial of 0. For linguistic deduction, the base cases should for example be the lexical entries of the words contained in a string to be parsed, or a non-chain rule having the same semantics as a sign to be generated.

At present, we have no general method for determining the appropriate base cases for given goals. Therefore, the user must specify which base case is appropriate for which goal. This is done by providing clauses for the relation `lookup/3` for head-driven processing, and for `lookup/4` for bottom-up Earley deduction. The prodcures `lookup/3` and `lookup/4` have the following arguments (the first three are shared).

1. The first argument is the goal to be proven.

2. The second argument is the base case that is used to start a bottom-up proof of the goal.

3. The third argument is the goal type according to which the base case is proven. If the base case must simply be looked up as a fact, then it can be specified as a **p**-goal.

4. The extra fourth argument in the case of bottom-up Earley deduction is the index of the passive item that is created as the result of base case lookup.

Figure 5.10 shows a specification of the lookup relation for bottom-up Earley deduction (based on the specification of the lookup relation in figure 3.5).

Assuming that the derivation via a chain rule or a non-chain rule is encoded in a sign as the value of the feature DERIV, the following is the lookup relation for semantic-head driven generation (goal type: **h**).

```
lookup(sign(sem!Sem), sign(sem!Sem & deriv!non_chain), d).      (5.10)
```

5.3.1.4 Priorities

The goals in the body of the clause can be given a numerical priority which determines the order in which they are processed. This is merely a notational convenience which makes re-ordering goals easier than cut-and-paste which always leads into trouble when the last goal of the clause is affected.

Goals without a priority are assumed to have the priority 0.

```
lookup(sign(phon!PhonList),
       lexical_sign(phon![Word] & synsem!X ),
       d,
       Begin-End ) <-
        nth_member(Word,Begin,End,PhonList),
        lexicon(Word,X).

nth_member(X,0,1,[X|_]).
nth_member(X,N1,N2,[_|R]) <-
        nth_member(X,N0,N1,R),
        N2 is N1 + 1.
```

Figure 5.10: Specification of the relation `lookup` in GeLD

5.3.2 Coroutining

GeLD supports the same form of coroutining as Sicstus Prolog (`when/2`). Wherever possible, the coroutining is delegated to Sicstus Prolog. For example, the relevant clause in the interpreter for **d** goals is the following.

```
prove(when(Condition,Goal)) :-
     when(Condition,prove(Goal)).
```

5.3.3 Preference Values

In the source file, predicates can optionally be associated with preference values. The syntax for this is given in (5.11).

$$\text{Predicate} \ \# \ \text{Pref} \tag{5.11}$$

If a predicate is associated with a preference value, its preference value must be present on all occurrences. For reasons of efficiency, an n-place predicate with a preference value is compiled into an $n+1$ place predicate, where the last argument is reserved for the preference value.[8]

In the following example, the 3-place predicate `insert` with preference value is compiled into a 4-place predicate. The same kind of compilation of preference values takes place whether the predicate is defined as a Prolog clause, a GeLD clause or a macro.

[8] For this reason, using an n-place predicate with preference value *and* an unrelated $n+1$-place predicate without preference value in the same program must be avoided. This situation is similar to the caveat against using an n-place DCG category and an unrelated $n+2$-place predicate in the same program.

Input clauses:

```
insert(X,R,[X|R])#1.
insert(X,[F|R],[F|New])#0.9*Pref :-
        insert(X,R,New)#Pref.
```

Compiled clauses:

```
insert(X,R,[X|R],1).
insert(X,[F|R],[F|New],0.9*Pref) :-
        insert(X,R,New,Pref).
```

Figure 5.11: Compilation of clauses with preference values

This addition of an extra argument (similar to the addition of difference list arguments in case of DCG compilation) allows the direct compilation of programs/grammars with preference values into Prolog programs, and their efficient processing without the need for a meta-interpreter.

During the processing, variables in preference formulae get further instantiated. Preference formulae are only evaluated where this is necessary to control the processing, i.e., in order to calculate the priority of an item that is about to be added to the agenda.

5.3.4 Compilation

The first step of the compilation consists in bringing the program clauses into a normal form to which partial deduction can be applied. This step is necessary because we want to perform partial deduction to simplify clauses of a program by using other clauses of the same program. This is different from partial deduction during the compilation of Prolog programs by making use of `term_expansion/2` or `goal_expansion/2`, which can only use clauses for partial deduction that have been previously consulted.

Bringing a clause into normal form consists of three steps:

1. Integration of preference values (cf. section 5.3.3)

2. Ordering of goals in the body of a clause according to their priority (cf. section 5.3.1.4)

3. Storage of clauses in normal form in the internal Prolog database

All clauses in the internal database have the same normal form, irrespective of whether they are Prolog clauses, GeLD clauses, macros or facts (although this information is preserved):

```
geld_clause(Conseq,Pref,Type,Body)
```

`Conseq` is the consequent of the clause. `Pref` is the preference value associated with the consequent, if any, and the atom `nil` otherwise. `Type` is an atom that encodes the difference between Prolog (unit and non-unit) clauses (`prolog`), Geld clauses (`geld`), and macros (`macro`). `Body` is a list of goals, which are also in a normal form:

```
goal(Goal,Type,Pref)
```

Here, `Goal` is the goal itself, `Type` is the goal type (cf. section 5.3.1.2), and `Pref` is the preference value of the goal. The priority given to a goal does not appear here any more because the goals are already ordered according to their priority.

This normal form forms the basis for the partial deduction step described in the following section.

5.3.5 Partial Deduction

As described in chapter 2, partial deduction is a technique for program transformation which takes a logic program as input and gives an equivalent logic program, in which some goals are expanded, as output. In our case, the input logic program can be the output of the ProFIT compiler, transformed into a normal form described in the previous section.

In the actual partial deduction step, each clause is transformed into (zero, one, or more) new clauses by partial deduction, making use of the other clauses stored in the internal database. The clauses which result from the second step are written to a file and then compiled (in order to be able to benefit from the indexing which takes place for compiled predicates).

i goals

In this section, we discuss how the goals with types **i** and **m** are treated by the partial deduction system. The Prolog code of the core of the partial deduction system is given in appendix C.

In the partial deduction step, all **i** goals in Prolog clauses and GeLD clauses are executed and disappear, and the constraint that is the solution of the goal is added to the constraint of the clause.

$$\frac{A \leftarrow \Gamma \ \wedge \mathbf{i}B \ \wedge \Delta}{\sigma(A \leftarrow \Gamma \ \wedge \Delta)} \tag{5.12}$$

σ is the constraint (a substitution in case of Prolog) that is returned as a solution of proving goal B.

m goals

M goals also disappear in the partial deduction step, but unlike **i** goals their are replaced by a (possibly empty) sequence of goals. The sequence of goals that replaces the **m** goal can be the body of a Prolog clause, a GeLD clause, or of a macro.

$$\frac{\begin{array}{c} A \leftarrow \Gamma \wedge \mathbf{m}B \wedge \Delta \\ B' \leftarrow \Theta \end{array}}{\sigma(A \leftarrow \Gamma \wedge \Theta \wedge \Delta)} \tag{5.13}$$

σ is the merged constraint (unifying substitution in case of Prolog) of B and B'.

Both of the above inference rules are applied until no more **i** or **m** goals are present in the clause. In case where a rule contains more than one goal to which one of the inference rules can be applied, then it is always applied to the leftmost such goal.

This order of execution is often necessary in order to ensure that the partial deduction process terminates and generates only a finite number of clauses. Often partial deduction performed on one **m** or **i** goal which can instantiate variables that ensure that a goal following it has only a finite number of solutions.

The inference rules are applied to a clause until it contains no more **i** or **m** goals; then the clause is written to a file in a format that allows efficient execution and exploits the indexing of Prolog when it is compiled. Clauses with an empty antecedent are always represented as Prolog facts, since they don't need any control information, and in order to provide the most efficient processing and indexing.

Prolog rules are translated back into Prolog format after partial deduction. This allows them to be executed directly by Prolog, thereby exploiting Prolog compilation and processing.

GeLD clauses are written to a file in the following form.

```
geld_clause(Consequent,Body,Preference)
```

If a GeLD clause starts with a waiting goal, then it is also written into the output file as an active item.

Macros are only used during the partial deduction step, but are never simplified during partial deduction or written to the output file. This does not only make the output file more compact, but also avoids non-termination in case of recursively defined macros.

5.4 The Deduction Engine

In this section, we discuss the implementation of the deduction systems for the various goal types. The Prolog code of the core of the deduction engine is given in appendix C.

GeLD includes the bottom-up Earley deduction described in chapter 3. In addition, it provides interpreters for a top-down Prolog proof strategy, top-down Earley deduction and for a head-driven strategy, as well the the option to pass calls directly to the underlying Prolog system for more efficient execution or for side effects.

Each goal can be annotated with one of the following goal types (in addition to the goal types **i** and **m** that are dealt with by the partial deduction system) which are described in section 5.3.1 on control information (cf. figure 5.8, page 174).

c chart-based proof (Bottom-Up Earley Deduction)

d top-down search (Prolog search strategy

e top-down Earley Deduction

h head-driven search (head-driven strategy)

p Prolog call

w waiting goal

The deduction system is completely modular so that other deduction algorithms (e.g. LR strategies) can be added as needed.

Unlike Prolog, the order of clauses in the source file does not determine the order in which clauses are executed.[9] This is due to the fact that we are do not want to provide a procedural programming language, but a deduction system that enumerates all solutions to a given query. The instrument for controlling the order of execution are the preference values given to clauses.

In Prolog clauses, the order is of course respected — their execution should (apart from the effects of partial deduction) be no different than it would be if they had been consulted directly by Prolog.

[9] In reality, the order of clauses happens to be respected with the exception that facts are always looked up first before more complex deduction procedures are attempted for proving a goal.

5.4.1 Top-down processing

D-goals are executed in the same fashion as Prolog **p**-goals: top-down, left-to-right, backtracking. The following is the inference step that is used in the processing of **d** goals. σ is the merged constraint (the unifying substitution) of B and B'.

$$\frac{\begin{array}{c} A \leftarrow B \wedge \Gamma \\ B' \leftarrow \Delta \end{array}}{\sigma(A \leftarrow \Delta \wedge \Gamma)}$$

The reason why **d**-goals are used instead of **p**-goals is that a **d**-goal may be defined by a clause whose body contains goals of another goal type. If something is called as a **p**-goal, control is passed to Prolog entirely. An example for a predicate which should be called as a **d**-goal is the following top-level predicate `parse/0`, which reads a string to be parsed, and then uses the grammatical specification (the predicate `sentence/2`) to parse the sentence using head-driven methods.

```
parse <-
  p read_string(String),
  u sentence(String,Structure),
  p pretty_print(Structure).
```

Alternatively, this clause can be specified as follows.[10]

```
parse :-
    read_string(String),
  u sentence(String,Structure),
    pretty_print(Structure).
```

This will be compiled into the following Prolog clause (where `prove/2` is a predicate which calls the deduction system with a goal type and a goal).

```
parse :-
    read_string(String),
    prove(u,sentence(String,Structure)),
    pretty_print(Structure).
```

[10] At the present stage of the implementation, there is no absolute necessity for **d**-goals since they could always be replaced by **p**-goals. However, we keep them in the implementation to keep the system extensible.

5.4.2 Head-Driven Processing

Head-driven processing is appropriate if only one base case is sufficient to start the bottom-up process. In head-driven processing, as soon as some predicate P has been proven or put into the bottom-up process as the base case of a recursion, the system looks for a clause C which starts with a **w**-goal which unifies with P. Then the remaining goals of the clause C are proven, and the head of the clause again looks for a **w**-goal that waits for it. This is continued until a solution to the **h**-goal has been found.

The following is the inference rule that is used in head-driven processing. During the processing, A is either entered as a base case or has been derived by the inference rules, and A' is a waiting goal. σ is the merged constraint (unifying substitution) of A and A'.

$$\frac{\begin{array}{c} A \\ B \leftarrow A' \ \Gamma \end{array}}{\sigma(B \leftarrow \ \Gamma)} \tag{5.14}$$

As soon as a **h**-goal G is called, the system looks for an instance of the `lookup/3` relation, which specifies the base case with which a head-driven proof of a goal can be started. The goal B returned as the second argument the relation `lookup/3` is proven according to the goal type returned as the third argument. The goal type can be a **d**-goal, which is defined by a clause containing a **h**-goal, in which case a head-driven proof is executed for finding the base case from which the head-driven process can be started (in the case of semantic-head driven processing, this can be a non-lexical non-chain rule).

The goal B that has been looked up and proven serves as input to the bottom-up proof procedure. If B unifies with the original goal G, then the proof is finished. Otherwise, the system looks for a clause C that needs B as its first goal (a waiting goal). Then the remaining goals of C are proven according to their goal types. The consequent of the clause then serves as the next input to the bottom-up process.

Note that there is no inference rule for base case lookup because there is no inference involved in base case lookup. Base case lookup only instantiates known (or provable) facts as a in the inference rule, but it does not produce any new conclusions.

5.4.3 Chart-Based Algorithms

Two chart-based algorithms (top-down Earley deduction and bottom-up Earley deduction) are implemented in the GeLD system. These two algorithms share all the basic procedures for managing the agenda, combination of active and passive items, and for the representation of items. They only differ in their rule invocation strategy (prediction in the top-down case and lookup of a set of initial items in the bottom-up case).

In general, there can be more than one chart active at the same time. This is the case if several independent goals are proved as **c**-goals or **e**-goals. The items of the different charts are stored in different databases, so that an active item of one chart cannot be combined with a passive item of another chart.

5.4.3.1 Earley Deduction

This component implements top-down Earley deduction for logic programs. Top-down Earley deduction makes use of prediction and a subsumption check. When goals other than **e**-goals occur, these are handled by the appropriate proof procedure.

Like the bottom-up Earley deduction procedure described in chapter 3, top-down Earley deduction makes use of indexing and allows other goal types than e-goals. Therefore, in figure 5.12, we extend the algorithm given in chapter 1 (figure 1.7) to handle indexing and other goal types.

5.4.3.2 Bottom-Up Earley Deduction

This component is a faithful implementation of the bottom-up Earley deduction algorithm (figure 3.7 described in chapter 3, with best-first search based on preference values (cf. section 4.1).

For bottom-up Earley deduction, all non-unit clauses of the program which start with waiting goals are assumed to be present as active items. In order to increase efficiency, these are detected at compile time, and represented in such a way as to make optimal use of Prolog's first-argument indexing for most efficient lookup.

Bottom-up Earley deduction involves a lookup step which relies on the user-defined relation `lookup/4`.

5.4.4 Prolog Call

P goals are executed directly by Prolog. This may be done either for efficiency reasons, or to call other system components implemented in Prolog (e.g., knowledge processing), or for side effects (e.g., reading and pretty-printing as in the above example.)

5.5 Best-First Search

Best-first search based on preferences is implemented for bottom-up and top-down Earley deduction. The agenda is represented as a list of items which is ordered according to the priority of the items.

In the current stage of the implementation, the priority of each item is simply the upper bound of its preference value. If two items on the agenda have the same

```
procedure prove(Goal):
– predict(Goal)
– consume-agenda
– for any item ⟨G,I⟩
  – return mgu(Goal,G) as solution if it exists

procedure add item I to agenda
– compute the priority of I
– if there is no item I′ in the chart or the agenda such that I′ subsumes I
  then agenda := agenda ∪ {I}
  else agenda := agenda

procedure consume-agenda
– while agenda is not empty
  – remove item I with highest priority from agenda
  – add item I to chart

procedure predict G
– for all rules C = G′ ← Γ
  – if σ = mgu(G, G′) exists
    then create index I for σ(C)
    add item ⟨σ(G′ ← Γ),I⟩

procedure add item ⟨C, I1⟩ to chart
– chart := chart ∪ {⟨C, I1⟩}
– if C is a unit clause
  – for all items ⟨H ← G ∧ Ξ ∧ Ω, I2⟩
    – if I = I2 ⋆ I1 exists
      and σ = mgu(C, G) exists
      and goals Ξ are provable with solution τ
      then add item ⟨τσ(H ← Ω), I⟩ to agenda
– if C = H ← G ∧ Ξ ∧ Ω is a non-unit clause
  – for all items ⟨G′ ←, I2⟩
    – if I = I1 ⋆ I2 exists
      and σ = mgu(G, G′) exists
      and goals Ξ are provable with solution τ
      then add item ⟨τσ(H ← Ω), I⟩ to agenda
  – predict G
```

Figure 5.12: Algorithm for best-first Earley deduction with indexing and different goal types

priority, the newer one is preferred. As a consequence, depth-first search results if all items on the agenda have the same priority.

5.6 Performance of the System

Efficiency has been achieved in the implementation by following a simple principle: Let Prolog do the work (wherever possible). Since Prolog is an efficiently implemented deduction system for definite clauses, control is passed to Prolog wherever possible, and only taken away from Prolog in those cases where Prolog's proof strategy is inappropriate and leads to inefficiency (e.g. by duplicated deduction steps through backtracking) or non-termination (e.g. in cases of left-recursion). For other tasks, such as unification, the handling of coroutining, indexing of clauses, simple top-down proofs, the use of Prolog is appropriate, and this has been exploited as much as possible.

Appendix B provides details about experiments with various types of grammars and compares the performance of the different algorithms with respect to the time needed and the number of items.

It should be clear, however, that there are possibilities for further efficiency improvements. For example, information about the index of an item and about the predicate in the case of passive items, and the first waiting goal in the case of active items could be used to index the items at the Prolog level to achieve more efficient lookup, as it has been done in the Prolog-based chart parser of [Erbach, 1987]. However, since the GeLD system is an experimental system for evaluating different deduction strategies for natural language processing, we have not found it necessary to squeeze out the last bit of efficiency, but leave this to the construction of actual applications. It should be noted, however, that the simple design and closeness to Prolog of the GeLD system ensures reasonable efficiency.

5.7 Conclusion

While the feature term language (ProFIT) provided can be used directly by a grammar writer for the statement of linguistic knowledge, the same is probably not true of the definite clause part with control information. This part is intended to provide the programmer (algorithm developer) of NLP systems with an experimental system for trying out various combinations of different processing strategies. Once a suitable processing strategy has been found, it could be reimplemented in order to exploit special properties of the grammar to achieve more efficiency. It is up to the algorithm developer to choose a combination of control annotations that ensures termination and completeness.

For the actual writing of grammars, a neat notation for rules, principles and lexical entries should be provided which is then mapped into clauses with control

information, or for which a compilation step can be provided.

Such a translation to a grammar rule notation has been implemented for ALE grammars. An existing ALE grammar can be converted to a set of GeLD clauses, which are annotated with control information for partial deduction. The output of the partial deduction procedure can then either be translated back automatically to ALE clauses for processing with the ALE parser, or can be handled by the GeLD deduction engine.

So far, the extended constraint language and the deduction engine have not yet been combined with each other. Currently, a re-implementation of the constraint language is under way, which makes use of the attributed variable mechanism of Sicstus Prolog 3.1. On the basis of this implementation, it will not be necessary to rewrite the deduction engine to handle the constraints, but this will be done by defining hook predicates that are called whenever Prolog unifies two terms with attributed variables. Once this implementation is available, a new experiment will be carried out to evaluate the performance of the combination of the extended constraint language and tabulation techniques.

5.8 Availability

All systems described here are in the public domain, and can be obtained free of charge from the ftp server `ftp.coli.uni-sb.de`.[11] The files are kept in the following directories:

ProFIT	sorted feature system	`pub/coli/systems/profit`
CL-ONE	extended constraint language	`pub/coli/systems/cl1`
GeLD	linguistic deduction system	`pub/coli/systems/geld`

Further information about the systems is available through the World Wide Web (`http://www.coli.uni-sb.de/~erbach`).

Acknowledgements

For the implementation of finite domains, I re-used Prolog code developed by Gertjan van Noord in the PlayMoBild system.

The usability of the ProFIT language was greatly improved through the addition of pretty-printing facilities and error messages by Christian Braun, who also wrote the ProFIT-to-LaTeX converter with which the program examples in this thesis were typeset.

[11] Users without internet access should contact the author for alternative delivery arrangements.

Chapter 6

Conclusion and Future Research

In this chapter, we summarise the results of the thesis, discuss how they fit together, and indicate potential problems where further research is needed.

6.1 Summary

In the thesis, we have discussed the processing of principle-based grammars such as HPSG, with an application to best-first processing for disambiguation, selection of paraphrases and possibly also handling ill-formed input.

We have taken the view of a grammar as a definite clause program, to which program transformation and deduction techniques could be applied. In particular, we have adopted the view of constraint logic programming, which has enabled us to abstract away from the handling of constraints (which is regarded as a service provided by the constraint solver) and concentrate on resolution strategies.

The contributions of the thesis are the following:

- A constraint language supporting sorted feature terms, Prolog terms, multi-dimensional inheritance, finite domains, by compilation to Prolog terms. The constraint language has been extended with external constraint solvers for set constraints, LP constraints, guarded constraints.
- A partial deduction system for compiling a principle-based grammar into a set of grammar rules. Partial deduction can be applied selectively through control information in the program, which makes it possible to do experimental work in order to determine the selection of goals to which partial deduction is best applied in order to bring the greatest performance improvement.

- Bottom-Up Earley deduction as a deduction system which generalises bottom-up chart parsing to a wider class of grammars/programs than just those with a context-free backbone. Bottom-up Earley deduction is better suited for handling discontinuous constituency than its top-down counterpart, it allows best-first search based on bottom-up information (e.g. tag probabilities), and provides different indexing schemes for different modes of combination of items. The correctness, completeness and termination properties of the algorithm have been shown.
- A generalised linguistic deduction system, which allows combination of different deduction strategies via control annotations, and which is tightly integrated with the underlying programming language in order to achieve efficiency.
- A fully incremental algorithm for bottom-up Earley deduction has been specified, which can cope efficiently with a change in the query by re-using intermediate deduction results as much as possible.
- Augmentation of a definite clause language with preference values. We have discussed how preference values from a variety of sources can be combined.

Figure 6.1 shows how the different techniques developed in the thesis fit together. A check mark ($\surd$) indicates that they can be combined without problems, a question mark indicates that problems can arise, and further research is needed, a cross ($\times$) indicates that they are incompatible. The numbers in parentheses refer to the comments given below.

Comments on Integration

1. **Feature Terms / Constraints:** The addition of constraints to feature terms has been problematic on the basis of older Prolog systems, but is no problem with newer CLP-approaches such as attributed variables that allow user-defined semantic unification.
2. **Partial Deduction / Constraints:** The interaction of partial deduction and constraint handling is no problem, and partial deduction can lead to a simplification of the constraint set associated with a clause or to the replacement of a goal by a set of constraints.
3. **Feature Terms / Preferences:** There is no interaction between terms and preferences because preferences are attached to clauses (lexical entries, grammar rules, principles), and not to components of terms.
4. **Constraints / Preferences:** There is a potential conflict here between eager evaluation strategies based on the most preferred solution to a constraint and a strategy in which constraints are delayed until they can be

	Feature Terms	Constraints	Partial Deduction	Preferences
Constraints	√ (1)			
Partial Deduction	√	√ (2)		
Preferences	√ (3)	? (4)	√ (5)	
Earley Deduction	√	√ (6)	√ (7)	√ (8)

Figure 6.1: Compatibility of the techniques

solved deterministically. More research is needed in order to handle this kind of interaction.

5. **Partial Deduction / Preferences:** The use of preferences and partial deduction can be combined with each other. The result of applying partial deduction to a program with preference values is another program in which the preference formulae associated with the consequents of clauses are more instantiated than in the original program, i.e., they specify a narrower preference interval.

6. **Earley Deduction / Constraints:** The combination of Earley deduction and constraints is theoretically no problem in a CLP model, but with some implementation strategies, efficiency problems may arise through the copying of constrained terms when items are stored.

7. **Earley Deduction / Partial Deduction:** Earley Deduction and Partial Deduction can well be combined with each other, but additional program transformation techniques such as LR methods or EBL may further improve the efficiency.

8. **Earley Deduction / Preferences:** Earley Deduction and preference-driven processing can be ideally combined with each other by implementing best-first search through the use of an agenda.

6.2 Future Research

This section outlines some directions for future research.

Combining top-down and bottom-up. One problem with the bottom-up Earley deduction algorithm is that all non-unit clauses of the program are added to the chart. The addition of non-unit clauses should be made dependent on the goal and the base cases in order to go from a purely bottom-up algorithm to a directed algorithm that combines the advantages of top-down and bottom-up processing. It has been repeatedly noted [Kay, 1980; Wirén, 1987; Bouma and van Noord, 1993; Bouma, 1994] that directed methods are more efficient than pure top-down or bottom-up methods. However, it is not clear how well the directed methods are applicable to grammars which do not depend on concatenation and have no unique 'left corner' which should be connected to the start symbol.

Specification of control information. At the current state of the implementation, the specification of the lookup relation, and the choice of an indexing scheme must be done individually for each grammar either by the grammar writer or by the linguistic deduction expert. It would be desirable to derive this information automatically from a given grammar for both the parsing and the generation direction. This is a question of future research, but in the near future there is no prospect of a system in which the grammar developer is freed from worrying about control issues (cf. [Seiffert, 1993]).

Parallel Algorithms. The algorithms in this thesis have been formulated as sequential operations. There are various ways to turn them into parallel algorithms, by making use of the work in logic programming on and-parallelism for conjunctions and or-parallelism for disjunctions. This question deserves further research.

Mathematical and empirical foundations of preference values. More theoretical and empirical work is required in order to arrive at a satisfactory foundation of preference values. The exact relationship between preference values and probabilities must be clarified, and methods for obtaining preference values from observable data (such as corpora) must be developed.

Ill-formed Input. The framework developed in this thesis appears suitable for handling ill-formed input because relaxed versions of rules/principles can be added incrementally to the chart, because preference values can be used to model degrees of grammaticality [Erbach, 1993a], and because preference-driven linguistic deduction can be used to control the search and avoid an explosion of the search space that would result from the unrestricted introduction of hypotheses about possible

ill-formedness. However, there are still a number of problems to be solved. One of them is the fact that ill-formed input often involves a unification failure. In order to handle unification failures, it is necessary to develop a model that can localise the source of the unification failure, and determine how "serious" the mismatch is that caused the unification failure. Another problem is that handling ill-formed input by introducing relaxed versions of principles introduces a lot of disjunction into the grammar, which makes the application of partial deduction difficult since it would lead to very large number of rules, each of which accounts for a particular kind of ill-formedness.

Appendix A

Programs with Control Information

A.1 Partial deduction for DCG

This section shows the program from page 47 with appropriately specified control information in order to turn it into the program shown on page 48.

```
% The top-down parser
parse(Cat,String) <-
  m  word_list(Word,String),
  m  word(Cat,Word).
parse(Cat,String) <-
  m  rule(Cat,RHS),
  m  parse_rhs(RHS,String).

parse_rhs([Cat1|Cats],String) <-
  m  concat(Prefix,Rest,String),
  d  parse(Cat1,Prefix),
  m  parse_rhs(Cats,Rest).
parse_rhs([],Elist]) <-
  m  empty_list(Elist).

% Auxiliary predicates
concat(A-B,B-C,A-C).
word_list(Word,[Word|R]-R).
empty_list(A-A).

% The grammar
rule(s,[np(Num,Pers),vp(Num,Pers)]).
```

```
rule(np(Num,3),[det(Num),n(Num)]).
rule(np(Num,Pers),[pron(Num,Pers)]).
rule(vp(Num,Pers),[vi(Num,Pers)]).
rule(vp(Num,Pers),[vt(Num,Pers),np(_,_)]).

% The lexicon
word(pron(sg,third),she).
word(pron(pl,first),we).
word(det(sg),a).
word(det(_),the).
word(n(sg),house).
word(n(pl),carpenters).
word(vi(sg,third),burns).
word(vt(pl,_),build).
```

A.2 Partial deduction for GB

This section shows the source code of the GB grammar from section 2.2.2 that formed the input to the partial deduction procedure. Simple disjunction of bar-levels is encoded by means of finite domains. The resulting parser can be called by the procedure `rec/2`.

```
:- goal_type('='/2,i).
:- goal_type('dif'/2,i).
:- goal_type(terminal/1,m).
:- goal_type(non_terminal/1,m).

top > [-node].
node intro [cat,bar,string].
bar fin_dom [0,1,2].

non_terminal(n).
non_terminal(d).
non_terminal(v).

terminal(the).
terminal(film).
terminal(saw).

branch([cat!M&string!SM,cat!L&string!SL,cat!R&string!SR]) <-
     non_terminal(M),
     non_terminal(L),
     non_terminal(R),
   m concat(SL,SR,SM).
branch([cat!M&string!S,D]) <-
```

```
     non_terminal(M),
     terminal(D),
   m word_to_list(D,S).

proper_branch(X) <-
   m  branch(X),
   m  conditions(X).

conditions(X) <-
   m  x_bar_theory(X),
   m  case_filter(X),
   m  theta_criterion(X).

x_bar_theory([cat!X&bar!2@bar,cat!Y&bar!2@bar,cat!Z&bar!K]) <-
     K = 0 or 1,
     X = Z,
   m is_spec_of(Y,X).
x_bar_theory([cat!X&bar!I,cat!Y&bar!0@bar,cat!Z&bar!2@bar]) <-
     I = 1 or 2,
     X = Y.
x_bar_theory([cat!Cat&bar!Bar,Word]) <-
   m cat(Word,Cat),
     Bar = 0 or 1 or 2.

case_filter([_,X,cat!n&bar!2@bar]) <-
   m is_a_case_assigner(X).
case_filter([_,X,cat!n&bar!(0 or 1)]).
case_filter([_,X,cat!Cat]) <-
     dif(Cat,n).
case_filter([_,_]).

theta_criterion([_,X&bar!0@bar,Y&bar!2@bar]) <-
  m  theta_marks(X,Y).
theta_criterion([_,bar!(1 or 2),bar!(0 or 1)]).
theta_criterion([_,_]).

cat(the,d).
cat(film,n).
cat(saw,v).
is_a_case_assigner(cat!v).
theta_marks(cat!v&bar!0@bar, cat!n&bar!2@bar).
is_spec_of(d,n).

concat(A-B,B-C,A-C).
word_to_list(Word,[Word|R]-R).
```

```
rec(Node) :-
     m  proper_branch([Node,_]).
rec(Node) :-
     m  proper_branch([Node,
                       LeftDtr,
                       RightDtr]),
        rec(LeftDtr),
        rec(RightDtr).
```

Appendix B

Performance of the Algorithms

This appendix contains some runtimes for different parsing, generation and logic programming tasks. The purpose of these runtimes is to evaluate the effect of different compilation and partial deduction strategies, and the performance of different deduction algorithms and indexing schemes for different kinds of grammars. The following questions were addressed by the experiments:

- How big is the efficiency advantage of compiling feature terms into Prolog terms, compared to unification algorithms implemented on top of Prolog, such as ALE?
- What performance benefits can be achieved by partial deduction techniques?
- How does the performance of bottom-up Earley deduction compare to its top-down counterpart, to other deduction strategies, and to dedicated, specialised parsers?

The figures given here can only give a rough indication of the actual performance, which may differ for specific grammars, and for particular choices of the implementation strategy.

B.1 Compilation to Prolog Terms

In order to evaluate the performance gain achieved by compilation of sorted feature terms to Prolog terms, we have performed experiments with the following tasks, which occur as steps in NLP algorithms.

1. unification of (unsorted) feature structures,
2. unification of inconsistent feature structures (unification failure),
3. unification of sorts,
4. lookup of one of 10000 feature structures (e.g. lexical items),
5. parsing with an HPSG grammar to provide a mix of the above tasks.

Figure B.1 gives an overview of the results.[1] The time needed by the ProFIT system is always given as 1.

Problem	ProFIT	ALE	Eisele-Dörre algorithm
1. Unification of feature structures	1		3-5
2. Unification failure	1		5
3. Unification of sorts	1		—
4. Lookup of lexical entry	1		13
5. HPSG parsing	1	5-10	—

Figure B.1: Comparison of performance between ProFIT, ALE, and the Eisele-Dörre algorithm

We have performed an experiment in which we compared the runtime of the ALE grammar processed by ALE (with sorted feature term unification implemented on top of Prolog), and by ProFIT. For ProFIT, we have used the same parsing technology as ALE, namely a bottom-up chart parser, which scans the string from right to left, and hence does not make use of active items. The results are summarised in figure B.2. Time 1 is the time needed by ALE, Time 2 is the time needed by the ProFIT implementation, and the last column gives time needed by ProFIT as a percentage of the time needed by ALE.[2]

From these experiments, we can conclude that compilation of feature structures to Prolog terms brings substantial performance benefits, while the implementation of sorted feature unification on top of Prolog is prohibitively slow.

The alternative to a compilation to Prolog terms is a lower-level implementation of feature unification algorithms, such as the new implementation of the XEROX LFG system, in which a unification algorithm with disjunction has been implemented in C. The current efforts to re-implement ALE with an abstract machine approach go in the same direction.

[1] Note that these are only rough results which may differ depending on the particular sort hierarchy that is used, and on the problem instances.

[2] The entry in the last row of the last column is the average of the percentages, rather than the percentage of the averages. The possessive suffix s counts as one word.

Input string	Time 1 (ms)		Time 2 (ms)		% diff
	total	ms/word	total	ms/word	
Kim persuades every person to walk	13135	2189	559	93	4.25
that Kim walks bothers Sandy	8081	1616	218	44	2.69
Kim can see every red red red book	19241	2405	653	82	3.30
Kim walks	2102	1050	56	28	2.66
Kim tries to persuade Sandy to walk	10258	1465	398	57	3.87
Kim tries to persuade Sandy to try to persuade Kim to try to walk	26558	1897	1415	101	5.32
that Kim is persuaded to believe Sandy to walk bothers every person	33426	1936	1149	96	3.43
Kim can walk	4283	1428	124	41	2.89
Kim is believed to see Sandy	11406	1901	261	44	2.28
that Kim tries to persuade Sandy to try to persuade Kim to try to walk bothers every person's red book	173823	8277	7965	379	4.58
Kim gives Sandy the red book	12151	2025	263	44	2.16
every person's red book	5984	1197	263	53	4.39
Average	26704	2281	1110	88	3.49

Figure B.2: Comparison of runtimes for ALE grammar with ALE and ProFIT compilation

B.2 Partial Deduction

In order to evaluate the effect of partial deduction, we have performed several experiments with the HPSG grammar for ALE written by Gerald Penn. We have measured the runtimes for the following partial deduction setups:

1. No partial deduction. The original grammar has 9 rules, 14 daughters, 2 indefinite sequences of daughters, and 85 goals (9.4 per rule).

2. All deterministic calls to HPSG principles are expanded (i.e., head feature principle, spec principle, and marking principle). The resulting grammar has 9 rules, 14 daughters, 2 indefinite sequences of daughters, and 50 goals (5.6 per rule).

3. All deterministic calls to HPSG principles are expanded, (i.e. head feature principle, spec principle, and marking principle) *and* calls to `synsems_to_non_words` and `cats` are expanded up to list length 3. The resulting grammar has 13 rules, 30 daughters, and 119 goals (9.1 per rule).

4. The subcat principle was expanded by partial deduction, in addition to the deterministic principles. This reduced the number of goals to 108 (8.3 per rule).

5. In addition to the above, the clausal rel prohibition was expanded by partial deduction. The resulting grammar has 69 rules, 157 daughters, and 574 goals (7.1 per rule).

The runtimes obtained in the partial deduction experiments can be summarised as follows:

- Partial deduction applied to deterministic principles did not bring any noticeable speedups. This is probably due to the fact the Prolog is very good at handling deterministic procedures efficiently.

- Likewise, it did not make any difference whether the sequence of goals (COMP-DTRS) was expanded by partial deduction, as long as the subcat principle was not expanded.

- If partial deduction was applied to a very large extent, e.g., by expansion of the clausal rel prohibition, which resulted in 69 rules, which showed uninteresting variation, the runtime more than doubled.

- When partial deduction was applied to the COMP-DTRS and to the Subcat principle, the runtime could be reduced to less than 50 percent.

Figure B.3 gives the runtimes for the grammar without partial deduction (case 1) and for the grammar in which partial deduction had been applied to all deterministic principles, the sequence of head daughters, and to the Subcat principle (case 4). The runtimes given are for the enumeration of all analyses. Time 1 is the time needed with the original ALE grammar, Time 2 is the time needed with the compiled grammar after the application of partial deduction and '% diff' is Time 2 expressed as a percentage relative to Time 1 (which is 100 per cent). All times are given in ms.[3]

Input string	Time 1 (ms)		Time 2 (ms)		% diff
	total	ms/word	total	ms/word	
Kim persuades every person to walk	1046	174	559	93	53.4
that Kim walks bothers Sandy	501	100	218	44	43.5
Kim can see every red red red book	1518	190	653	82	43.0
Kim walks	83	42	56	28	67.5
Kim tries to persuade Sandy to walk	898	128	398	57	44.3
Kim tries to persuade Sandy to try to persuade Kim to try to walk	3547	253	1415	101	39.9
that Kim is persuaded to believe Sandy to walk bothers every person	3880	323	1149	96	29.6
Kim can walk	232	77	124	41	53.4
Kim is believed to see Sandy	510	85	261	44	51.2
that Kim tries to persuade Sandy to try to persuade Kim to try to walk bothers every person's red book	33503	1595	7965	379	23.8
Kim gives Sandy the red book	484	81	263	44	54.3
every person's red book	493	99	263	53	53.3
Average	3891	491	1110	140	46.4

Figure B.3: Comparison of runtimes for ALE grammar with and without partial deduction

[3] Like in figure B.2, the entry in the last row of the last column is the average of the percentages, rather than the percentage of the averages, in which the good performance of the very long sentence has too much of an influence.

B.3 Bottom-Up Earley Deduction

The bottom-up Earley deduction algorithm was applied to a grammar based on the ALE grammar, to which a call to the constituent order principle (making use of concatenation), and calls to `sign/1` were added. Otherwise the grammar was essentially the same as the one in which partial deduction had been applied to the deterministic principles and the subcat principle. We used indexing scheme 4 (the one which uses the string positions of a chart parser), and obtained the runtimes given in figure B.4.

Input string	Passive Items	Active Items	Total Time (in ms)
Kim persuades every person to walk	63	95	20560
Kim can see every red red red book	71	88	21650
Kim walks	17	11	1200

Figure B.4: Runtimes for bottom-up Earley deduction with indexing scheme 4

In the next experiment, we used indexing scheme 3, and replaced the two versions of schema 5, which combine an adjunct with a head to its left or to its right by one rule which is underspecified for direction. The number of items and runtimes are given in figure B.5.

Input string	Passive Items	Active Items	Total Time (in ms)
Kim persuades every person to walk	79	120	31960
Kim can see every red red red book	134	197	80080
Kim walks	17	8	970

Figure B.5: Runtimes for bottom-up Earley deduction with indexing scheme 3

It must be noted that the runtimes are much worse than those obtained with a dedicated chart parser, such as the one used in ALE. We attribute this to the fact that the ALE parser does not make use of active items. Under current Prolog implementations, the use of active items is a source of inefficiency because of the amount of copying that must be performed when an item is stored. Since active items must encode information about the consequent of a clause, *and* about the goals that must still be proven, it is generally substantially larger than a passive item. This is a strong efficiency disadvantage because most of the time in Earley

deduction is spent with the copying of constrained terms (feature structures) when items are stored in the chart. For this reason, it is advantageous to restrict the number and the size of the items stored in the chart. In practice, it has turned out that the avoiding of copying can contribute more to efficiency than the storing of partial solutions through active items.

Appendix C

Prolog Code of the Deduction Algorithms

This appendix contains listings of the Prolog code for the most important predicates used in the deduction system.

Goals of all goal types are called with the predicate `prove/3`.

```
prove(+Type,-Goal,-Pref)
```

The first argument is the goal type, the second argument the goal, and the third the preference value in case it needs to be accessed for controlling the search or returned with the result.

C.1 Prolog goals

For Prolog goals, control is entirely passed to Prolog.

```
% goal type p: Prolog goals
prove(p,Goal,_Pref) :-
        call(Goal).
```

C.2 Top-down goals

Top-down goals fall into three cases:

Goals delayed by coroutining. In this case, the corouting is handled by Prolog.

Goals provable as facts. For this case, control is passed to Prolog to permit more efficient access to clauses.

Other goals. In this case a meta-interpreter is used to select a matching clause and prove the goals in the body according to their respective goal types.

```
% goal type d: top-down goals
prove(d,when(Cond,Goal),Pref) :-
        !,
        when(Cond,prove(d,Goal,Pref)).
prove(d,Goal,_) :-                          % takes care of facts
        call(Goal).
prove(d,Goal,PrefFn) :-
        geld_clause(Goal,Subgoals,PrefFn),  % select a matching clause
        prove_goals(Subgoals,[]).           % exit without waiting goals
                                            %    for top-down processing

% prove_goals(+Goals,-WaitingGoals)
prove_goals([],[]).
prove_goals([goal(Goal,Type,PrefVal)|Goals],WaitingGoals) :-
        (  Type == w
        -> WaitingGoals = goals(Goal,Goals)
        ;  (prove(Type,Goal,PrefVal),
            prove_goals(Goals,WaitingGoals)
           )
        ).
```

C.3 Head-driven Processing

Head-driven processing consists of looking up a potential base case of a recursive definition (procedure `look/4`), and using it as input to the bottom-up step.

`init/4` is the representation of non-unit clauses starting with a waiting goal for more efficient access. A clause $[A\#Pref \leftarrow \mathbf{w}G \wedge \Omega]$ is represented as `init(G,A,Omega,Pref)`.

`look/4` is the precompiled version of the lookup relation. The fourth argument represents the index and is not needed for head-driven processing.

```
% goal type h: head-driven processing
prove(h,Goal,Pref) :-
     look(Goal,Fact,Pref0,_),
     bu_step(Fact,Pref0,Goal,Pref).

bu_step(Goal,Pref,Goal,Pref).
```

```
bu_step(Fact,_Pref0,Goal,Pref) :-
     init(Fact,IntGoal,Goals,PrefFn),
     prove_goals(Goals,[]),
     bu_step(IntGoal,PrefFn,Goal,Pref).
```

C.4 Top-down Earley Deduction

The implementation of top-down Earley deduction is straightforward. A unique identifier for each chart based proof is generated in order to allow for separate charts for separate proofs.

```
% goal type e: Earley deduction
prove(e,Goal,Pref) :-
    gensym2(td_proof,ProofID),
    recorda(proof,ProofID,_),
    predict(Goal,Goal,Pref,ProofID),
    get_agenda(ProofID,[],Agenda),
    consume_agenda(Agenda,td,ProofID,Goal,Pref).

predict(Goal,InitGoal,Pref,ProofID) :-
    call(Goal),
    add_item([],Goal,Pref,id,ProofID,InitGoal,subs_chk),
    fail.
predict(Goal,InitGoal,Pref,ProofID) :-
    geld_clause(Goal,Subgoals,Pref),
    prove_goals(Subgoals,WaitingGoals),
    add_item(WaitingGoals,Goal,Pref,id,ProofID,InitGoal,subs_chk),
    fail.
predict(_,_,_,_).
```

C.5 Bottom-Up Earley Deduction

Bottom-up Earley deduction differs from the top-down counterpart in that prediction has been replaced by lookup (`c_prove_init/2`).

```
% goal type c: bottom-up Earley deduction
prove(c,Goal,Pref) :-
```

```
    gensym2(bu_proof,ProofID),
    recorda(proof,ProofID,_),
    c_prove_init(Goal,ProofID),
    get_agenda(ProofID,[],Agenda),
    consume_agenda(Agenda,bu,ProofID,Goal,Pref).

c_prove_init(Goal,ProofID) :-
    look(Goal,Fact,Pref,Index),
    add_item([],Fact,Pref,Index,ProofID,Goal,no_subs_chk),
    fail.
c_prove_init(_,_).
```

C.6 Shared Code for Bottom-Up and Top-Down Earley Deduction

`consume_agenda/5` drives both Earley deduction processes by adding items to the chart, returning solutions, and executing tasks from the agenda. The first argument is the agenda, and the other arguments are information about direction of processing (top-down vs. bottom-up), the proof identifier, the goal in the query, and the preference value that is passed to procedures which are called by `consume_agenda/5`.

`perform_task/4` takes an item, and combines it with items in the chart by means of the completion rule, and also predicts new items in case of top-down Earley deduciton. For non-unit clauses that have been turned into active items (`init/4`), a specialised instance of the completion rule is used.

```
% consume_agenda(Agenda,Direction,ProofID,BigGoal,Pref)
% clause1: if there are any solutions to the goal, retrieve them
consume_agenda([_-Item|_],_,ProofID,_,_) :-
    recorda(ProofID,Item,_),
    fail.
consume_agenda(_,_,ProofID,Goal,Pref) :-
    recorded(ProofID,solution(Goal,Pref),Ref),
    erase(Ref).
% clause2: do the tasks on the agenda
consume_agenda([_-Task|Agenda],Direction,ProofID,Goal,Pref) :-
    perform_task(Task,Direction,Goal,ProofID),
    get_agenda(ProofID,Agenda,NewAgenda),
    !, % this cut should help Prolog detect the determinism
```

```
    consume_agenda(NewAgenda,Direction,ProofID,Goal,Pref).

% perform_task(Item,BU_or_TD,BigGoal,ProofID)
% combination of active with passive items
perform_task(act(Goal,Sub1,Subgoals,PrefFn,IndexA),_,G,Proof) :-
    pas_item(Proof,Sub1,_Pref,IndexP),
    combine_index(IndexA,IndexP,Index),
    prove_goals(Subgoals,WaitingGoals),
    add_item(WaitingGoals,Goal,PrefFn,Index,Proof,G,no_subs_chk),
    fail.
% prediction; only for top-down
perform_task(act(_Goal,Sub1,_Subgoals,_PrefFn,_IndexA),td,G,Proof) :-
    predict(Sub1,G,_Pref,Proof),
    fail.
% combination of passive with active items
perform_task(pas(Goal1,_Pref,IndexP),_,G,Proof) :-
    act_item(Proof,Pred,Goal1,Goals,Pref,IndexA),
    combine_index(IndexA,IndexP,Index),
    prove_goals(Goals,WaitingGoals),
    add_item(WaitingGoals,Pred,Pref,Index,Proof,G,no_subs_chk),
    fail.
% combination with init. active items; only for bottom-up
perform_task(pas(Goal1,_Pref,Index),bu,G,Proof) :-
    init(Goal1,Pred,Goals,Pref),
    prove_goals(Goals,WaitingGoals),
    add_item(WaitingGoals,Pred,Pref,Index,Proof,G,no_subs_chk),
    fail.
perform_task(_,_,_,_).
```

C.7 Handling of Items

`add_item/6` adds items to the agenda. If the first argument is a compound term (`goals/3`), then an active item is added; otherwise if it is the empty list, a passive item is added. Each item is assigned a priority based on the upper bound of its preference value at this step, which is used for ordering it in the agenda. If an item has no preference value, it is given the value 1.

```
% add_item(Goals,Pred,Pref,Index,+ProofID,InitGoal)
% purpose: add items to the agenda
add_item(goals(Type,Goal,Goals),Pred,Pref,Index,ProofID,InitGoal) :-
```

```
    Item = act(Pred,Goal,Goals,Pref,Index),
    prio(Pref,P),
    recorda(ProofID,agenda(P-Item),_),
    (  Type == e
    -> (restriction(Goal,RestrGoal),
        predict(RestrGoal,InitGoal,ProofID))
    ;  true
    ),
    !.
add_item([],Pred,Pref,Index,Proof,InitGoal) :-
    Item = pas(Pred,Pref,Index),
    prio(Pref,P),
    recorda(Proof,agenda(P-Item),_),
    record_solution(Pred,InitGoal,Proof,Pref),
    !.

% function for priority calculation
prio(PrefFn,P) :-
    upper_estimate(PrefFn,Up),
    P is Up,
    !.
prio(_,1).
```

Access Predicates

Items are accessed with the predicates `act_item/6`, `pas_item/4`. Initial active items (those created at compile time) are accessed with the special predicate `init/4`.

Appendix D

GeLD Interface Specification

This appendix describes the exported procedures of the GeLD system, which provide its interface to the user and to other applications.

D.1 Proving Goals

`Goal # Pref`	call the goal Goal with the preference value Pref
`prove(Goal)`	call the goal as top-down proof
`prove(Type,Goal)`	call the goal with goal type Type
`? Goal`	A goal must be prefixed by '?' if it contains any ProFIT terms.

D.2 Loading Programs

`fit(File)`	load the file as ProFIT program
`lg(File)`	load the file as GeLD program
`lpg(File)`	pass the file through ProFIT and load the compiled file as a GeLD program

D.3 Inspection of Clauses and Items

`show_clause`	shows all GeLD clauses
`show_pas`	shows all passive items in the chart and the number of passive items
`show_act`	shows all active items in the chart and the number of active items
`show_init`	shows all initial active items (GeLD clauses starting with **w**-goals.
`show_solution`	shows all solutions to the last query which are recorded in the chart and the time needed for each solution.
`stats`	show statistics about the number of active and passive items and about the runtime

Bibliography

[Abeillé, 1993] Anne Abeillé. *Les Nouvelles Syntaxes: Grammaires d'unification et analyse du français*. Armand Colin, Paris, 1993.

[Abeillé, 1994] Anne Abeillé. The flexibility of french idioms: a representation with lexicalized tags. In André Schenk and Erik-Jan van der Linden, editors, *Idioms*. Lawrence Erlbaum, 1994.

[Aït-Kaci and Podelski, 1991] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. Technical Report 11, DEC Paris Research Laboratory, Paris, June 1991.

[Aït-Kaci and Podelski, 1994] Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and System*, 16(4):1 – 40, July 1994.

[Aït-Kaci, 1991] Hassan Aït-Kaci. An overview of LIFE. In J. W. Schmidt and A. A. Stogny, editors, *Next Generation Information System Technology, Proceedings of the 1st International East/West Data Base Workshop (Kiev)*, pages 42 – 58. Springer, 1991.

[Ajdukiewicz, 1935] K. Ajdukiewicz. Die syntaktische Konnexität. *Studia Philosophica*, 1:1 – 27, 1935.

[Alshawi *et al.*, 1991] H. Alshawi, D. J. Arnold, R. Backofen, D. M. Carter, J. Lindop, K. Netter, J. Tsujii, and H. Uszkoreit. Eurotra 6/1: Rule formalism and virtual machine design study — Final report. Technical report, SRI International, Cambridge, 1991.

[Alshawi, 1991] Hiyan Alshawi, editor. *The Core Language Engine*. MIT Press, 1991.

[Bäck *et al.*, 1991] T. Bäck, F. Hoffseister, and H. Schwefel. A survey of evolution strategies. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA, 1991.

[Backofen and Smolka, 1995] Rolf Backofen and Gert Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146:243 – 268, 1995.

[Backofen *et al.*, 1995] Rolf Backofen, James Rogers, and K. Vijay-Shanker. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language and Information*, 1995. To appear.

[Balari *et al.*, 1990] Sergio Balari, Luis Damas, Nelma Moreira, and Giovanni B. Varile. CLG(n): Constraint Logic Grammars. In *13th International Conference on Computational Linguistics*, pages 7–12, 1990.

[Bancilhon and et al., 1986] F. Bancilhon and et al. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth PODS Symposium*, pages 1 – 15, 1986.

[Barnett and Mani, 1991] Jim Barnett and Inderjeet Mani. Shared preferences. In Tomek Strzalkowski, editor, *ACL SIG workshop "Reversible Grammar in Natural Language Processing"*, pages 109 – 118, Berkeley, CA, 1991.

[Barnett, 1994] Jim Barnett. Bi-directional preferences. In T. Strzalkowski, editor, *Reversible Grammar in Natural Language Processing*, pages 201–234. Kluwer, Boston, 1994.

[Beckstein and Kim, 1991] Clemens Beckstein and Michelle Kim. Generalized Earley deduction and its correctness. In Th. Christaller, editor, *GWAI 91 - 15. Fachtagung für Künstliche Intelligenz*, Bonn, FRG, 1991. Springer.

[Berwick *et al.*, 1991] R. Berwick, S. Abney, and C. Tenny. *Principle-Based Parsing*. Kluwer, Dordrecht, 1991.

[Berwick, 1991] Robert Berwick. Principles of principle-based parsing. In R. Berwick, S. Abney, and C. Tenny, editors, *Principle-Based Parsing*. Kluwer, Dordrecht, 1991.

[BIM-SEMA, 1993] BIM-SEMA. *ALEP System Documentation: The ALEP Linguistic Subsystem, Version 1.0.* Commission of the European Communities, March 1993.

[Block, 1991] Hans Ulrich Block. Compiling trace and unification grammar for parsing and generation. In Tomek Strzalkowski, editor, *ACL SIG workshop "Reversible Grammar in Natural Language Processing"*, pages 100 – 108, Berkeley, CA, 1991.

[Bobrow and Webber, 1980] Robert J. Bobrow and Bonnie Lynn Webber. Knowledge representation for syntactic/semantic processing. In *AAAI-80*, pages 316–323, 1980.

[Böttcher, 1993] Martin Böttcher. Disjunctions and inheritance in the context feature structure system. In *EACL93*, pages 54 – 60, Utrecht, NL, 1993.

[Bouma and van Noord, 1993] Gosse Bouma and Gertjan van Noord. Head-driven parsing for lexicalist grammars: Experimental results. In *EACL93*, pages 71 – 80, Utrecht, NL, 1993.

[Bouma *et al.*, 1988] Gosse Bouma, Esther König, and Hans Uszkoreit. A flexible graph-unification formalism and its application to natural-language processing. *IBM Journal of Research and Development*, 32(2):170 – 184, 1988.

[Bouma, 1994] Gosse Bouma. Prediction in chart parsing algorithms for categorial unification grammar. In *Proceedings of the 5th European Conference of the Association for Computational Linguistics*, pages 179 – 184, Berlin, 1994.

[Bowen *et al.*, 1982] David L. Bowen, Lawrence Byrd, Fernando C. N. Pereira, Luís M. Pereira, and David H. D. Warren. DECsystem-10 Prolog User's Manual. Occasional Paper 27, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, 1982.

[Brew, 1991] Chris Brew. Systemic classification and its efficiency. *Computational Linguistics*, 17(4):375 – 408, 1991.

[Brew, 1993] Chris Brew. Adding preferences to CUF. In Jochen Dörre, editor, *DYANA-2 Deliverable R1.2.A: Computational Aspects of Constraint-Based Linguistic Description I*, pages 57 – 69. Esprit Basic Research Project 6852, 1993.

[Brew, 1995] Chris Brew. Stochastic HPSG. In *Proceedings of the 7th European Conference of the Association for Computational Linguistics*, Dublin, 1995.

[Burheim, 1995] Tore Burheim. Indexed languages and unification grammars. In *10th Nordic Conference on Computational Linguistics, NoDaLiDa'95,*, Helsinki, 1995.

[Calder and Humphreys, 1993] J. Calder and K. Humphreys. Pleuk overview. Technical report, University of Edinburgh, Centre for Cognitive Science, Edinburgh, 1993.

[Calder *et al.*, 1988] Jonathan Calder, Ewan Klein, and Henk Zeevat. Unification categorial grammar: a concise, extendable grammar for natural language processing. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 83–86, 1988.

[Carpenter and Penn, 1994] Bob Carpenter and Gerald Penn. *The Attribute Logic Engine: User's Guide (Version 2.0.1).* Carnegie-Mellon University, Pittsburgh, December 1994.

[Carpenter, 1991] Bob Carpenter. The generative power of categorial grammars and head-driven phrase structure grammars with lexical rules. *Computational Linguistics*, 17(3):301 – 314, 1991.

[Carpenter, 1992] Bob Carpenter. *The logic of typed feature structures*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[Carpenter, 1993a] Bob Carpenter. *ALE Version β: User Manual*. University of Pittsburgh, 1993.

[Carpenter, 1993b] Bob Carpenter. An attribute-value logic for sets. In *Presented at the 3rd ASL/LSA Conference on Logic and Language*, Columbus, OH, 1993.

[Carpenter, 1993c] Bob Carpenter. Compiling typed attribute-value logic grammars. In *Third International Workshop on Parsing Technologies*, pages 39 – 48, Tilburg, NL and Durbuy, B, 1993.

[Chanod *et al.*, 1994] Jean-Pierre Chanod, Simonetta Montemagni, and Frédérique Segond. Dynamic relaxation: Measuring the distance from text to grammar. In Carlos Martín-Vide, editor, *Current Issues in Mathematical Linguistics*, pages 373 – 379. Elsevier, 1994.

[Chomsky, 1981] Noam Chomsky. *Lectures on Government and Binding*. Studies in Generative Grammar. Foris, Dordrecht, 1981.

[Chomsky, 1986] Noam Chomsky. *Barriers*. Linguistic Inquiry Monographs. MIT Press, Cambridge, MA, 1986.

[Chomsky, 1993] N. Chomsky. A minimalist program for linguistic theory. In K. Hale and S.J. Keyser, editors, *The view from building 20. Essays in linguistics in honor of Sylvain Bromberger*. MIT Press, 1993.

[Clocksin and Mellish, 1981] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, Berlin, 1981. (2nd edition 1984, 3rd edition 1987).

[Colmerauer, 1970] Alain Colmerauer. Les Systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Technical Report 43, Département d'Informatique, Université de Montreal, Canada, 1970.

[Colmerauer, 1982] Alain Colmerauer. Prolog II: Manuel de réference et modèle théorique. Technical report, Groupe d'Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, 1982.

[Colmerauer, 1987] Alain Colmerauer. Opening the Prolog III universe. *Byte*, August 1987.

[Cottrell, 1987] Garrison Cottrell. A connectionist model of lexical access of ambiguous words. In Garrison Cottrell Steven Small and Michael Tanenhaus, editors, *Lexical Ambiguity Resolution*. Morgan Kaufman, Los Altos, 1987.

[Covington, 1989] Michael Covington. GULP 2.0: an extension of Prolog for unification-based grammar. Technical Report AI-1989-01, Advanced Computational Methods Center, University of Georgia, 1989.

[Crocker and Lewin, 1992] Matthew W. Crocker and Ian Lewin. Parsing as deduction: rules versus principles. In Bernd Neumann, editor, *Tenth European Conference on Artificial Intelligence*, pages 508 – 512, Vienna, 1992. Wiley.

[Crouch, 1994] Richard Crouch. A prototype ALEP constraint solver. Technical report, SRI International, Cambridge Research Centre, February 1994.

[Damas and Varile, 1992] Luis Damas and Giovanni B. Varile. On the satisfiability of complex constraints. In *Proceedings of the 14th International Conference on Computational Linguistics*, Nantes, 1992.

[Damas *et al.*, 1991] Luis Damas, Nelma Moreira, and Giovanni B. Varile. The formal processing models of CLG. In *Proceedings of the 5th European Conference of the Association for Computational Linguistics*, Berlin, 1991.

[de Kleer, 1986] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[Den, 1994] Yasuharu Den. Generalized chart algorithm: An efficient procedure for cost-based abduction. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 218 – 225, Las Cruces, NM, 1994.

[Deransart and Małuszyński, 1993] Pierre Deransart and Jan Małuszyński. *A Grammatical View of Logic Programming*. MIT Press, Cambridge, 1993.

[Dörre and Dorna, 1993] Jochen Dörre and Michael Dorna. CUF – A formalism for linguistic knowledge representation. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description. Deliverable R1.2.A.* DYANA-2 – ESPRIT Basic Research Project 6852, 1993.

[Dörre and Eisele, 1991] Jochen Dörre and Andreas Eisele. A comprehensive unification-based grammar formalism. Technical report, DYANA-ESPRIT Basic Research Action BR3175, 1991.

[Dörre and Seiffert, 1991] Jochen Dörre and Roland Seiffert. Sorted feature terms and relational dependencies. IWBS Report 153, IBM Germany, February 1991.

[Dörre, 1993] Jochen Dörre. Generalizing Earley deduction for constraint-based grammars. In Jochen Dörre, editor, *DYANA-2 Deliverable R1.2.A: Computational Aspects of Constraint-Based Linguistic Description I*, pages 23 – 41. Esprit Basic Research Project 6852, 1993.

[Dowty, 1993] David Dowty. Towards a minimalist theory of syntactic structure. In Wietske Sijtsma and Arthur van Horck, editors, *Discontinuous Constituency*. Mouton de Gruyter, Berlin, 1993.

[Dymetman *et al.*, 1990a] M. Dymetman, P. Isabelle, and F. Perrault. A symmetrical approach to parsing and generation. In *13th International Conference on Computational Linguistics*, pages 90 – 96, Helsinki, 1990.

[Dymetman *et al.*, 1990b] Marc Dymetman, Pierre Isabelle, and Francois Perrault. A symmetrical approach to parsing and generation. In *Proceedings of the 13th International Conference on Computational Linguistics*, Helsinki, 1990.

[Earley, 1970] Jay Earley. An efficient context-free parsing algorithm. *CACM*, 6(8):451–455, 1970.

[Eisele and Dörre, 1990] Andreas Eisele and Jochen Dörre. Disjunctive unification. IWBS Report 124, IBM Germany, Stuttgart, 1990.

[Eisele, 1994] Andreas Eisele. Towards probabilistic extensions of constraint-based grammars. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Descriptions II*, pages 1 – 21. ESPRIT Basic Research Project 6852 DYANA-2, Deliverable R1.2.B, 1994.

[Emele and Zajac, 1990] Martin Emele and Rémi Zajac. Typed unification grammars. In *coling90*, Helsinki, 1990.

[Engelkamp *et al.*, 1992] Judith Engelkamp, Gregor Erbach, and Hans Uszkoreit. Handling linear precedence constraints by unification. In *30th Annual Meeting of the Association for Computational Linguistics*, pages 201 – 208, Newark, Delaware, 1992.

[Erbach and Krenn, 1994] Gregor Erbach and Brigitte Krenn. Idioms and support verb constructions. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar*. CSLI, Stanford, CA, 1994.

[Erbach and Uszkoreit, 1995] Gregor Erbach and Hans Uszkoreit. Linear precedence constraints in "lean" formalisms. CLAUS Report 55, Universität des Saarlandes, Saarbrücken, April 1995.

[Erbach *et al.*, 1994a] G. Erbach, M. van der Kraan, S. Manandhar, D. Moshier, H. Ruessink, and C. Thiersch. *The Reusability of Grammatical Resources. Deliverable C:*. Commission of the European Communities, Edinburgh, Saarbrücken, Tilburg, Utrecht, 1994.

[Erbach *et al.*, 1994b] G. Erbach, M. van der Kraan, S. Manandhar, D. Moshier, H. Ruessink, and C. Thiersch. Reusability of grammatical resources. *ELSNews*, 3(2):5–7, 1994.

[Erbach *et al.*, 1995a] G. Erbach, M. van der Kraan, S. Manandhar, D. Moshier, H. Ruessink, and C. Thiersch. *The Reusability of Grammatical Resources. Deliverable D:.* Commission of the European Communities, Edinburgh, Saarbrücken, Tilburg, Utrecht, 1995.

[Erbach *et al.*, 1995b] G. Erbach, M. van der Kraan, S. Manandhar, H. Ruessink, W. Skut, and C. Thiersch. Extending unification formalisms. In *2nd Language Engineering Convention*, London, 1995.

[Erbach *et al.*, 1995c] Gregor Erbach, Suresh Manandhar, and Wojciech Skut. CL-ONE user's manual. in preparation, 1995.

[Erbach, 1987] Gregor Erbach. An efficient chart parser using different strategies. Discussion Paper 52, University of Edinburgh, Department of Artificial Intelligence, Edinburgh, August 1987.

[Erbach, 1991a] Gregor Erbach. An environment for experimenting with parsing strategies. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 931 – 936, Sydney, 1991. Morgan Kaufmann.

[Erbach, 1991b] Gregor Erbach. A flexible parser for a linguistic development environment. In O. Herzog and C.-R. Rollinger, editors, *Natural language understanding in LILOG*. Springer, Berlin, 1991.

[Erbach, 1993a] Gregor Erbach. Towards a theory of degrees of grammaticality. CLAUS-Report 34, Universität des Saarlandes, Saarbrücken, 1993.

[Erbach, 1993b] Gregor Erbach. Using preference values in typed feature structures to exploit non-absolute constraints for disambiguation. In Harald Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 173 – 185. Ellis-Horwood, 1993.

[Erbach, 1994a] Gregor Erbach. Bottom-up Earley Deduction. In *COLING*, pages 796 – 802, Kyoto, 1994.

[Erbach, 1994b] Gregor Erbach. Multi-dimensional inheritance. In H. Trost, editor, *Proceedings of KONVENS '94*, pages 102 – 111, Vienna, 1994. Springer.

[Erbach, 1994c] Gregor Erbach. ProFIT - Prolog with Features, Inheritance, and Templates. CLAUS Report 42, Universität des Saarlandes, Saarbrücken, July 1994.

[Erbach, 1995] Gregor Erbach. ProFIT: Prolog with features, inheritance and templates. In *Seventh Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, Dublin, 1995.

[Finkler and Neumann, 1989] Wolfgang Finkler and Günter Neumann. POPEL-HOW: A distributed parallel model for incremental natural language production with feedback. In *11th International Joint Conference on Artificial Intelligence*, pages 1518–1523, Detroit, MI, 1989.

[Fitting, 1990] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, New York, 1990.

[Fodor and Frazier, 1970] J. Fodor and L. Frazier. The Sausage Machine: A new two-stage parsing model. *Cognition*, 6, 1970.

[Fong, 1991] S. Fong. The computational implementation of principle-based parsers. In R. Berwick, S. Abney, and C. Tenny, editors, *Principle-Based Parsing*. Kluwer, Dordrecht, 1991.

[Frisch, 1993] Alan M. Frisch. Feature-based grammars as constraint grammars. In J. Cole, G. Green, and J. Morgan, editors, *Computational linguistics and the foundations of linguistic theory*. 1993.

[Fujisaki *et al.*, 1991] T. Fujisaki, F. Jelinek, J. Cocke, E. Black, and T. Nishino. A probabilistic parsing method for sentence disambiguation. In M. Tomita, editor, *Current issues in parsing technology*, pages 139–152. Kluwer, Boston, 1991.

[Garside and Leech, 1987] Roger Garside and Fanny Leech. The UCREL probabilistic parsing system. In G. Leech R. Garside and G. Sampson, editors, *The Computational Analysis of English: a corpus-based approach*, pages 66–81. Longman, London, 1987.

[Gazdar *et al.*, 1985] Gerald Gazdar, Ewan Klein, Geoffrey K. Pullum, and Ivan A. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford, 1985.

[Gerdemann, 1991] Dale Douglas Gerdemann. *Parsing and Generation of Unification Grammars*. PhD thesis, University of Illinois at Urbana-Champaign, 1991. Cognitive Science technical report CS-91-06 (Language Series).

[Giannesini *et al.*, 1985] F. Giannesini, H. Kanoui, R. Pasero, and M. van Caneghem. *Prolog*. InterÉditions, Paris, 1985.

[Goldberg, 1989] David E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley, Reading, MA, 1989.

[Götz and Meurers, 1995] Thilo Götz and Walt Detmar Meurers. Compiling HPSG type constraints into definite clause programs. In *ACL 95*, Cambridge, MA, 1995.

[Haddock, 1987] Nicholas J. Haddock. Incremental interpretation and combinatory categorial grammar. *IJCAI-87*, pages 661–663, 1987.

[Harbusch *et al.*, 1991] Karin Harbusch, Wolfgang Finkler, and Anne Schauder. Incremental syntax generation with tree adjoining grammars. Technical Report RR-91-25, DFKI, Saarbrücken, 1991.

[Hashida, 1994] Kôiti Hashida. Emergent parsing and generation with generalized chart. In *COLING 94*, pages 468 – 474, Kyoto, Japan, 1994.

[Haugeneder and Gehrke, 1986] Hans Haugeneder and Manfred Gehrke. A user friendly ATN programming environment (APE). In *COLING-86*, pages 399–401, 1986.

[Haviland, 1979] J. Haviland. Guugu Yimidhirr. In R. Dixon and B. Blake, editors, *Handbook of Australian Languages*. Benjamins, Amsterdam, 1979.

[Hawkins, 1990] John A. Hawkins. A parsing theory of word order universals. *Linguistic Inquiry*, 21(2):223 – 261, 1990.

[Hawkins, 1994] John A. Hawkins. *A Performance Theory of Order and Constituency*. Cambridge University Press, Cambridge, 1994.

[Hirsh, 1986] Susan Beth Hirsh. P-PATR: A compiler for unification-based grammars. Master's thesis, Stanford University, Stanford, CA, December 1986.

[Hobbs *et al.*, 1993] J. R. Hobbs, M. E. Stickel, D. E. Appelt, and P. Martin. Interpretation as abduction. *Artificial Intelligence*, 63:69–142, 1993.

[Höhfeld and Smolka, 1988] Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG Report 53, IBM Deutschland, Stuttgart, October 1988.

[Holland, 1975] John H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[Holzbaur, 1992] C. Holzbaur. DMCAI CLP reference manual. Technical Report TR-92-24, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, 1992.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Series in Computer Science. Addison Wesley, Reading, MA, 1979.

[Huckle, 1995] Christopher C. Huckle. Grouping words using statistical context. In *EACL 95 Student Session*, 1995.

[Jaffar and Lassez, 1987] J. Jaffar and J. L. Lassez. Constraint logic programming. In 14th *ACM Principles of Programming Languages Conference*, Munich, 1987.

[Jensen *et al.*, 1992] K. Jensen, G. Heidorn, and S. Richardson. *Natural Language Processing: the PLNLP Approach*. Kluwer, Dordrecht, 1992.

[Johnson and Dörre, 1995] Mark Johnson and Jochen Dörre. Memoization of coroutined constraints. In *Proceedings of the 32rd Annual Meeting of the Association for Computational Linguistics*, Cambridge, 1995. (Computation and Language e-print archive cmp-lg/9504028).

[Johnson and Kay, 1994] Mark Johnson and Martin Kay. Parsing and empty nodes. *Computational Linguistics*, 20(2):289 – 300, 1994.

[Johnson, 1988] Mark Johnson. *Attribute-Value Logic and the Theory of Grammar*. CSLI Lecture Notes 14. Center for the Study of Language and Information, Stanford, CA, 1988.

[Johnson, 1991] Mark Johnson. Features and formulae. *Computational Linguistics*, 17(2):131 – 152, 1991.

[Johnson, 1993] Mark Johnson. Memoization in constraint logic programming. In *Proceedings of the International Conference on Constraint Programming*, Newport, Rhode Island, 1993.

[Johnson, in press] Mark Johnson. Memoization of top down parsing. *Computational Linguistics*, in press. cmp-lg/9504016.

[Joshi and Vijay-Shanker, 1985] Aravind K. Joshi and K. Vijay-Shanker. Some computational properties of tree adjoining grammars. In *ACL Proceedings, 23rd Annual Meeting*, pages 82–93, 1985.

[Joshi *et al.*, 1975] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *JCSS*, 10(1):136–163, 1975.

[Kaplan and Bresnan, 1982] Ronald Kaplan and Joan Bresnan. Lexical Functional Grammar: a formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, chapter 4, pages 173 – 281. MIT Press, Cambridge, MA, 1982.

[Kaplan, 1973] Ronald M. Kaplan. A general syntactic processor. In Randall Rustin, editor, *Natural Language Processing*, pages 193 – 241. Algorithmics Press, New York, 1973.

[Kasper and Rounds, 1986] Robert Kasper and William Rounds. A logical semantics for feature structures. In *ACL Proceedings, 24th Annual Meeting*, pages 257–266, 1986.

[Kasper *et al.*, 1995] Robert Kasper, Bernd Kiefer, Klaus Netter, and Krishnamurti Vijay-Shanker. Compilation of HPSG to TAG. In *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 92–99, Cambridge, MA., 1995.

[Kay, 1967] Martin Kay. Experiments with a powerful parser. In *Second International Conference on Computational Linguistics*, 1967.

[Kay, 1979] Martin Kay. Functional grammar. In *Fifth Annual Meeting of the Berkeley Linguistics Society*, Berkeley, CA, 1979. Berkeley Linguistics Society.

[Kay, 1980] Martin Kay. Algorithm schemata and data structures in syntactic processing. Technical report, Xerox PARC, Palo Alto, CA, October, 1980 1980.

[Kay, 1984] Martin Kay. Functional unification grammar: A formalism for machine translation. In *10th International Conference on Computational Linguistics*, pages 75–78, Stanford, CA, 1984.

[Kay, 1985] Martin Kay. Parsing in functional unification grammar. In Lauri Karttunen and Arnold M. Zwicky David R. Dowty, editors, *Natural Language Parsing*, pages 251–278. Cambridge University Press, Cambridge, 1985.

[Kay, 1993] Martin Kay. Chart-based generation. Lecture given at the German Research Center for Artificial Intelligence (DFKI), 1993.

[Kempen and Vosse, 1989] Gerard Kempen and Theo Vosse. Incremental syntactic tree formation in human sentence processing: a cognitive architecture based on activation decay and simulated annealing. *Connection Science*, 1(3):273 – 289, 1989.

[Kim, 1994] Albert Kim. Graded unification: A framework for interactive processing. In *ACL-94 Student Session*, 1994.

[Konieczny *et al.*, 1991] Lars Konieczny, Barbara Hemforth, and Gerhard Strube. Psychologisch fundierte Prinzipien der Satzverarbeitung jenseits von Minimal Attachment. *Kognitionswissenschaft*, (1):58 – 70, 1991.

[König, 1990] Esther König. *Der Lambek-Kalkül. Eine Logik für lexikalische Grammatiken.* PhD thesis, Universität Stuttgart, 1990.

[Kowalski, 1979] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22:424 – 436, 1979.

[Krieger and Schäfer, 1994] Hans-Ulrich Krieger and Ulrich Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics*, Kyoto, 1994.

[Lehner, 1993] Christoph Lehner. *Grammatikentwicklung mit Constraint-Logikprogrammierung*. Dissertationen zur Künstlichen Intelligenz. INFIX, Sankt Augustin, 1993.

[Lehner, 1994] Christoph Lehner. Modellierung von lexikalischer Multifunktionalität mit Constraint-Logikprogrammierung. In Harald Trost, editor, *KONVENS '94*, pages 200 – 209, Vienna, 1994.

[Lenerz, 1977] Jürgen Lenerz. *Zur Abfolge nominaler Satzglieder im Deutschen*, volume 5 of *Studien zur deutschen Grammatik*. Narr, Tübingen, 1977.

[Lloyd, 1984] John Wiley Lloyd. *Foundations of Logic Programming*. Symbolic Computation Series. Springer, Berlin, 1984. (2nd edition, 1987).

[Manandhar, 1993] Suresh Manandhar. *Relational Extensions to Feature Logic: Applications to Constraint Based Grammars*. PhD thesis, University of Edinburgh, 1993.

[Manandhar, 1994] Suresh Manandhar. An attributive logic of set descriptions and set operations. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 255 – 262, Las Cruces, NM, 1994.

[Manandhar, 1995] Suresh Manandhar. Deterministic consistency checking of LP constraints. In *Proceedings of the 7th European Conference of the Association for Computational Linguistics*, Dublin, 1995.

[Manaster-Ramer, 1992] Alexis Manaster-Ramer. Towards transductive linguistics. Technical report, Wayne State University, 1992.

[Marcus, 1980] Mitchell P. Marcus. *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, MA, 1980.

[Martinović, 1994] Miroslav Martinović. Universal guides and finiteness and symmetry of grammar processing algorithms. In *COLING 94*, pages 916 – 921, Kyoto, 1994.

[Matiasek and Heinz, 1993] J. Matiasek and W. Heinz. A CLP based approach to HPSG. Technical Report OEFAI-93-26, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, 1993.

[Matiasek, 1993] Johannes Matiasek. CLP-based HPSG parsing. Technical Report OEFAI-93-02, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, 1993.

[Matiasek, 1994a] Johannes Matiasek. Conditional constraints in a CLP-based HPSG implementation. In Harald Trost, editor, *KONVENS '94*, pages 230 – 239, Vienna, 1994.

[Matiasek, 1994b] Johannes Matiasek. *Principle-based Processing of Natural Language using CLP Techniques*. PhD thesis, Technische Universität, Vienna, October 1994.

[Matsumoto *et al.*, 1983] Yuji Matsumoto, Hozumi Tanaka, H. Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing*, 1:145–158, 1983.

[Maxwell and Kaplan, 1993] John T. Maxwell and Ronald M. Kaplan. The interface between phrasal and functional constraints. *Computational Linguistics*, 19(4):571 – 590, 1993.

[Maxwell III and Kaplan, 1991] John T. Maxwell III and Ronald M. Kaplan. A method for disjunctive constraint satisfaction. In Masaru Tomita, editor, *Current issues in parsing technology*, pages 173 – 190. Kluwer, Boston, 1991.

[Mellish, 1983] Christopher S. Mellish. Incremental semantic interpretation in a modular parsing system. In Karen Sparck Jones and Yorick A. Wilks, editors, *Automatic Natural Language Parsing*, pages 148–155. Ellis Horwood, Chichester, 1983.

[Mellish, 1985] Christopher S. Mellish. *Computer Interpretation of Natural Language Descriptions*. Ellis Horwood/Wiley, Chichester/New York, 1985.

[Mellish, 1988] Christopher S. Mellish. Implementing systemic classification by unification. *Computational Linguistics*, 14(1):40–51, 1988.

[Mellish, 1992] Christopher S. Mellish. Term-encodable description spaces. In D. R. Brough, editor, *Logic Programming: New Frontiers*, pages 189 – 207. Intellect, Oxford, 1992.

[Menzel, 1995] Wolfgang Menzel. Robust processing of natural language. Computation and Language E-Print Archive: cmp-lg9507003, July 1995.

[Meurers, 1994] Walt Detmar Meurers. On implementing an HPSG theory. In Erhard W. Hinrichs, W. Detmar Meurers, and Tsuneko Nakazawa, editors, *Partial-VP and Split-NP Topicalization in German — An HPSG Analysis and its Implementation. Arbeitspapiere des SFB 340, Nr. 58.* Universität Tübingen, 1994.

[Meylemans, 1994] Paul Meylemans. ALEP - arriving at the next platform. *ELS-News*, 3(2):4–5, 1994.

[Michie, 1968] Donald Michie. Memo functions and machine learning. *Nature*, (218):19 – 22, 1968.

[Moshier and Pollard, 1994] M. Andrew Moshier and Carl Pollard. The domain of set-valued feature structures. *Linguistics and Philosophy (Special Issue: Mathematics of Language)*, 17(6):607 – 631, 1994.

[Müller, 1995] Stefan Müller. Head-Driven Phrase Structure Grammar für das Deutsche. (WWW: http://www.compling.hu-berlin.de/˜stefan), 1995.

[Nerbonne, 1994] John Nerbonne. Partial verb phrases and spurious ambiguities. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar*, pages 109 – 150. CSLI, Stanford, CA, 1994.

[Netter, 1992] Klaus Netter. On non-head non-movement. In Günther Görz, editor, *KONVENS 92*, Erlangen, 1992. Springer.

[Neumann and Finkler, 1990] Günter Neumann and Wolfgang Finkler. A head-driven approach to incremental and parallel generation of syntactic structures. In *Proceedings of the 13th International Conference on Computational Linguistics*, pages 288–293, Helsinki, 1990.

[Neumann and van Noord, 1992] Günter Neumann and Gertjan van Noord. Self-monitoring with reversible grammars. In *Proceedings of the 14th International Conference on Computational Linguistics*, Nantes, F, 1992.

[Neumann, 1994a] Günter Neumann. Application of explanation-based learning for efficient processing of constraint-based grammars. In *Proceedings of the Tenth IEEE Conference on Artificial Intelligence for Applications*, pages 208–215, San Antonio, Texas, March 1994.

[Neumann, 1994b] Günter Neumann. *A Uniform Computational Model for Natural Language Parsing and Generation*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1994. (ftp://cl-ftp.dfki.uni-sb.de/pub/papers/local/gn-diss.ps.gz).

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1980.

[Norvig, 1991] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91 – 98, 1991.

[Norvig, 1992] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, CA, 1992.

[Oehrle *et al.*, 1988] Richard T. Oehrle, Emmon Bach, and Deirdre Wheeler, editors. *Categorial Grammars and Natural Language Structures*. Studies in Linguistics and Philosophy. Reidel, Dordrecht, 1988.

[O'Keefe, 1990] Richard O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, 1990.

[Oliva, 1992] Karel Oliva. Word order constraints in binary branching structures. CLAUS Report 20, Computerlinguistik, Universität des Saarlandes, Saarbrücken, February 1992.

[Pareschi and Steedman, 1987] Remo Pareschi and Mark J. Steedman. A lazy way to chart-parse with categorial grammars. In *ACL Proceedings, 25th Annual Meeting*, pages 81–88, 1987.

[Partee *et al.*, 1990] Barbara H. Partee, Alice ter Meulen, and Robert E. Wall. *Mathematical Methods in Linguistics*. Studies in Linguistics and Philosophy. Kluwer, Dordrecht, 1990.

[Pechmann *et al.*, 1994] Thomas Pechmann, Hans Uszkoreit, Johannes Engelkamp, and Dieter Zerbst. Word order in the German middle field: Linguistic theory and psycholinguistic evidence. Technical Report 43, Universität des Saarlandes, Saarbrücken, August 1994.

[Pereira and Shieber, 1984] Fernando C. N. Pereira and Stuart M. Shieber. The semantics of grammar formalisms seen as computer languages. In *10th International Conference on Computational Linguistics*, pages 123–129, Stanford, CA, 1984.

[Pereira and Shieber, 1987] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. CSLI Lecture Notes 10. Center for the Study of Language and Information, Stanford, 1987.

[Pereira and Warren, 1980] Fernando C.N. Pereira and David H.D. Warren. Definite Clause Grammars for language analysis - a survey of the formalism and a comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3):231–278, 1980.

[Pereira and Warren, 1983] Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.

[Pfahringer and Matiasek, 1992] B. Pfahringer and J. Matiasek. A CLP schema to integrate specialized solvers and its application to natural language processing. Technical Report TR-92-37, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, 1992.

[Pfahringer, 1992] Bernhard Pfahringer. How to integrate specialised solvers: A CLP approach. Technical Report TR-92-31, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, 1992.

[Pollard and Moshier, 1990] Carl Pollard and M. Andrew Moshier. Unifying partial descriptions of sets. In Philip P. Hanson, editor, *Information, Language and Cognition, Vancouver Studies in Cognitive Science*. University of British Columbia Press, Vancouver, 1990.

[Pollard and Sag, 1987] Carl Pollard and Ivan Sag. *Information-Based Syntax and Semantics; Volume 1: Fundamentals*. CSLI Lecture Notes 13. Center for the Study of Language and Information, Stanford, CA, 1987.

[Pollard and Sag, 1994] Carl Pollard and Ivan Sag. *Head-driven Phrase Structure Grammar*. University of Chicago Press, Chicago, 1994.

[Pollard, 1984] Carl Pollard. *Generalized context-free grammars, head grammars and natural language*. PhD thesis, Stanford, 1984.

[Ramakrishnan, 1991] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3-4):189 – 216, 1991.

[Ramalingam and Reps, 1991] G. Ramalingam and Thomas W. Reps. On the computational complexity of incremental algorithms. Technical Report 1033, University of Wisconsin, Computer Science Department, Madison, Wisconsin, 1991.

[Reape, 1990] Mike Reape. A theory of word order and discontinuous constituency in West Germanic. In E. Engdahl and M. Reape, editors, *Parametric Variation in Germanic and Romance: Preliminary Investigations*, pages 25–40. ESPRIT Basic Research Action 3175 DYANA, Deliverable R1.1.A, 1990.

[Reape, 1991] Mike Reape. Parsing bounded discontinuous constituents: Generalisations of some common algorithms. In M. Reape, editor, *Word Order in Germanic and Parsing*, pages 41–70. ESPRIT Basic Research Action 3175 DYANA, Deliverable R1.1.C, 1991.

[Reape, 1993a] Mike Reape. *A Formal Theory of Word Order: A Case Study in West Germanic*. PhD thesis, Centre for Cognitive Science, University of Edinburgh, 1993.

[Reape, 1993b] Mike Reape. Getting things in order. In Wietske Sijtsma and Arthur van Horck, editors, *Discontinuous Constituency*. Mouton de Gruyter, Berlin, 1993.

[Reape, 1994] Mike Reape. Domain union and word order variation in German. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar*, pages 151 – 197. CSLI, Stanford, CA, 1994.

[Reithinger, 1992a] Norbert Reithinger. *Eine parallele Architektur zur inkrementellen Generierung multimodaler Dialogbeitraege*. DISKI. INFIX, St. Augustin, 1992. (Dissertation, Universität des Saarlandes).

[Reithinger, 1992b] Norbert Reithinger. The performance of an incremental generation component for multi-modal dialog contributions. In R. Dale and et al., editors, *Aspects of Automated Natural Language Generation, Lecture Notes in Artificial Intelligence No.587*, pages 263–276. Springer, Berlin, 1992.

[Robinson, 1992] J. A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3):40 – 65, March 1992.

[Rounds and Kasper, 1986] W. Rounds and R. Kasper. A complete logical calculus for record structures representing linguistic information. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1986.

[Ruessink, 1994] Herbert Ruessink. Manual to the RGR extensions for ALEP. STT/OTS Utrecht. Preliminary Version., July 1994.

[Samuelsson, 1994a] Christer Samuelsson. *Fast Natural-Language Parsing Using Explanation-Based Learning*. PhD thesis, The Royal Institute of Technology and Stockholm University, Stockholm, Feb 1994.

[Samuelsson, 1994b] Christer Samuelsson. Notes on LR parser design. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 386 – 390, Kyoto, 1994.

[Samuelsson, 1995a] Christer Samuelsson. An efficient algorithm for surface generation. In *IJCAI 95*, Montreal, 1995.

[Samuelsson, 1995b] Christer Samuelsson. Example-based optimization of surface-generation tables. In *Recent Advances in Natural Language Processing*, Velingrad, Bulgaria, 1995.

[Schnelle, 1990] Helmut Schnelle. Netzlinguistische Implementierung von Konstituentenstrukturgrammatiken nach den Prinzipien des Earley-Parser Algorithmus. In Felix, Kanngießer, and Rickheit, editors, *Sprache und Wissen - Studien zur kognitiven Linguistik*. Westdeutscher Verlag, Opladen, 1990.

[Schöter, 1993] Andreas P. Schöter. Compiling feature structures into terms: A case study in Prolog. Technical Report RP-55, University of Edinburgh, Centre for Cognitive Science, 1993.

[Seiffert, 1993] Roland Seiffert. What could a good system for practical NLP look like? In Harald Trost, editor, *Feature Formalisms and Linguistic Ambiguity*, pages 187 – 204. Ellis-Horwood, 1993.

[Sells, 1985] Peter Sells. *Lectures on Contemporary Syntactic Theories*. CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA, 1985.

[Shepherdson, 1984] C. J. Shepherdson. Negation as failure: A comparison of Clark's completed data base and Reiter's closed world assumption. *Journal of Logic Programming*, 1(1):51 – 79, 1984.

[Shieber *et al.*, 1983] Stuart M. Shieber, Hans Uszkoreit, Fernando C. N. Pereira, Jane J. Robinson, and Mabry Tyson. The formalism and implementation of PATR-II. In Barbara J. Grosz and Mark E. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge*, pages 39–79. SRI International, Menlo Park, CA, 1983.

[Shieber *et al.*, 1990] Stuart M. Shieber, Gertjan van Noord, Fernando C. N. Pereira, and R. C. Moore. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42, 1990.

[Shieber *et al.*, 1994] Stuart M. Shieber, Yves Shabes, and Fernando C. N. Pereira. Principles and implementation of deductive parsing. Technical Report TR-11-94, Center for Research in Computing Technology, Harvard University, April 1994. (Computation and Language e-print archive cmp-lg/9404008).

[Shieber, 1983] Stuart M. Shieber. Sentence disambiguation by a shift-reduce parsing technique. In *IJCAI-83*, pages 699–703, 1983.

[Shieber, 1985] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, 1985.

[Shieber, 1986] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Lecture Notes No. 4. Center for the Study of Language and Information, Stanford, CA, 1986.

[Shieber, 1988a] Stuart M. Shieber. Separating linguistic analyses from linguistic theories. In Uwe Reyle and Christian Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 33–68. Reidel, Dordrecht, 1988.

[Shieber, 1988b] Stuart M. Shieber. A uniform architecture for parsing and generation. In *COLING-88*, pages 614–619, 1988.

[Sikkel, 1994a] Klaas Sikkel. A framework for parsing algorithm specification and analysis. In Harald Trost, editor, *KONVENS '94*, pages 300 – 309, Vienna, 1994.

[Sikkel, 1994b] Klaas Sikkel. How to compare the structure of parsing algorithms. In *First Workshop on Algebraic and Syntactic Methods in Computer Science*, Milano, 1994.

[Sikkel, 1994c] Klaas Sikkel. *Parsing Schemata*. PhD thesis, University of Twente, Enschede, 1994.

[Smolka and Treinen, 1994] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Smolka *et al.*, 1995] Gert Smolka, Martin Henz, and Jörg Würtz. Object-oriented concurrent constraint programming in Oz. In P. van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, chapter 2, pages 27–48. The MIT Press, 1995.

[Smolka, 1988] Gert Smolka. A feature logic with subsorts. LILOG Report 33, IBM Deutschland, May 1988.

[Smolka, 1992] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51 – 87, 1992.

[Smolka, 1993] Gert Smolka. Residuation and guarded rules for constraint logic programming. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 405–419. MIT, Cambridge, Mass., 1993.

[Stabler, 1990] Edward P. Stabler. *The Logical Approach to Syntax*. PhD thesis, University of Western Ontario, Dept. of Computer Science, London, Ontario, 1990.

[Sterling and Shapiro, 1986] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

[Strzalkowski, 1991] Tomek Strzalkowski. A general computational method for grammar inversion. In Tomek Strzalkowski, editor, *ACL SIG workshop "Reversible Grammar in Natural Language Processing"*, pages 91 – 99, Berkeley, CA, 1991.

[Thompson, 1983] Henry Thompson. MCHART: a flexible, modular chart parsing system. In *Proceedings of the Meeting of the American Association for Artificial Intelligence*, pages 408 – 410, 1983.

[Thompson, 1990] Henry Thompson. Best-first enumeration of paths through a lattice — an active chart parsing solution. *Computer Speech and Language*, 4:263 – 274, 1990.

[Trost and Matiasek, 1994] Harald Trost and Johannes Matiasek. Morphology with a null-interface. In *Proceedings of the 15th International Conference on Computational Linguistics*, Kyoto, Japan, 1994.

[Trost, 1993] Harald Trost, editor. *Feature Formalisms and Linguistic Ambiguity*. Ellis Horwood, Chichester, 1993.

[Uszkoreit, 1986] Hans Uszkoreit. Categorial unification grammar. In *COLING-86*, pages 187–194, 1986.

[Uszkoreit, 1987a] Hans Uszkoreit. Linear precedence in discontinuous constituents: Complex fronting in German. In Geoffrey J. Huck and Almerindo E. Ojeda, editors, *Discontinuous Constituency*, pages 405–425. Academic Press, Orlando, FL, 1987.

[Uszkoreit, 1987b] Hans Uszkoreit. *Word Order and Constituent Structure in German*. CSLI Lecture Notes 8. Center for the Study of Language and Information, Stanford, CA, 1987.

[Uszkoreit, 1988] Hans Uszkoreit. From feature bundles to abstract data types: New directions in the representation and processing of linguistic knowledge. In Blaser, editor, *Natural Language at the Computer — Contributions to Syntax and Semantics for Text Processing and Man-Machine Communication*. Springer, Heidelberg, Berlin, New York, 1988.

[Uszkoreit, 1991] Hans Uszkoreit. Strategies for adding control information to declarative grammars. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pages 237–245, Berkeley, CA, 1991.

[van der Linden, 1993] Erik-Jan van der Linden. *A Categorial, Computational Theory of Idioms*. PhD thesis, University of Utrecht, Utrecht, 1993.

[van Harmelen and Bundy, 1988] Frank van Harmelen and Alan Bundy. Explanation-based generalization = partial evaluation. *Artificial Intelligence*, 36:401 – 412, 1988.

[van Hentenryck and Dincbas, 1986] P. van Hentenryck and M. Dincbas. Domains in logic programming. In *Fifth National Conference on Artificial Intelligence (AAAI-86)*, Los Altos, CA, 1986. Morgan Kaufmann.

[van Hentenryck, 1989] Pascal van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[van Noord, 1993] Gertjan van Noord. *Reversibility in natural language processing*. PhD thesis, Rijksuniversiteit Utrecht, NL, 1993.

[van Noord, 1994] Gertjan van Noord. *HDRUG: A Graphical User Environment for Natural Language Processing in Prolog*. Rijksuniversiteit Groningen, Groningen, 1994. (http://grid.let.rug.nl/~vannoord/diss/diss.html).

[Vijay-Shanker *et al.*, 1987] K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting*, pages 104–111, Stanford, CA, 1987. Association for Computational Linguistics.

[Wahlster, 1993] W. Wahlster. Verbmobil: Translation of face-to-face dialogs. In O. Herzog, T. Christaller, and D. Schütt, editors, *Grundlagen und Anwendungen der künstlichen Intelligenz(17. Fachtagung)*, pages 393–402. Springer, Berlin, Heidelberg, 1993.

[Wanner, 1992] Leo Wanner. Lexical choice and the organization of lexical resources in text generation. In Bernd Neumann, editor, *Tenth European Conference on Artificial Intelligence*, pages 495 – 499, Vienna, 1992. Wiley.

[Warren, 1975] David H. D. Warren. Earley deduction. Unpublished note, 1975.

[Warren, 1989] David S. Warren. The XWAM: a machine that integrates Prolog and deductive database query evaluation. Technical Report 89/25, Department of Computer Science, SUNY at Stony Brook, Oct. 1989.

[Warren, 1992] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93 – 111, 1992.

[Weir, 1988] David J. Weir. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.

[Wirén and Rönnquist, 1993] Mats Wirén and Ralph Rönnquist. Fully incremental parsing. In *Third International Workshop on Parsing Technologies*, Tilburg, NL and Durbuy, B, 1993.

[Wirén, 1987] Mats Wirén. A comparison of rule-invocation strategies in context-free chart parsing. In *ACL Proceedings, Third European Conference*, pages 226–235, 1987.

[Wirén, 1992] Mats Wirén. *Studies in Incremental Natural-Language Analysis*. Linköping Studies in Science and Technology: Dissertation No. 292. Department of Computer and Information Science, Linköping University, Sweden, Linköping, 1992.

[Wirén, 1994] Mats Wirén. An approach to bounded incremental parsing. In *COLING*, 1994.

[Zadrozny, 1992] Wlodek Zadrozny. On compositional semantics. In *COLING*, Nantes, F, 1992.

[Zajac, 1992] Rémi Zajac. Inheritance and constraint-based grammar formalisms. *Computational Linguistics (Special Issue on Inheritance: I)*, 18(2):159 – 182, 1992.