EAGLES

Expert Advisory Group on
Language Engineering Standards

EAGLES

Formalisms Working Group Final Report

Version of September, 1996

The EAGLES Formalisms Working Group
Final Report

Rolf Backofen
Tilman Becker
Jo Calder
Joanne Capstick
Luca Dini
Jochen Dörre
Gregor Erbach
Dominique Estival
Suresh Manandhar
Anne-Marie Mineur
Gertjan van Noord
Stephan Oepen
Hans Uszkoreit

December 18, 1996

Contents

Executive Summary

A single standard could not be specified.

At present a convergence of European language technology on one single grammar formalism is neither realistic nor desirable. The reasons for this conclusion are:

- No grammar formalism fulfils all requirements posed by the spectrum of ongoing R&D projects.
- For different types of projects, different formalisms constitute the optimal choice.
- the current variety of employed formalisms is a better basis for the emergence of improved formalisms than premature standardization.

From these observations, the Formalisms Working Group has drawn the following conclusions:

- The Formalisms Working Group has not tried to propose a single standard or pre-standard specification or exchange format for grammar formalisms.
- the Formalisms Working Group has not participated in the proposal for EAGLES II.

However, considerable progress has been achieved.

The insight that time is not ripe for a single standard is not meant to imply that no progress can be reported. Quite the opposite is true. During the survey phase, the Formalisms Working Group was able to witness and describe considerable progress in the area of linguistic formalisms and grammar engineering. This progress is measurable– development times for implemented grammars could be drastically reduced by exploiting the advantages of contemporary formalisms. The Formalisms Working Group can report on the first successful cases of efficient multilingual parallel grammar development. The Formalisms Working Group can also report on the first cases of grammar reuse by migration from one formalism to another.

Through its workshops and through the work on exchange formats, reusability and actual grammar migration, the Formalisms Working Group was able to make relevant contributions to this development. The Formalisms Working Group was able to contribute to the choice of formalisms and engineering approaches in several large European LT projects.

WWW pointers to relevant information on grammar formalisms, implemented grammars, and development platforms have been compiled and published on the web to assist new industrial and academic LT projects.

Noticeable progress has been achieved by the members of the Working Group in the following areas:

- survey and comparison of a wide variety of existing implemented formalis m
- description of ongoing convergence in language technology on a small class of formalisms

- initiation of exchange between the research areas of constraint logic programming and grammar formalisms
- promotion of exchange and cooperation between major academic and industrial research centers
- grammar resusability and migration

Convergence can be reported.

The Formalisms Working Group has found a convergence of R&D in language technology on a small class of formalisms. Although a great number of formalisms have been proposed and utilized in basic research; application-oriented projects involving large-scale grammar engineering have converged on formalisms from the following three linguistic grammar models:

- Head-Driven Phrase Structure Grammar (HPSG)
- Lexical Functional Grammar (LFG)
- Tree-Adjunction Grammar (TAG)

HPSG is by far the most widely used framework. It is employed in many large R&D projects. TAG and LFG are much more restricted in distribution but utilized in a few very important projects.

Other linguistic frameworks such as Categorial Grammar, Dependency Grammar, and Government and Binding are widespread in the academic community and play an important role in basic research. However, these frameworks are neither used in large-scale grammar engineering projects nor in application development. None of their implemented formalisms are shared by more than two research sites.

We have surveyed properties of the abstract and implemented formalisms of these frameworks and investigated the question of standardisation accross frameworks. We arrived at the following conclusions:

- A common format for the three classes of formalisms is as impossible or nonsensical as a common syntax for LISP, PASCAL and PROLOG.
- A translation or migration between HPSG and LFG does not make sense because of their different architectures.
- The compilation of HPSG into TAG has proven very useful in certain contexts (Verbmobil).
- The compilation of LFG into TAG might be possible and useful as well.

Important steps towards reusability have been taken.

Concerning standardization within the three classes of formalisms, we can report the following findings:

- In LFG, two almost identical formalisms are used: the LFG-Workbench formalism and the XLE formalism (both by XEROX PARC). Tools for nearly automatic migration from the older LFG-Workbench formalism to the XLE formalism are being developed together with the XLE development platform. The XLE is used in the ParGram project carried out jointly by XEROX PARC, RXRC Grenoble, and the IMS at the University of Stuttgart. There is no need for further standardisation efforts.

- In TAG, there are two major formalisms used for grammar engineering. The XTAG formalism by the University of Pennsylvania and the TAGDevEnv/TAGGEN formalism by the DFKI Saarbruecken. The close exchange and collaboration of the two centers might eventually lead to exchange formats or migration tools.
- The greatest need for exchange formats or migration tools exists for formalisms utilized for the development of HPSG grammars in many projects. The most widely used formalisms are ALE, TDL, ALEP, CUF, and TFS.

Two major results could be achieved in the area of common formats, resusability and migration:

- A common specification format was developed that ensures translatability of coded grammars into ALE, ALEP, and CUF.
- The automatic migration from ALEP grammars as they were specified in the project LS-GRAM into TDL was developed.

This development permits the migration of the LS-GRAM grammars into TDL, the formalism of the DFKI grammar engineering platform PAGE. Since PAGE is used in two very large projects (Verbmobil and ERGO), this development will be of immediate practical relevance.

Future progress can be predicted.

We expect new impulses and tangible results to emerge from:

- combination of grammar specification with grammar induction from corpora
- exploitation of the advantages of high-level logic-based grammar formalisms for grammar engineering
- improved methods for modular distributed linguistic engineering
- exploitation of run-time analysis tools, annotated corpora and large structured test data into tools for glass-box evaluation and diagnosis
- exploitation of improved compilation methods for deriving efficient and task-specific performance grammars from large-scale resource grammars
- induction of control knowledge from corpora
- **emergence of truly open system architectures for grammar engineering and for rapid prototyping of applications**

Chapter 1

Introduction

This document gives an overview of current developments in grammar formalisms, and closely related issues such as the underlying grammatical theories, mathematical foundations, implemented grammars and development environments. We outline trends towards convergence that have become apparent in recent years, and make some recommendations based on an analysis of these trends.

Even though the imposition of standards would be premature in the rapidly evolving area of grammar formalisms, the convergences in grammar formalisms which have been observed for particular grammatical theories have encouraged us to propose draft encoding standards, which can serve as a formalism-independent, albeit not theory-independent, notation for grammatical knowledge, which can be mapped to particular existing formalisms for processing.

The Working Group

The EAGLES initiative Working Group on Linguistic Formalisms brings together experts on the design and implementation of linguistic formalisms from academia and industry in order to :

- come to a consensus on the basic features and properties for NLP formalisms and indicate likely and needed future features;
- promote consensus with respect to the definition of de facto standards for grammar formalisms;
- exchange information about each other's projects and, as far as is compatible with intellectual property rights, know-how and results, thus increasing the awareness of possible synergies;
- where appropriate, concretize potential synergies by promoting cooperative actions, thus furthering the definition of de-facto standards in the field;
- disseminate information about the working group's activities, participate in and organize events aimed at make these activities better known (round-tables, workshops, conferences);
- coordinate and cooperate with national and international initiatives; and
- suggest actions needed for the creation of formal and computational prerequisites for the development of multilingual, reusable, grammatical resources.

The group is hosted by the DFKI in Saarbrücken.

The members of the WG are:

H. Ulrich Block - Siemens AG, Munich,

Ewan Klein - University of Edinburgh,
Jeremy Peckham - Logica Cambridge,
Steve Pulman - SRI Cambridge,
Christian Rohrer - University of Stuttgart,
Hans Uszkoreit - DFKI Saarbrücken (chair).

Many additional industrial and academic research institutions are represented by specialists in the three subgroups of the Working Group:

Linguistic Adequacy,
Computability and Implementation,
Industrial Requirements.

Activities of the Working Group 1993 - 1995

At the beginning of EAGLES, there were a number of open questions concerning grammar formalisms:

1. **Existing Formalisms:** Which formalisms do currently exist? What are the design decisions embodied in each formalism? How does each formalism relate to other formalisms? What has it been used for? For which grammatical theories is it intended? How adequate is it for these theories? How efficient is it? How does it support the development of large-scale grammars? What are the plans for the further development of the formalism?

2. **Linguistic Adequacy:** How does one determine the linguistic adequacy of a formalism? Are existing formalisms expressive enough, and do they enable generalisations to be expressed? How can complexity be handled or reduced? How are efficient processing, robustness, modularity, scalability and reusability achieved?

3. **Relationship between Constraint Logic Programming and NLP Formalisms:** What are the current developments and trends in CLP? Which of these developments are relevant to open problems in NLP? Are existing CLP languages appropriate platforms for the implementation of advanced linguistic formalisms? What are the main approaches for integrating CLP methodology into NLP? How should the exchange and collaboration between the CLP and NLP communities be improved?

In order to arrive at answers to these fundamental questions, which are prerequisites for any kinds of recommendation or standardisation efforts, three workshops were organised which brought together leadings researchers and practitioners for an exchange of information and ideas. In the following, we briefly summarise the results of these workshops.

Workshop on implemented formalisms

The workshop on implemented formalisms took place at the German Research Center for Artificial Intelligence (DFKI) in Saarbrücken, from March 1st to 3rd, 1993.

The main objective of the workshop was to obtain an urgently needed overview of existing software systems, including their development platforms, that implement state-of-the-art grammar formalisms. It is not possible to derive such an overview by surveying the literature. In their publications, the developers of such formalisms usually focus on certain selected aspects of their systems that constitute novel scientific approaches. The state of the implementation of a system and its robustness, performance and overall usability can never be judged from the literature.

Another motivation was the broadly-felt necessity to exchange experience gathered in implementing constraint-based grammar formalisms among the relevant developers.

Therefore, the focus of this workshop was not on the linguistic, philosophical and semantic foundations of advanced typed feature-unification formalisms. The meeting concentrated on existing implementations. To participants working in the area of formalism development, the workshop offered a unique opportunity to learn those facts about other researchers' formalisms that cannot be found in the literature.

14 formalisms were presented at the workshop, 13 of these through system demonstrations. All participants filled out a questionnaire with detailed information about their formalism. The results of the workshop were published as a 110-page report.

One goal of the workshop was to obtain an overview of what is feasible and usable today. This overview served as the starting point for the Working Group's activities during the survey phase. It is an important part of the ongoing survey of existing implemented linguistic formalisms.

Workshop on the Linguistic Adequacy of Linguistic Formalisms for NLP

The workshop on the linguistic adequacy took place in Lisbon during the European Summer School for Logic, Language and Information on August 28-29 1993. The goal of the workshop was to provide key researchers in the field an opportunity for brainstorming on the definition of the concepts involved in the question of the linguisitic adequacy of formalisms for natural language processing.

The following key topics were addressed at the workshop:

- How do we determine linguisitic adequacy?
- Are existing formalisms expressive enough to allow for grammars with substantial coverage?
- Are the formalisms committed to a particular theory? Up to what degree do they allow generalisations to be expressed?
- How can the complexity be managed or reduced? Can large grammars be processed with complex formalisms?
- How can weights be used?

The workshop provided valuable input to the work of the FWG. Although it did not provide clear-cut answers to all the questions, the discussions of the workshop already provide a very good idea of the directions in which researchers in the field are looking for answers. The meeting was also important for increasing the visibility of the EAGLES work among the users of linguistic formalisms. Their requirements are essential for the recommendations of the working group, and the concrete suggestions made at the workshop have guided the activities of the FWG.

Workshop on the relationship between grammar formalisms and Constraint Logic Programming

Although researchers in grammar formalisms and in constraint logic programming had been working on similar and complementary issues for quite some time, and had taken over methods from each other, it was felt that more contact between the fields was needed. During the workshop on implemented formalisms, many grammar formalisms were explicitly described as CLP instances.

In order to provide a forum for exchange between the CLP and NLP communities, the EAGLES FWG organised a workshop in Porto from 6-9.02.1994, for which leading researchers from both fields were invited.

The following questions were discussed at the workshop:

1. What are the most interesting current developments and trends in CLP?
2. Which of these developments are directly relevant to open problems in NLP?

3. Are existing LP and CLP languages such as LIFE and Oz appropriate platforms for the implementation of advanced linguistic formalisms?

4. What do LP and CLP languages have that is still missing in NLP?

5. What are the main approaches for integrating CLP methodology into NLP?

6. Will the CL work on integrating CLP methods into NLP yield any results that are of real interest to the CLP community?

7. Is NLP a good application domain? Is it a good test case? Is is a typical/diagnostic test case or are CLP languages designed for quite different tasks?

8. What are good examples for cross-fertilisation and collaboration between the communities?

9. How should the exchange or collaboration between the communities be improved?

An important result of the workshop was a discussion of the main approaches for integrating CLP methodology into NLP (integrating CLP techniques into NLP formalisms, mapping high-level linguistic descriptions into existing CLP languages, and expressing grammars directly in a CLP language by operationalising the grammatical specification through the facilities of the CLP language). The benefits and drawbacks of each approach were discussed.

It became clear from the workshop that both fields can only benefit from closer cooperation, ideally through a project staffed with the right people from both fields.

The success of this workshop prompted further events such as the workshop *Computational Logic and Natural Language Processing* which was held in South Queensferry, Scotland, in the spring of 1995. This workshop was jointly organised by EAGLES, ELSnet, and Compulog Net.

In December 1995, a paper outlining the requirements covering a CLP language for the needs of NLP was presented by members of the EAGLES FWG during the workshop *Visions for the Future of Logic Programming* held as part of the International Logic Programming Symposium.

Encoding Standards

In the spring of 1995 it had become evident that there was a slowdown in the rate at which new formalisms were being developed, and older ones being upgraded. Among the widely-used implemented formalisms, there was some convergence in the expressive power and the descriptive means they provide. At the same time, it was also possible to see some convergence in the formal devices employed for the development of large grammars.

At the same time it had become clear that ALEP would not be universally accepted as a European standard for a grammar formalism. This was due to problems in the ALEP implementation, which hindered its acceptance for some grammar writers, and due to the fact that different projects have different needs, which are best fitted by different formalisms (see chapter 5).

At this point, the EAGLES FWG decided to make an effort to specify an encoding standard that could serve as a common interchange format between different grammar formalisms, and as a specification language for a repository of grammatical knowledge.

A workshop was arranged in Barcelona during the European Summer School in Logic, Language and Information. During this workshop, several alternative proposals for an encoding standard were considered, and the conclusion in the end was to have three separate encoding standards for LFG, TAG and HPSG, instead of trying to devise one standard that unites all of them and others. These three grammatical theories were chosen because many large-scale computational grammars are developed in them.

For the rest of 1995, the encoding standards were worked out in more detail by a group of people, and a working draft of the resulting encoding standards is presented in chapter 7 of this report.

The proposed standards try to reconcile two conflicting requirements: on the one hand, they are intended to be expressive enough to accommodate the needs of grammatical descriptions, and on the other hand they should be able to translate grammars encoded in the standard into existing formalisms. In order to accommodate these requirements, the encoding standards are modular, and for each construct defined in a module it is specified which constructs can be directly encoded or approximated in which formalism.

The encoding standards will have to go through a series of tests and revisions before they can be widely used. The first step will be to port some existing grammars into the encoding standard. There is strong support from the HPSG, TAG and LFG communities to continue the work on the encoding standards which was inaugurated by the EAGLES FWG.

Guide to grammar formalisms

It is often not easy for people from outside the field of grammar formalisms to get an overview of the available formalisms, to fully comprehend the implications and consequences of the choice of a formalism, and to make wise and well-informed decisions.

As a service to current and future users of grammar formalisms, this report contains some advice to help in the choice of a formalism, and lists sources of information about existing formalisms, grammars and tools (see chapters 5 and 6).

Acknowledgements

The report contains contributions from the following people:
Gabriel Amores (JULIETTA and Lekta), Universidad de Sevilla,
Jo Calder (Pleuk), University of Edinburgh,
Brigitte Krenn (Constraint Grammar), University of the Saarland,
Mary Dalrymple (various LFG parts), Xerox PARC, Palo Alto,
Jose F. Quesada (JULIETTA and Lekta), Universidad de Sevilla,
Wojciech Skut (Dependency Grammar), University of the Saarland,
Axel Theofilidis (LS-GRAM coordination), IAI Saarbrücken,
Craig Thiersch (GB), ITK Tilburg.

We made use of the results and experiences of other projects, especially those involved in the specification, implementation, support, extension and application of the ALEP formalism (ET 6/2, ET 10/52, RGR, LS-GRAM). In addition, we used experiences from the German Verbmobil project, which also carries out grammar development in several formalisms.

The design of the encoding standards was greatly helped and influenced by discussions with Jo Calder, Nicoletta Calzolari, Dan Flickinger, Hans-Ulrich Krieger, Rob Malouf, Johannes Matiasek, Paola Monachesi, Martin Müller, Klaus Netter, Susanne Riehemann, Herbert Ruessink, Ivan Sag, Antonio Sanfilippo, Wojciech Skut, Gert Smolka, Ralf Steinberger, and Pete Whitelock.

Presentations of early versions of the encoding standards were given at Sharp Labs of Europe in Oxford, and during the EAGLES workshop on subcategorisation standards in Abingdon. The feedback received during these workshops contributed greatly to the design of the encoding standards.

Chapter 2

State of Affairs at the Beginning of EAGLES

2.1 Introduction

This chapter covers the most important aspects relating to grammar formalisms: grammar models, mathematical formalisms, implemented formalisms and development platforms, and implemented grammars.

We follow the established distinction between grammar models, grammatical theories, mathematical formalisms, implemented formalisms, and implemented grammars.

A grammar model is a set of devices for talking about analyses of phenomena in natural language. The individual analyses of natural languages within these models are theories about grammar.

A mathematical formalism is a set of abstract datatypes, which formalises the descriptive and explanatory devices utilised in grammar models. A mathematical formalism can correspond exactly to one grammar model, or it can be more abstract and accommodate several grammar models. In the latter case, the mathematical formalism must allow the assumptions made in the grammar model to be stated explicitly.

An implemented formalism is the computational realisation of a mathematical formalism. In addition to the abstract datatypes, it normally contains a set of tools for grammar writing, inspection, debugging, and parsers or generators for processing. An implemented formalism may be bound to one grammar model, or can be used to accommodate several grammar models.

An implemented grammar is the formalisation of a grammatical theory within an implemented formalism.

Each section or subsection was compiled by an expert in the field and gives a more or less subjective view of the state of affairs at the beginning of the EAGLES initiative. The overview is not meant to be complete, but rather it is a selection of issues which are considered important by experts in their respective field.

2.2 Grammar Formalisms/Models

2.2.1 Introduction

This section provides an overview of the state of the art of various grammatical theories at the beginning of the project, i.e. 1993. Computational relevance has guided both the choice of the theories under account (Categorial Grammar, Dependency Grammar, Government and Binding Theory, Head-driven Phrase Structure Grammar, Lexical Functional Grammar and Tree Adjoining Grammar) and the description of the theories themselves. Thus theoretical aspects are sometimes backgrounded in favour of the description of works which present more interesting (or innovative) features from the computational point of view.

Since different authors developed various subsections no homegeneous format of description has been adopted. This was in order to favour a personal interpretation of the state of art of the Grammatical Formalisms/Models over an aseptic standard description, which could be found in any textbook of linguistics.

2.2.2 Categorial Grammar

What makes a grammar a categorial grammar is the insight that the composition of a larger constituent from smaller pieces can always be viewed in terms of functional application. One part of the constituent can be seen as the functor, that takes the rest of the constituent as its argument. Concatenation (•) of a string of type A/B (the functor) to a string of type B (the argument) results in a compound string of type A. Or, more formally:

$$\mathrm{A/B} \bullet \mathrm{B} \rightarrow \mathrm{A}$$

A functor looking for an argument to its right can be written as

$$\mathrm{B} \bullet \mathrm{B} \backslash \mathrm{A} \rightarrow \mathrm{A}$$

or, alternatively, as

$$\mathrm{B} \bullet \mathrm{A} \backslash \mathrm{B} \rightarrow \mathrm{A}$$

The insight that composition can be seen as functional composition dates back to Montague ([Montague, 1974]), who used it in his Lambda-calculus. Independently, Ajduckiewicz ([Ajduckiewicz, 1935]) devised a categorial grammar based on this insight.

Steedman and Dowty ([Steedman, 1984], [Dowty, 1988]) used these insights to develop a linguistically motivated Combinatorial Categorial Grammar (CCG), whereas Van Benthem and Moortgat ([van Benthem, 1986], [Moortgat, 1988]) combined Lambek calculus with Categorial Grammar, and went in a more logically oriented direction. For Steedman *et.al.* linguistic motivations may supersede logical ones, and their line of work can be characterized as using categorial notation to describe possible grammatical configurations.

For Van Benthem *et.al.*, on the other hand, no operation that endangers the correctness and completeness of the current system is allowed. Deviating grammatical configurations (take for instance long-distance dependencies, where the argument does not appear immediately next to the functor: A/B, X_1, ... X_n, B) impose a change on the properties of the entire system, and imply the introduction of new connectives. A second important distinction is that in Lambek CG there is no longer an isomorphic relation between the dependency tree and the parse tree of a sentence. Unlike Steedman, these researchers claim that ambiguous derivations can only be considered ambiguous if they have any semantic impact, otherwise they are *spurious*. These two fundamental differences result in a very clean, but rather abstract framework.

2.2.3 Dependency Grammar

Dependency Grammar as a linguistic theory dates back to Tesnière ([Tesnière, 1959]). It describes language in terms of *dependency structures*, i.e. relationships holding between words and motivated primarily by their syntactic and semantic valency. Because of this close connection to valency phenomena, purely configurational explanations are far less common in Dependency Grammar than in theories rooted in the tradition of Phrase Structure Grammar. It is presumably for this reason that Dependency Grammar has often been used to describe free word order languages (Slavic languages, but also German).

Dependency relations are usually sub-divided into classes such as *complementation* and *modification*, also known from other linguistic theories. In contrast to GB and related approaches, these relationships are primitives rather than notions derived from syntactic structure.

Three tendencies are characteristic for the current situation within Dependency Grammar:

- extending the traditional concepts of DG with new formal devices (mostly borrowed from the unification-oriented branch of Phrase Structure Grammars),
- finding convergence between DG and other linguistic formalisms,
- exploring the specific formal properties of DG and how they can contribute to the explanation and formalisation of a number of linguistic phenomena

The following demonstrates these tendencies by example of linguistic theories. Recently there has been research into adopting some formal tools used in the other branches of contemporary linguistics. Hellwig (cf. [Hellwig, 1986]) has attempted to reformulate the basic concepts of Dependency Grammar in terms of a unification-based feature formalism. The approach combines the use of some standard formal solutions (e.g. partially specified feature slots for encoding valency frames) with new concepts: the handling of word order is based on *positional attributes*, i.e. the position of a word or constituent is always explicitly encoded in the feature structure with the help of a number of morphosyntactic features. In [Hellwig, 1986] there are three of them: *sequence*, which encodes the linear precedence relation, *adjacency* (immediate precedence), and *delimitation* (being the leftmost/rightmost dependent of the head).

This encoding allows for representing discontinuous constituents, a phenomenon which most of the theories rooted in Phrase Structure Grammar or Categorial Grammar have difficulties to deal with and which is usually handled by means of additional mechanisms such as transformations, functional composition and the so-called SLASH attribute.

In [Hellwig, 1988], Hellwig also shows how a chart parser can be adapted to the task of processing possibly discontinuous dependency structures. The idea is to give up the requirement that functors be adjacent to their arguments and represent substrings of the input by bit vectors, so that for each constituent a word marked with a one bit is contained in this constituent, whereas 0 marks those words that are not parts of it. When a number of constituents combine, the representation of the result can be computed by OR-ing the representation of the sub-phrases.

While Hellwig concentrates on the formal aspects of implementing and processing dependency grammars, there have also been new DG-oriented analyses of linguistic phenomena, especially those difficult to handle within Phrase Structure and Categorial Grammar.

Word Grammar (cf. [Hudson, 1984]) rejects the notions of constituent and phrase as intermediate structures of language, and postulates that the description of a language should be formulated in terms of relations holding between words. This claim may be viewed as a consequent continuation of the current tendency towards lexicalised grammars with a prominent notion of syntactic head, i.e. a word that bears all the important information that is associated with the constituent. In comparison with other theories of grammar, dependencies between phrases are, in Hudson's theory, replaced by dependencies between their heads.

Thus, in the phrase *a student of linguistics* the determiner *a* is the head of *student* which itself is the head of the preposition *of*. The noun *linguistics* is a dependent of *of* (or, in Hudson's terminology, a *modifier*[1] of *of*). This solution also allows for an easier treatment of idiosyncrasies concerning argument selection.

[1] Hudson uses the term *modifier* in a different meaning than authors with a GB or HPSG background do. For him, this word is just synonymous to *non-head*.

Unlike theories based on the notion of syntactic tree, Word Grammar allows for a straightforward representation (in terms of dependency structures) of the situation in which a word is dependent on more than one head. This proves advantageous when, for instance, control constructions are analysed and a noun phrase turns to be the subject of both the matrix and the embedded verb.

Hudson's approach has also proved capable of formalising some further linguistic phenomena which Phrase Structure Grammar has problems with, i.e. cross-serial dependencies in Dutch.

Another theory close to Dependency Grammar is Lexicase (cf. [S. Starosta, 1986]). It combines certain semantic concepts from Fillmore's Case Grammar (e.g. the 'deep case' relation) with a dependency-grammatical approach to valency, which is brought to bear especially in the case of parsing. The valency information (which may be viewed as a dependency tree) is stored in the lexical entries, and the parsing algorithm just looks for constituents to fill the valency slots, basically in a bottom-up way (there are some minor differences between the major categories in this respect).

2.2.4 Government and Binding Theory

At the time of the submission of the tender for the Eagles project, so-called Government/Binding theory was in a (fairly) stable state, based on the Principles and Parameters approach outlined in Chomsky's (1986) *Barriers*, and his (1981) *Lectures on Government and Binding*. There was also a great deal of descriptive grammatical analysis, much of which aimed at revising the various hypothesized conditions. Briefly summarized, the then-current paradigm was relatively old: the theory assumed a set of levels of description (usually four: D-Structure, S-Structure, Logical Form, and Phonological Form). Each level (except perhaps PF) consisted of trees (later tree-like structures), which were mapped onto each other by the derivational rule Move-α. D-structure corresponded roughly to thematic structure, and S-structure was an intermediate stage from which the mapping diverged to PF and LF. It was justified as the stage of a derivation where certain conditions were tested. Overgeneration was prevented by restrictions on Move-α in the form of (universal) principles, modified by language-specific parameters. In the course of time the parameters were localized in the Lexicon. As this was a static theory, intended to reflect (universal human) grammatical competence, the relation to on-line parsing and generation was unclear, and the considerable computational problems were the subject of much discussion and the focus of a number of implementations.

2.2.5 Head-Driven Phrase Structure Grammar

In pages 14- 18 of this section we will present part of the contents of the HPSG theory as described in [Pollard and Sag, 1987] (henceforth PS87). Rather than providing a systematic description of the theory (which is already done in the book) we will try to focuss on those aspects which are either interesting from a formal point of view or which are at the center of some theorical discussion. In pages 18- ?? we will explore ideas which have been put forward in other work and which had a significant influence in the field of HPSG linguistics.

Grammar Rules

> A *grammar rule* is simply a very partially specified phrasal sign which constitutes one of the options offered by the language in question for making big signs from little ones (PS87, p. 147)

Thus grammar rules, together with the lexicon, are *disjunctively* specified, whereas the standard HPSG principles are required to be satisfied *conjunctively* As for their format, rules are encoded not through standard PS rules, but as templates. The following, for instance, takes care of attaching the subject to some X' projection:

(1) Rule 1

$$\begin{bmatrix} \text{SYN}\,|\,\text{LOC}\,|\,\text{SUBCAT} & \langle\rangle \\ \text{DTRS} & \begin{bmatrix}\text{HEAD-DTR}\,|\,\text{SYN}\,|\,\text{LOC}\,|\,\text{LEX} & - \\ \text{COMPS-DTRS} & \left\langle[\,]\right\rangle\end{bmatrix}\end{bmatrix}$$

As is noted, the phrasal structure of the sign is encoded in the sign itself. This fact has several positive consequences and only one drawback:

- It allows the encoding of the principles as types or constraints. Indeed, the daughter nodes of a structure are accessible as standard linguistic descriptions and principles can thus be coded as constraints over the type *sign* (or whatever).
- It provides an elegant way of encoding principles of phrasal organization, which can be structured as any other linguistic description (use of types, subtypes, partitions...).
- Conceptually, there is the possibility to preserve a certain homogeneity in the descriptive tools of a language, as, in principle, no additional structure such as *trees*, distinct from *feature structures*, is needed.
- On the other hand, the requirement of locality seems to be violated by encoding phrase structures directly within the sign. Indeed, even though Pollard and Sag are rather attentive to avoiding violations of the principle of locality (especially in subcategorization), this remains one of the formal possibilities of the system: for instance one would have the formal tools to encode a phrasal type which says that every sign is grammatical, which has a head with a daughter with a specifier which begins with the letter *t*. Even though nobody is likely to write a constraint like this, this remains, unfortunately, a possibility of the grammar.

Besides Rule 1, which we already stated, the following rules are assumed in PS87:

(2) Rule 2

$$\begin{bmatrix} \text{SYN}\,|\,\text{LOC}\,|\,\text{SUBCAT} & \left\langle[\,]\right\rangle \\ \text{DTRS}\,|\,\text{HEAD-DTR}\,|\,\text{SYN}\,|\,\text{LOC} & \begin{bmatrix}\text{LEX} & + \\ \text{HEAD}\,|\,\text{INV} & -\end{bmatrix}\end{bmatrix}$$

(3) Rule 3

$$\begin{bmatrix} \text{SYN}\,|\,\text{LOC}\,|\,\text{SUBCAT} & \langle\rangle \\ \text{DTRS}\,|\,\text{HEAD-DTR}\,|\,\text{SYN}\,|\,\text{LOC} & \begin{bmatrix}\text{LEX} & + \\ \text{HEAD}\,|\,\text{INV} & +\end{bmatrix}\end{bmatrix}$$

(4) Rule 4

$$\begin{bmatrix}\text{DTRS} & {}_{\textit{head-adjunct-structure}}[\,]\end{bmatrix}$$

Rules 2 and 3 are self evident: rule 2 simply states that a phrase whose head is lexical and whose SUBCAT has length 1 is grammatical (complement attachment); rule 3 allows for structures where both the subject and the complements are retrieved "one shot" (,e.g. English inverted structures). Rules 1 and 2 are self evident. We simply note that they license unary projections from the lexicon, a fact which has been recently at the center of some debate and which is likely to weaken the spirit of the non-configurational reinterpretation of X-bar theory provided by Pollard and Sag.

Rule 4 deserves more attention as it constitutes a first attempt to handle the attachment of adjuncts in HPSG. It simply states that a structure containing a HEAD-DTR and some ADJ-DTRS is well formed. Obviously, such a rule needs to be completed by the following principle (here we provide the simplest formulation of PS87, as we are not interested in the problem of extraposed adjuncts):

(5) Adjuncts Principle

$$_{\textit{constituent-structure}}[\,] \Rightarrow \begin{bmatrix} \text{HEAD-DTR}\,|\,\text{SYN}\,|\,\text{LOC}\,|\,\text{HEAD}\,|\,\text{ADJUNCTS} & \boxed{1} \\ \text{ADJ-DTRS } \boxed{2} & \end{bmatrix}$$

CONDITIONS: $\forall$ X$\in\boxed{2}$ $\exists$ Y$\in$ $\boxed{1}$ such that SYNTAX(X)=Y

The reason why such a principle is not stated directly in the rule is that, in this way, it can also apply to structures which are not of a type *head-adjunct-structure*. For instance, it will allow for complements interleaved with adjuncts. More interestingly, (5) requires a major extension of the basic HPSG formalism, in that it makes use of a relational constraint. In other words, " a *relational* dependency holds among the values of two or more paths in a feature structure" (p.163). This is apparently the only case of relational dependency in PS87, but, as we will see, successive works have made heavy use of this kind of annotation, in such a way that it is now considered one of the standard tools for HPSG descriptions. Finally, the rule makes use of quantification over subparts of feature structures (the members of ADJ-DTRS), a non trivial move, under the standard HPSG assumptions. However, we will analyze such a quantificational format when describing the Subcategorization principle, which makes an analogous use of quantification over feature structures.

Linear Precedence Statements

Following the practice of GPSG, PS87 adopt the distinction between immediate dominance principles and linear precedence statements. Such a distinction, while extremely useful for formulating interesting linguistic generalizations, seems to be rather irrelevant from a computational point of view, as most of the known formalisms do not allow for the separation between structural and linear knowledge. Even those formalisms which allow for such a separation, such as ALEP, are not able to capture the intended use of LP statements, as they can only implement "rule bounded" LP statements; whereas, in the original spirit, LP statements should serve as rule independent ordering constraints.

The reason for such a scarce use of LP constraints in computational linguistics lies, in part, in their poorly understood formal properties. Indeed, in the formulation which is provided in PS87, they should apply conjunctively as *filters*: if two objects A and B match the description A' B' of an LP, then the relevant order must be satisfied. The point, here is that it is not clear at all what this matching should be, whether unification or, as in [Gazdar *et al.*, 1985], subsumption. Moreover, the heavy use of implicational constraints and universal quantification makes the implementation of this kind of tool extremely difficult, at least under a non trivial interpretation.

Subcategorization

The subcategorization principle is formulated in PS87 in the following way:

(6) Subcategorization Principle

$$\begin{bmatrix} \text{DTRS} & _{\textit{headed-structure}}[\,] \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} \text{SYN}\,|\,\text{LOC}\,|\,\text{SUBCAT} & \boxed{2} \\ \text{DTRS} & \begin{bmatrix} \text{HEAD-DTR}\,|\,\text{SYN}\,|\,\text{LOC}\,|\,\text{SUBCAT} & \text{append}(\boxed{1},\boxed{2}) \\ \text{COMP-DTRS } \boxed{1} & \end{bmatrix} \end{bmatrix}$$

However simple this principle may appear, it presents some formal features which are worthy of investigation. Firstly, the function **append** is used, which is not allowed in all the formalisms under consideration in this project. Secondly, and most importantly, the tag $\boxed{1}$ stands for *the concatenation of the synsem values of all the members of* $\boxed{1}$. Thus the real implementation of the subcategorization principle, as it stands, involves some use of relational constraints. For instance:

(7) Subcategorization Principle

$$\begin{bmatrix} \text{DTRS} & _{\textit{headed-structure}}[\,] \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} \text{SYN} \mid \text{LOC} \mid \text{SUBCAT} & \boxed{2} \\ \text{DTRS} & \begin{bmatrix} \text{HEAD-DTR} \mid \text{SYN} \mid \text{LOC} \mid \text{SUBCAT} & \text{append}(\boxed{1},\boxed{2}) \\ \text{COMP-DTRS } \boxed{1} & \end{bmatrix} \end{bmatrix}$$

$$\text{CONSTRAINT: } \forall \text{ X } [\textbf{member } (\text{X},\boxed{1}) \leftrightarrow \textbf{member}(\text{SYNSEM(X)},\boxed{2})]$$

Now, it is not easy to see how such a constraint could be implemented in a declarative fashion, given the use it makes of universal quantification. Obviously, one could define a particular type of **append** which takes care of "extracting" the relevant *synsems*. Such a function should, however, be implemented directly in the programming language, as we will see in the next section, a situation which is excluded *a priori* by the premises of this project, according to which the functionalities of the programming language should not be accessible from the level of the linguistic description (grammar writing).

Fortunately, limited implementations can, in fact, be conceived in almost any framework, assuming the following restrictions:

- The lenght of the SUBCAT list is considered bounded, as every predicative item can subcategorize for a limited number of complements. Under this view the subcategorization principle could be implemented disjunctively by specifying all the possible alternatives. Such a solution, however, could not work with approaches which make use of *argument composition* [Hinrichs and Nakazawa, 1989] (see below). Indeed, since this technique is based on the idea of appending the SUBCAT lists of a governed predicate to the SUBCAT list of the matrix predicate, and since there is no theoretical limit to the lenght of a head-complement chain, the lenght of a SUBCAT list cannot be fixed *a priori*.

- The grammar is structured in such a way that only binary projections are allowed. In this case, the problem of universal quantification would not arise, because, for the subcat principle to be satisfied, it would be enough that, informally, the synsem of the non head daughter be a member of the SUBCAT list of the head daughter.

A final point concerns the treatment of complement optionality in PS87. To mark the optionality of certain complements, such as the direct object of the verb *to eat*, a notation is adopted, which makes use of parentheses to mark the complement which need not to be realized. For instance, the lexical entry for *eat* would look like the following:

$$(8) \quad \begin{bmatrix} \text{SYN} \mid \text{LOC} \mid \text{SUBCAT} & \left\langle (\text{NP}_{\boxed{1}}),\ \text{NP}_{\boxed{2}} \right\rangle \\ \text{SEM} \mid \text{CONT} & \begin{bmatrix} \text{RELN} & \text{eat} \\ \text{EATER} & \boxed{2} \\ \text{EATEN} & \boxed{1} \end{bmatrix} \end{bmatrix}$$

It goes without saying that such a notation is simply shorthand for a disjunction, so that it can be implemented without any major problems. However, such a notation is in fact not implemented in any "mainstream" framework. Nor, to the best of our knowledge, has it been employed in any major linguistic work in the HPSG field. Moreover, such notation has been proven inadequate for handling certain phenomena concerning optionality. For instance, [Kiss, 1991b] shows that it is inadequate for capturing the simple generalization according to which a German verb such as *antworten* ('to answer') can show any kind of optionality *provided at least one complement is present*. [Dini, to appear] also shows that a pure syntactic treatment of optionality, as assumed in PS87, cannot account for certain basic cases of complement optionality in languages such as Italian and French.

Semantics

The semantic principle of PS87 is expressed in the following form:

$$(9)\ \begin{bmatrix}\text{DTRS} & {}_{\textit{headed-structure}}[\,]\end{bmatrix} \Rightarrow$$
$$\begin{bmatrix} \text{SEM} & \begin{bmatrix}\text{CONT} & \text{successively-combine-semantics}(\boxed{1},\boxed{2})\\ \text{INDICES} & \text{collect-indices}(\boxed{3})\end{bmatrix}\\ \text{DTRS} & \boxed{3}\begin{bmatrix}\text{HEAD-DTR}\,|\,\text{SEM}\,|\,\text{CONT} & \boxed{1}\\ \text{COMP-DTRS} & \boxed{2}\end{bmatrix}\end{bmatrix}$$

Here we will not evaluate the linguistic adequacy of such a principle, which has been almost completely replaced in [Pollard and Sag, 1994b]. We simply note that, from a formal point of view, it presupposes both union operations and a function which, as in the case of the subcat principle, cannot be expressed declaratively, but needs to be coded directly in some programming language. For instance, in PS87, the following formulation of **successively-combine-semantics** is provided:

```
successively-combine-semantics(A,L)0
       if length(L)=0
       then A
       else
              successively-combine-semantics
              (combine-semantics (A, SEM|CONT of first(L)), rest(L)).

combine-semantics (A,B) =
       if
              A has type 'circumstance' and
              B has type 'quantifier'
       then [QUANT =B & SCOPE=A]
       else A
```

In these cases, however, the adoption of hard wired functions can also be avoided, either by assuming that the number of complements is bounded or by designing a grammar based on a binary branching concept.

Lexicon and Morphology

The organization of the lexicon in PS87 relies on a hierarchy of lexical types:

> (...) language users have knowledge of a taxonomic system of lexical *types*. That is, lexical information is organized on the basis of relatively few word types arranged in cross-cutting hierarchies which serve to classify all words on the basis of shared syntactic, semantic and morphological properties. By factoring out information about words which can be predicted from their membership in types, the amount of idiosyncratic information that needs to be stipulated in individual lexical signs is dramatically reduced.
> (PS87, p. 192)

The point here is: "how powerful should a type theory be in order to encode all the relevant linguistic generalization?"

First of all, the type system has to be at least *well-typed.* This is indeed the formal translation of the concept of "making sense", which is adopted, for instance, during the descriptions of the attribute that the type *head* should have:

> For of the head feature listed, only MAJOR and PRD are appropriate for all HEAD values. Of the others, NFORM and CASE make sense only for HEAD values of nouns.
> (p. 200)

In fact, it is important to note that the use that PS87 make of the concept of well-typedness is slightly weaker than the use assumed in systems such as ALEP or TDL (under the option `*check-welltypedness*=t`) and closer in spirit to the one assumed in [Carpenter, 1992a] and implemented in ALE. The crucial difference is that, whereas in ALEP and TDL every time an attribute is used in a linguistic description the type by which it is introduced must be also specified, ALE does not enforce such such a requirement. The lexical encoding format of PS87 works in the latter way. For instance, the information specific to the noun *dog* is described as:

(10)

$$\begin{bmatrix} \text{PHON} & \text{dog} \\ \text{SEM} \mid \text{CONT} \mid \text{IND} \mid \text{REST} \mid \text{RELN} & \text{dog} \end{bmatrix}$$

Such a description would be well typed under the ALE interpretation of well-typedness, but it amounts to a well typedness violation in a framework such as ALEP or TDL(under the option `*check-welltypedness*=t`).

The second important point concerning the organization of the type hierarchy in PS87 is the use of *deep types*: a subtype can be specified not only by subtyping the values of the attributes which it introduces, but also by subtyping the values of attributes which can be arbitrarily nested within that type. For instance:

(11)

$$\mathit{major\text{-}lexical\text{-}sign}\,[\,]$$

$$\begin{bmatrix} \text{SYN} \mid \text{LOC} \mid \text{HEAD} & \text{nhead} \\ \text{SEM} \mid \text{CONT} & \text{indexed-obj} \end{bmatrix}_{\mathit{noun}} \qquad \begin{bmatrix} \text{SYN} \mid \text{LOC} \mid \text{HEAD} & \text{vhead} \\ \text{SEM} \mid \text{CONT} & \text{basic-circumstance} \end{bmatrix}_{\mathit{verb}}$$

This feature is worthy of being mentioned, since not every formalism allows for this kind of specification, even though it can be emulated in a number of ways.

Finally, it should be noted that in PS87 explicit mention of the concept of *instance* is made:

> In type subsumption graphs, we used dotted lines to connect "instances" to the type they belong to. In terms of our formalism, of course, instances are just minimal types, i.e. types for which no finer distinctions exists. In our system of lexical subtypes the instances will just be individual lexical signs

Thus, the distinction between types and instances is conceptual in nature, as words could well be subtypes, and nothing would change in the formal system. Such a conceptual distinction is implemented in some of the formalisms under analysis: for instance, TDL makes use of instances exactly along the same lines drawn in PS87.

We will now take into account lexical rules, which play a substantial role in determining the way in which the lexicon is structured according to PS87. They are conceived to handle both morphological inflection and lexical alternation (valency changing alternations, semantic alternations, and so on). For instance, the following lexical rule is assumed to take care of third singular inflection in English:

(12)

$$\begin{bmatrix} \text{PHON} & \boxed{1} \\ \text{3RDSNG} & \boxed{2} \\ \text{SYN} \mid \text{LOC} \mid \text{SUBCAT} & \boxed{3} \\ \text{SEM} \mid \text{CONT} & \boxed{4} \end{bmatrix}_{\mathit{base}} \rightarrow \begin{bmatrix} \text{PHON} & \text{f}_{3RDSNG}(\boxed{1},\boxed{2}) \\ \text{SYN} \mid \text{LOC} \mid \text{SUBCAT} & \boxed{3} \\ \text{SEM} \mid \text{CONT} & \boxed{4} \end{bmatrix}_{\mathit{3rdfng}}$$

On the formal side, a number of issues concerning lexical rules are still unclear. Firstly, it is not clear *why* they should be needed as an independent formal tool, besides "conceptual necessity". Indeed, their job can be fully emulated, either through the use of unary projections or, as in

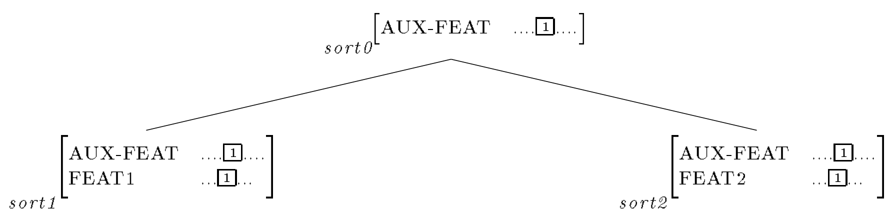
[Copestake and Briscoe, 1992] as special types with an INPUT and an OUTPUT attribute. Secondly, their notation is still a little unprecise. PS87 seem, indeed, to assume that whatever is not mentioned in the left hand side of a lexical rule remains unchanged in the right hand side. But then, if something is mentioned *and* changed which is structure shared with some unmentioned attribute of the input object, what should the output be? There are several answers, none of which seem to be fully satisfactory. Thirdly, there could be applications of lexical rules which end up with an infinite recursion. For instance, to handle word formation in agglutinating languages (or, more simply, *Vor*-prefixation in German), recursive lexical rules are needed. However, while there do exist well-known techniques to prevent type recursion, it is not clear at all how loops of lexical rules could be stopped. In particular, if they are actually implemented as unary rewrite rules. Fourthly, conceptually, lexical rules tend to introduce a parameter of directionality which has very poor linguistic motivations: sometimes it is not clear why certain forms should be considered *more primitive* than the forms that they generate through lexical rules. For instance, considering the well-known semantic alternation building-institution one might want to avoid stating that one semantic realization has priority with respect to the other (cf. also [Krieger and Nerbonne, 1993] and [Kathol, 1994] for criticisms of the lexical rule approach) .

In spite of all these limitations, lexical rules have encountered great appreciation among linguists, and their use has become customary in successive works. Indeed, they are an extremely intuitive tool for expressing linguistic generalizations of various natures, and also those authors which contest their use in favour of type inference mechanisms, usually agree on this fact.

Lexicon and Morphology: various authors

Given the limitations of lexical rules we just mentioned, a number of researchers have tried to eliminate of them, by assuming mechanisms of type inference over a hierarchically structured lexicon.

The clearest formulation of this attempt is provided in [Kathol, 1994] (although this paper has been in circulation for quite a while):

> Whenever a lexical rule relates two sorts that have certain information in common, as in:
>
> $${}_{\mathit{sort1}}\begin{bmatrix}\text{FEAT1} & \ldots\boxed{1}\ldots\end{bmatrix} \Rightarrow {}_{\mathit{sort2}}\begin{bmatrix}\text{FEAT2} & \ldots\boxed{1}\ldots\end{bmatrix}$$
>
> It is possible two relate the two sorts instead by a common supersort, schematically:
>
> 
>
>
> The key difference is that whatever kind of "transformation" the lexical rule performs, the corresponding effect has to be modelled internally to the subsorts by stating in which way the information under AUX-FEAT is related to the way it is eventually used.
> (p. 262)

It is clear that in order for this treatment to be effective the system has to be sort resolved. Consider, indeed, that lexical entries, under this view, are assigned the less specific type, i.e. the type *sort0* above: whenever the syntactic or semantic context allows for a possible specification either into *sort1* or *sort2*, such an inference *has* to take place, otherwise the remaining constraints associated to these subsorts cannot be computed. It is clear, then, that systems which are intrinsically non sort resolved are poorly suitable for the implementation of these kinds of approaches, unless a disjunctive type specification is used, rather than a subtyping mechanism (cf. [Dini and Busa, 1994]).

Furthermore, [Kathol, 1994] notes that the crucial difference between type inference mechanisms of the kind described above and lexical rules is the property of recursiveness, i.e. whereas lexical rules can feed each other, the possible specifications of an underspecified type have to be predicted during the phase of design of the type system. This could pose some problem when handling recursive phenomena, where the hierarchical lexicon view seems to be inadequate.

An analogous attempt to eliminate lexical rules is described in [Krieger and Nerbonne, 1993], where lexical rules are substituted either by distributed disjunction or by schemata which are very close to the one which governs the phrasal organization, with the single difference that they apply in the lexicon rather than in syntax.

Distributed disjunction (a formal tool which is available to many of the frameworks under consideration, and which anyway can be emulated through compilation into many distinct lexical entries) is used to handle inflectional paradigms. The following description, for instance, represents the paradigm of weak verbs in German

(13)

$$\begin{bmatrix} \text{MORPH} & \begin{bmatrix} \text{STEM} & \boxed{2} \\ \text{ENDING} & \boxed{3}\left\{\$\,1\ \text{“e”}, \text{“st”}, \text{“t”}, \text{“n”}, \text{“t”}, \text{“n”}\right\} \\ \text{FORM} & \boxed{2}\ \&\ \boxed{3} \end{bmatrix} \\ \text{SYN}\,|\,\text{LOCAL}\,|\,\text{HEAD}\,|\,\text{AGR} & \left\{\$\,1 \begin{bmatrix}\text{PER} & \text{1st}\\ \text{NUM} & \text{sg}\end{bmatrix}, \begin{bmatrix}\text{PER} & \text{2nd}\\ \text{NUM} & \text{sg}\end{bmatrix}, \ldots \begin{bmatrix}\text{PER} & \text{3rd}\\ \text{NUM} & \text{spl}\end{bmatrix}\right\} \end{bmatrix}$$

In contrast, morphological schemata are supposed to handle derivational phenomena. The basic assumption is that the suffix serves as the head of a complex construction, whereas the base stem is a kind of morphological complement. Most principles of the grammar, such as the Head Feature Principle, the Subcategorization Principle, the Semantic principle, are active in morphology (in a slightly modified formulation) to ensure the well-formedness of complex words. For instance, the feature structure associated with the German adjective *lesbar* ('readable') would be as follows:

(14)

$${}_{\textit{bar-comp-A}}\begin{bmatrix} \text{MORPH}\,|\,\text{FORM}\ \boxed{2}\ \&\ \boxed{3} & \\ \text{SYN}\,|\,\text{LOC} & \begin{bmatrix} \text{LEX} & + \\ \text{SUBCAT} & \text{construct-subcat}(\boxed{5}) \\ \text{HEAD} & \boxed{3} \end{bmatrix} \\ \text{SEM}\ \ \boxed{4} & \\ \text{MORPHS}\ \ \boxed{5} & \begin{bmatrix} \text{HEAD-MORPH} & \begin{bmatrix} \text{MORPH}\,|\,\text{FORM} & \boxed{2}\ \text{“bar”} \\ \text{SYN}\,|\,\text{LOC} & \begin{bmatrix} \text{SUBCAT} & \boxed{6}\,[\,]_{\boxed{7}} \\ \text{HEAD} & \boxed{3}\,[\text{MAJ A}] \end{bmatrix} \\ \text{SEM} & \begin{bmatrix} \text{OPERATOR} & \diamond \\ \text{SCOPE} & \boxed{7} \end{bmatrix} \end{bmatrix} \\ \text{COMP-MORPH}\ \ \boxed{6} & \begin{bmatrix} \text{MORPH}\,|\,\text{STEM} & \boxed{1}\ \text{“les”} \\ \text{SYN}\,|\,\text{LOC}\,|\,\text{LEX} & + \\ \text{SEM} & \boxed{7} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

It is important to observe that, under the authors' view, structures of this kind are not the output of a real application of phrase structure rules on decomposed morphemes, but they are obtained through the exploitation of a lexical hierarchy, where the various principles are nodes in this hierarchy. In (15) we report the lexical entry associated with an adjective like *lesbar*. Its full fledged form in (14) is simply obtained as an effect of type inference under a sort resolved system.

(15) lesbar = bar-comp-A $\wedge$ [MORPHS|COMP-MORPH *lesen*]

It should also be noted, that this system *admits* recursive derivations, as in the case of *Vor*-prefixation. How this can be achieved will be explained in the next paragraph, where we describe a somewhat simpler system of morphological analysis through mechanisms of lexical inheritance, which *does not* assume that grammatical principles are active at the lexical level.

In [Riehemann, 1993] and [Riehemann, 1994] Krieger and Nerbonne's view of morphological derivation is criticized on the basis that (i) it introduces some redundancy, (ii) it makes use of mechanisms such as lexical default inheritance to handle exceptions. The system proposed in [Riehemann, 1993] and [Riehemann, 1994] is thus a simpler version of Kreiger and Nerbonne's one, in that "derivational processes" are encoded directly as types in the hierarchy. For instance, the type responsible for *bar*-suffixation is the following:

(16)

$$
{}_{\textit{reg-bar-adj}}\left[\begin{array}{ll}
\text{PHON} & \boxed{1} + \text{bar} \\
\text{MORPH-B} & \left\langle {}_{\textit{trans-verb}}\left[\begin{array}{ll}
\text{PHON} & \boxed{1} \\
\text{SYN | LOC} & \left[\begin{array}{ll}
\text{CAT | VAL | COMPS} & <\text{NP[acc]}:\boxed{2}\ .\ \boxed{3}> \\
\text{CONT | NUC} & \left[\begin{array}{ll} \text{ACT} & \\ \text{UND} & \boxed{2} \end{array}\right]
\end{array}\right]
\end{array}\right] \right\rangle \\
\text{SYN | LOC} & \left[\begin{array}{ll}
\text{CAT} & \left[\begin{array}{ll}
\text{HEAD} & \text{adj} \\
\text{VAL} & \left[\begin{array}{ll} \text{SUBJ} & <\text{NP}:\boxed{2}> \\ \text{COMPS} & \boxed{3} \end{array}\right]
\end{array}\right] \\
\text{CONT | NUC} & \left[\begin{array}{ll} \text{RELN} & \diamond \\ \text{ARG1} & \boxed{2} \\ \text{ARG2} & \boxed{4} \end{array}\right]
\end{array}\right]
\end{array}\right]
$$

Since, in this system, words *are* types, recursive derivations are possible, as in Krieger and Nerbonne's system. To see this, assume that words, in particular, are subtypes of a type *stem*. The type *stem* is in turn a value of the attribute STEM (which, for the sake of simplicity, we use instead of the list valued MORPH-B in (16)) and it *introduces* an attribute STEM. Under this view, as a word enters into the parsing process it has a maximally unspecified synsem. As soon as constraints are added to this word a compatible value of STEM would be inferred (for instance, a lexical type which has a corresponding phonology). However, since the value of STEM is in turn of type *stem*, a mechanism can be introduced, so as to produce a recursive structure. For instance, the following fragment could trigger a process of recursive type specification:

(17) *stupid-lexical rule* $<$ *stem*
John $<$ *stem*

$$
{}_{\textit{stupid-lexical-rule}}\left[\begin{array}{ll}
\text{STEM} & \left[\begin{array}{ll} \text{PHON} & \boxed{2} \\ \text{SYNSEM} & \boxed{1} \end{array}\right] \\
\text{PHON} & \mathbf{cons}(\mathbf{first}(\boxed{2}),\boxed{2}) \\
\text{SYNSEM} & \boxed{1}
\end{array}\right]
$$

$$
{}_{\textit{John}}\left[\begin{array}{ll}
\text{PHON} & \left\langle \text{John} \right\rangle \\
\text{SYNSEM} & \text{john-syns}
\end{array}\right]
$$

Under the assumption of a sort resolved system, objects of the following kind could be deduced:

(18)

$$
{}_{\textit{stupid-lexical-rule}}\left[\begin{array}{ll}
\text{STEM} & {}_{\textit{stupid-lexical-rule}}\left[\begin{array}{ll}
\text{STEM} & {}_{\textit{stupid-lexical-rule}}\left[\begin{array}{ll}
\text{STEM | STEM | ... | STEM} & \text{john} \\
\text{PHON} & \left\langle \text{John .... John} \right\rangle \\
\text{SYNSEM} & \boxed{1}
\end{array}\right] \\
\text{PHON} & \left\langle \text{John John .... John} \right\rangle \\
\text{SYNSEM} & \boxed{1}
\end{array}\right] \\
\text{PHON} & \left\langle \text{John John John .... John} \right\rangle \\
\text{SYNSEM} & \boxed{1}
\end{array}\right]
$$

In conclusion, it is not clear whether we want recursive systems such as the one described in (17), where some special control device has to be delivered in order to stop recursion (cf [Krieger and Nerbonne, 1993], p.33). It is a fact, however, that lexical rules and type inference mechanisms seem to be almost equivalent, since the latter, according to the conception of the lexicon as a set of types, also allows for recursive derivations.

Towards the Valence Attribute

In a number of papers ([Borsley, 1987a][Borsley, 1987b][Borsley, 1989]) it has been argued that the SUBCAT list of PS87 should be split in such a way that a particular attribute (SUBJ) takes care of encoding the category representing the so called *external argument*, whereas the SUBCAT list should contain only the remaining complements. Under this proposal, the subcategorization requirements associated to the lexical entry of a verb such as *admires* would be the ones in (20), rather than (19), as the theory presented in PS87 would predict:

(19) $$\begin{bmatrix}\text{SUBCAT} & \left\langle \text{NP, NP[3sg]} \right\rangle\end{bmatrix}$$

(20) $$\begin{bmatrix}\text{SUBJ} & \left\langle \text{NP[3sg]} \right\rangle \\ \text{SUBCAT} & \left\langle \text{NP} \right\rangle\end{bmatrix}$$

Here we will not enter into the question of the linguistic motivations for such a choice: as a matter of fact such a distinction seems now to be widely accepted by the HPSG linguistic community. Moreover, in [Pollard and Sag, 1994b] (henceforth PS94) such a distinction has become a part of the "official" HPSG theory. Since the merit of this revision is to be ascribed mainly to works by Borsley, which are anterior to the beginning of the project, we will antepose here the discussion concerning valence features which is carried out in chapter 9 of PS94.

First of all, in order to avoid possible confusion, in PS94, chapter 9, the feature SUBCAT of [Borsley, 1987b] becomes COMPS. Secondly, a third valence feature, SPR, is introduced, containing what in GB terms is defined as a *specifier* : thus, basically, the *determiner* of NPs, and the *measure phrase* or *degree expression* of adjectives, adverbs and comparative particles. These three valence features are grouped together under the attribute VALENCE. The Subcat Principle of PS87 is thus replaced in PS94 by the Valence Principle:

(21) In a headed phrase, for each valence feature F, the F value of the head daughter is the concatenation of the phrase's F values with the list of SYNSEM values of the F-DTRS value.

Under this new formulation, the principle, besides retaining the problems connected to the operation of concatenation of SYNSEM values of a list of arbitrary length, introduces a further formal complexity, as it makes use of *variables* over attributes (F). As far as we know, none of the implemented formalisms is able to handle statements such as the one in (21), which should be considered nothing but an informal way of expressing linguistic generalizations. As a matter of fact, the Valence principle should be reduced to a disjunctive principle along the following lines (we assume a binary branching grammar, as we described in a previous section):

(22)
$$\begin{bmatrix}\text{SYN}\,|\,....\,|\,\text{VALENCE} & \begin{bmatrix}\text{SUBJ} & \boxed{1} \\ \text{COMPS} & \boxed{2} \\ \text{SPR} & \boxed{4}\end{bmatrix} \\ \text{HEAD-DTR}\,|\,...\,|\,\text{VALENCE} & \begin{bmatrix}\text{SUBJ} & \text{cons}(\,\boxed{3}\,\boxed{1}) \\ \text{COMPS} & \boxed{2} \\ \text{SPR} & \boxed{4}\end{bmatrix} \\ \text{SUBJ-DTR}\ |\text{SYNSEM} & \boxed{3}\end{bmatrix} \vee$$

$$\left[\begin{array}{ll} \text{SYN}\,|\,....\,|\,\text{VALENCE} & \left[\begin{array}{ll}\text{SUBJ} & \boxed{1}\\ \text{COMPS} & \boxed{2}\\ \text{SPR} & \boxed{4}\end{array}\right] \\ \text{HEAD-DTR}\,|\,...\,|\,\text{VALENCE} & \left[\begin{array}{ll}\text{SUBJ} & \boxed{1}\\ \text{COMPS} & \text{cons}(\,\boxed{3}\,\boxed{2})\\ \text{SPR} & \boxed{4}\end{array}\right]\\ \text{COMP-DTR}\,|\,\text{SYNSEM} & \boxed{3}\end{array}\right] \vee$$

$$\left[\begin{array}{ll} \text{SYN}\,|\,....\,|\,\text{VALENCE} & \left[\begin{array}{ll}\text{SUBJ} & \boxed{1}\\ \text{COMPS} & \boxed{2}\\ \text{SPR} & \boxed{4}\end{array}\right] \\ \text{HEAD-DTR}\,|\,...\,|\,\text{VALENCE} & \left[\begin{array}{ll}\text{SUBJ} & \boxed{1}\\ \text{COMPS} & \boxed{2}\\ \text{SPR} & \text{cons}(\,\boxed{3},\,\boxed{4})\end{array}\right]\\ \text{SPR-DTR}\,|\,\text{SYNSEM} & \boxed{3}\end{array}\right]$$

Note, also, that, in spite of its triviality, this implementation could only be suited for a system which assumes welltypedness. Otherwise, three applications of the Valence Principle would be enforced for every level of phrasal projections, with the consequence of having a massive overgeneration of linguistically unmotivated structures. Moreover, a branch of the alternation, which is not mentioned in (22) should take care of structures where a FILLER-DTR, a MARKER-DTR or an ADJUNCT-DTR is present, as in these cases the Valence principle should apply vacuously. Apparently, such a “vacuous” alternation could not be avoided even by adopting an implicational constraint of the kind: *if a sign has a SUBJ-DTR.... then...*, which, more formally, amount to statements of the kind *if a sign is of type ‘head-subj-struc’.... then....*

Argument Composition

Since [Hinrichs and Nakazawa, 1989] a number of researchers have adopted a particular technique labelled *argument composition*. Such a technique has been exploited in analyses of German ([Hinrichs and Nakazawa, 1989], [Hinrichs and Nakazawa, 1990], [Hinrichs and Nakazawa, 1994], [Nerbonne, 1994], [Pollard, 1994]), French ([Miller and Sag, 1993], [Aranovich *et al.*, 1994], [Miller and Sag, 1993], [Abeillé *et al.*, to appear]), Dutch ([Rentier, 1994], [van Noord and Bouma, 1994]) and Italian ([Monachesi, 1993a], [Monachesi, 1993b], [Monachesi, 1993c], [Monachesi, to appear]). The core of the approach is based on the assumption that certain words (most notably auxiliaries and modal verbs) are able to “raise” the subcategorization properties of some phrase (or lexical item) of which they are head. For instance, an auxiliary which subcategorizes for a verb with [SUBCAT<NP, PP>] would be able to subcategorize, in turn, for an NP and a PP. This effect is obtained by retaining the subcategorization principle in its original formulation, and imposing to the relevant lexical entries a structure sharing of lists along the following lines:

(23) $${}_a\left[\text{SUBCAT}\ \ \boxed{1} \oplus \left\langle\ {}_b\left[\text{LOCAL}\,|\,\text{CAT}\,|\,\text{SUBCAT}\ \ \boxed{1}\right]\right\rangle\right]$$

The crucial point here is that the SUBCAT list of the sign b is appended *before* the obligatory complement for which a subcategorizes. This means that what is minimally needed to implement this kind of approache is an **append** function over lists, or, for those approaches which make use of sets, an operation of **union**. A poorer function of list manipulation such as **cons**, which is available to a formalism such as ALEP, could not work, unless we renounce the wish to express the fact that the raised complements must be less oblique (in the sense of PS94) than the “original complement”. Under this minimal hypothesis, (23) would reduce to:

(24) $${}_{a}\left[\text{SUBCAT} \quad \mathbf{cons}\left({}_{b}\left[\text{LOCAL} \,|\, \text{CAT} \,|\, \text{SUBCAT} \quad \boxed{1}\right], \boxed{1} \right)\right]$$

This formulation, besides the obvious wrong interactions with certain modules of the grammar, such as linear precedence constraints and binding theory, would also run into trouble when confronted with cases such as the Italian restructuring verbs, where the verb which we labelled with a in (23) can have an autonomous argument structure. We thus conclude that an append (or union) function is absolutely necessary if some formal implementation of "argument composition" approaches is to be provided.

θ-roles

Most works dealing with romance languages, most notably [Beaven, 1990], [Sanfilippo, 1990], [Sanfilippo, 1993] (in a Unification Categorial Grammar framework very close to HPSG), [Balari, 1991], [Sanfilippo:, to appear], [Dini, to appear] have assumed a different structuring of the CONTENT of signs, which makes use of θ roles rather than relations with a fixed arity. Also, certain implemented grammars exploit the notion of θ role lists, such as the Italian grammar developed under the project LSGRAM+(IT). The underlying idea of these approaches is rather simple: the semantic role played by the actors in a situation are not encoded as a value of an attribute, but as a list of roles which are in turn predicates of an event variable and an individual variable. Thus, the sentence *Brutus stabbed Caesar* would be semantically represented as (26) rather than as (25)

(25) $${}_{stab}\begin{bmatrix} \text{STABBER} & \boxed{1} \\ \text{STABBED} & \boxed{2} \end{bmatrix}$$

(26) $$\begin{bmatrix} \text{RELATION} & {}_{stabbing}\left[\text{INDEX} \quad \boxed{3}\right] \\ \text{ROLES} & \left\langle {}_{agent}\begin{bmatrix} \text{ARG1} & \boxed{1} \\ \text{ARG2} & \boxed{3} \end{bmatrix}, {}_{patient}\begin{bmatrix} \text{ARG1} & \boxed{2} \\ \text{ARG2} & \boxed{3} \end{bmatrix}, \right\rangle \end{bmatrix}$$

The list ROLES is not lexically instantiated, but it is built up incrementally through a process that moves θ-roles from a list labelled Θ-DOMAIN into the list ROLES, as soon as the complements to which they are associated are retrieved. The principle in charge of performing such a θ-discharging is labelled *Thematic Principle*. Its simplest formulation is found in [Balari, 1991] (it goes without saying that with the introduction of this principle also other modules of the grammar, such as ID schemata and the Content Principle, would undergo some slight modification):

(27) $$\begin{bmatrix} \text{CONT} & \begin{bmatrix} \text{C-INDEX} \,|\, \Theta\text{-DOMAIN} & \boxed{1} \\ \text{ROLES} \quad \boxed{2} & \end{bmatrix} \\ \text{DTRS} \,|\, \text{HEAD-DTR} \,|\, ... \,|\, \Theta\text{-DOMAIN} & \text{append } (\boxed{1}, \boxed{2}) \end{bmatrix}$$

Here we will not focus on the interactions of this principle with other HPSG modules, nor on the linguistic necessity of having such a principle in a grammar designed to handle languages such as Italian or Spanish (see [Balari, 1991] and [Dini, to appear] for an explanation). The reason why we mention it here is for the use it makes of the **append** function. Note that here even a binary branching grammar could not dispense with the need for using such a function in the grammar. Indeed, since both Italian and Spanish are (partially) free complement order languages, the problem of picking up a single θ-role from a list of arbitrary lenght would persist. This task can only be performed by a **cons** function at the cost of writing very long disjunctions.

2.2.6 Lexical Functional Grammar

See the LFG homepage: http://clwww.essex.ac.uk/LFG.

2.2.7 Tree Adjoining Grammar

For a more detailed description of the TAG formalism and the projects mentioned below, see section 3.2.8.

In 1993, the grammar format in XTAG had been more or less stable for some years. Especially the English grammar had been developed further, starting from the work reported in [Abeillé *et al.*, 1990]. In the same framework, grammar development had begun in Anne Abeillé's group in Paris.

The TAG-GEN system had been ported to various projects. Major new developments had started as VM-GEN in the VerbMobil project. The underlying feature formalism was still the non-monotonic UTAG formalism.

2.3 Mathematical Formalisms

2.3.1 Used Constraint Systems

At the beginning of '93, the basic feature description language included feature and sort constraints. Features are binary, functional predicates, and sorts are unary predicates that are ordered by a sort hierarchy. The formulae in the basic feature logic were composed using conjunction and disjunction.

The sort hierarchy was required to be at least a lower semi-lattice, since the unification of two sorts yields the greatest lower bound. There have also been formalisations requiring the sort lattice to be distributive. Distributivity is needed if the least upper bound of two different feature descriptions is also to be calculated. Sorts are distinguished from types in that there are no definitions for sorts in terms of feature structures themselves, as is the case for types.

On top of these basic feature description languages, several extensions were investigated. For example, [Smolka, 1992] investigated the negation of feature descriptions and showed, that classical negation can be translated into the basic language using disjunction and negated coreferences (= negated equations).

An important extension to feature logic was the introduction of type systems, which came out of feature description templates. A type is a unary predicate similar to sorts. The difference between types and sorts is that types are not only constrained by a type hierarchy, but also by a set of type definitions. The type definitions associate a corresponding feature description to every type. In every model, the interpretation of the types must satisfy the feature descriptions associated with the types. The problem is that a set of type definitions does not define a unique interpretation of the types. The different models can be generated as different fixpoints of a continuous operator. In the literature, both greatest and least fixpoint semantics were investigated [Pollard, 1989; Pollard and Moshier, 1990; Emele and Zajac, 1990b; Emele and Zajac, 1990a].

Other used constraints were set constraints and functional uncertainty. In a formalisation using set constraints, a variable can denote a single element of the domain as well as a set of elements. The additional constraints were multiple-valued features (i.e., features whose value can be sets) [Pollard and Moshier, 1990; Moshier and Pollard, 1993]. Functional uncertainty extends the basic feature description language by allowing constraints of the form xLy, where L is a regular language of paths (= strings of features). This constraint is satisfied if y is the element under some path $p \in L$ of x [Kaplan and Zaenen, 1988].

The use of feature logic as a constraint system for horn-clause deduction was proposed in [Aït-Kaci and Nasr, 1986b] and investigated in detail in [Höhfeld and Smolka, 1988], where a sound and complete deduction system was introduced.

2.3.2 Semantic Foundations

initially, feature descriptions were introduced as purely syntactic objects, together with the unification operations. Later, many different semantic foundations were given. One of the earliest foundations by [Kasper and Rounds, 1990] used a special non-standard logic to interpret feature descriptions. Later, domain-theoretic foundations have been added. Smolka and Johnson considered feature descriptions as formulae in a first-order language (where Smolka used a relational languages, and Johnson used a language having predicates and functions). As a standard model for interpreting feature descriptions, the first-order model consisting of feature graphs was introduced. Another important branch of semantic foundations was initiated by [Reape, 1991] and especially by [Blackburn and Spaan, 1993], who considered feature descriptions as formulae in special modal languages.

In all these logical approaches, it was agreed that unification corresponds to testing satisfiability of conjunction of feature descriptions. For the first-order approaches, it was shown that feature graphs are canonical for satisfiability, i.e., that a feature description is satisfiable if it is satisfiable in the feature graph model.

2.3.3 Decidability Questions

At the beginning of '93, the following decidability results were known:

- unification of purely conjunctive feature descriptions is decidable with a quasi-linear complexity [Aït-Kaci and Nasr, 1986b],
- the satisfiability problem for the existential fragment of feature logic (which is a fragment that includes conjunctive feature descriptions, disjunctive descriptions and negation of feature descriptions) is decidable with an NP-complete complexity (found by [Smolka, 1992] and independently by [Johnson, 1988]; for the modal languages this result was shown in [Blackburn and Spaan, 1993]),
- the satisfiability problem for cyclic-free feature description containing functional uncertainty constraints is decidable [Kaplan and Maxwell III, 1988].

Concerning undecidability, the following was known:

- satisfiability of typed feature description with respect to a type-system under least fixpoint semantics is undecidable [Smolka, 1992],
- satisfiability of functional uncertainty with unrestricted negation is undecidable [Baader *et al.*, 1993],
- satisfiability of feature description with subsumption constraints is undecidable [Dörre and Rounds, 1992],
- Further complexity results were given in [Blackburn and Spaan, 1993], who consider a whole bunch of modal feature description languages; the expressivity of some of these languages was even powerful enough to directly encode principles of HPSG.

2.4 Implemented Formalisms

2.4.1 ALE

ALE, a public domain system written in Prolog, integrates phrase structure parsing and constraint logic programming with typed feature structures as terms. This generalizes both the feature structures of PATR-II and the terms of Prolog II to allow type inheritance and appropriateness specifications for features and values. Grammars may also interleave unification steps with logic program goal calls (as can be done in DCGs), thus allowing parsing to be interleaved with other system components. While ALE was developed to handle HPSG grammars, it can also execute PATR-II grammars, DCG grammars, Prolog, Prolog-II, and LOGIN programs, etc.

Grammars and logic programs are specified using a typed version of Rounds-Kasper attribute-value logic, which includes variables and full disjunction. Programs are then compiled into low-level Prolog instructions corresponding to the basic operations of the typed Rounds-Kapser logic. There is a strong type discipline enforced on descriptions, allowing many errors to be detected at compile-time.

The logic programming and parsing systems may be used independently or together. Features of the logic programming system include negation, disjunction and cuts. It has last call optimization, but does not perform any argument indexing. On the "naive reverse" benchmark, it performed at 1000 LI/s on a DEC 5100 running SICStus 2.1, which is roughly 7% as fast as the SICStus interpreter and 0.7% as fast as the SICStus compiler.

The phrase structure system employs a bottom-up all-paths dynamic chart parser. A general lexical rule component is provided, including procedural attachment and general methods for orthographic transformations using pattern matching or Prolog. Empty categories are permitted in the grammar. Both the phrase structure and logic programming components of the system allow parametric macros to be defined and freely employed in descriptions. Parser performance is similar to that of the logic programming system. In an early HPSG grammar, where feature structures consisted of roughly 100–200 nodes each, a 10 word sentence producing 25 completed inactive edges parsed in roughly two seconds, using SICStus 2.1 on a DEC 5100.

2.4.2 ALEP

Introduction

ALEP, the Advanced Linguiatic Engineering Platform, was designed, implemented, improved and extended in a number of EU-funded projects.

As part of its LRE programme, the CEC has undertaken the development of a generic formal and computational environment, which will be put at the disposal of EC and natio nal R&D projects in relevant areas. By making the ALEP system widely available, the CEC intends to promote synergy between academic and industrial research centres and foster progress towards portability and re-use of research results.

Development of ALEP

The preparation and design of ALEP took place from 1991 to 1992, the prototyping stage in 1992, and the first development cycle from 1992 to 1993.

Several research centres and universities, such as SRI-CRC, UMIST, IAI, CAP GEMINI, SNI, were involved in the preparatory stage ([Alshawi *et al.*, 1991], [IAI *et al.*, 1991], [Devillers *et al.*, 1991]). The development of the final platform has been contracted to BIM, with subcontractors SEMA Group, SRI-CRC and IAI. Cray Systems have been charged with the development of a prototype system as well as maintenance and support services for the final system.

The main features of a single-user version of the system, developed by BIM during the first development cycle are:

- an architecture that is open and modular;

- a comprehensive user environment, including editors, browsers, etc., including a graphical user
- interface;
- linguistic processing tools based upon a unification-based formalism;
- a basic text-handling subsystem;
- an early implementation of the lexical database component.

This first version of the system, referred to as ALEP-1, was released in mid-1993 to a few pilot sites, for early assessment and feedback. The results of this assessment were taken into account in the second development cycle described in section 3.4.5.

Formalism

The basic ALEP formalisms were designed by SRI International Cambridge within the ET-6/1 rule formalism and virtual machine design study [Alshawi *et al.*, 1991]. Several different formalisms are provided for:

- analysis of word form variation (two-level rules)
- syntactic and semantic analysis/synthesis (typed unification grammar)
- transfer-based MT (general transfer rule formalism)

The central analysis and synthesis formalism has a context-free skeleton but does not intend to embody any particular linguistic theory. The formalism was designed to be conservative, 'mainstream', efficient, expressive, declarative, reversible and monotonic. This typed unification grammar has a three level architecture:

- Level 1: simple 'PATR like' terms, constraints applied directly by unification.
- Level 2: notational enrichments which can be compiled into constructs of level 1.
- Level 3: notational enrichments which cannot be compiled into constructs of level 1 and which require additional machinery over and above unification.

Alep formalism is based on the formalisms specified under the EUROTRA-6/1 design study, which have been marginally extended in line with developments under the ALEP-0 prototype system. The formalism is based around a very few simple notations and data structures which meet the design study objective characteristics:

- expressivity
- declarativity
- monotonicity
- efficient implementation
- multi-linguality
- multi-purpose for NLP applications
- easily teachable
- reversibility
- minimal ideological commitment

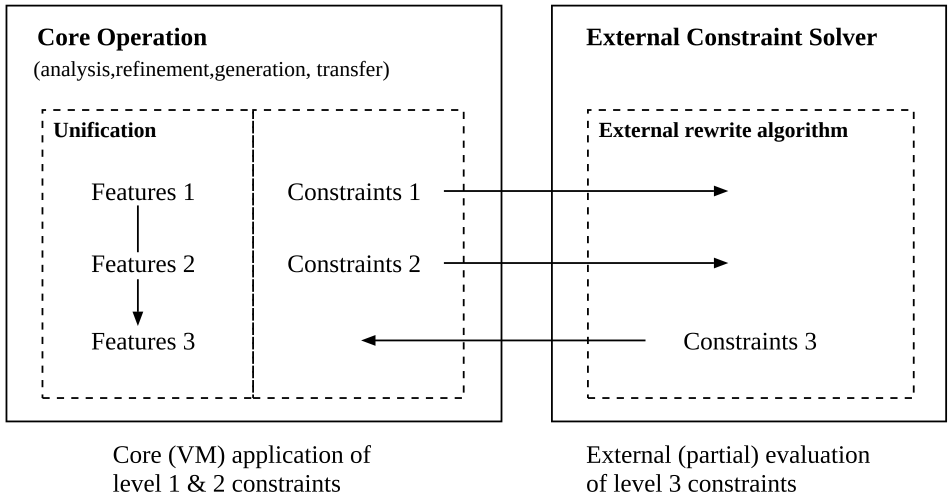


Figure 2.1: Interaction of core unification & external constraint solvers

These aspects of the formalism define the *core* of the formalism. The core formalism alone was never intended to be sufficient for research and development purposes, particularly in terms of characteristics such as expressiveness. However, the core formalism is thought to be sufficient in terms of genericity, that no one particular formalism or application is embodied by the basic principles, and it particularly supports re-usability, prescribed for low cost experimentation and larger scale development.
[2]

The algorithms which perform analysis, refinement, transfer and synthesis have been isolated from the main virtual machine component. This allows selection of an appropriate algorithm for a specific grammar when the system is invoked, and for third party contribution of new algorithms.

The core of the formalism (Level 1) has a conservative design for potential efficiency. As such this is under-expressive for some users. The third level of the formalism allows for unprescribed extension with 'external' constraint systems which operate in parallel, after unification (Figure 2.1) [Simpkins *et al.*, 1993].

The ALEP-0 algorithms now include an experimental set of calls to such an external system. An illustrative sample negation solver is supplied with the system.

2.4.3 CUF

CUF is a theory-neutral universal grammar formalism like PATR-II which has been developed in the ESPRIT-Project DYANA (BRA 3175 and 6852). It is based on defining feature structures and relations over these as encodings of linguistic principles and data. However, it is radically more expressive than conventional grammar formalisms, since it allows the definition of arbitrary recursive relational dependencies without tying recursion to phrase structure rules. Hence, CUF provides the basis for highly integrated processing of linguistic descriptions of different linguistic research areas.

[2] The prototype ALEP-0 implements a large part of the ET-6/1 specification with a few restrictions and differences: (i) The 'concrete' syntax is different (close to that of the ψ terms described in ET-6/1, p220). (ii) The user language (Level 2) is incomplete in not allowing some notations to be used (unordered elements) or not allowing their use at specific points (eg disjunction over sharing). Tuples and specifiers are also not supported. Most of these restrictions can be worked around. No preference mechanism is yet provided. (iii) The type system has been altered to impose a stricter typing. Attributes are associated with either a basic type (list, atom, boolean expression or term) or with the name of another user defined type, itself composed of typed attributes. All types are of fixed arity. The type system also allows for different attributes to have the same name when within different types and allows for simple compilable type hierarchies and inheritance.

The language of CUF uses a syntax especially well suited for a direct description of feature structures similar to Kasper/Rounds logic (feature-matrix notation) combined with the possibility of stating definite clauses over feature terms. Moreover, feature structures are typed, with the types possibly being ordered in a hierarchy. The CUF type discipline allows for an axiomatic statement of global restrictions on the structures in which the program is to be interpreted providing enough redundancy in the descriptions to detect mistakes without burdening the grammar writer with tedious repetitions.

CUF does not predefine any grammar rule formats like PATR's contextfree-based rules or GPSG's ID/LP rules. Instead, the grammar writer is free to define her own rule formats or even grammar architecture. For instance, an architecture based on principles and rules can straightforwardly be implemented.

Fig. 2.2 presents an overview of all kinds of language constructs that can be used to compose a CUF program, including its control part.

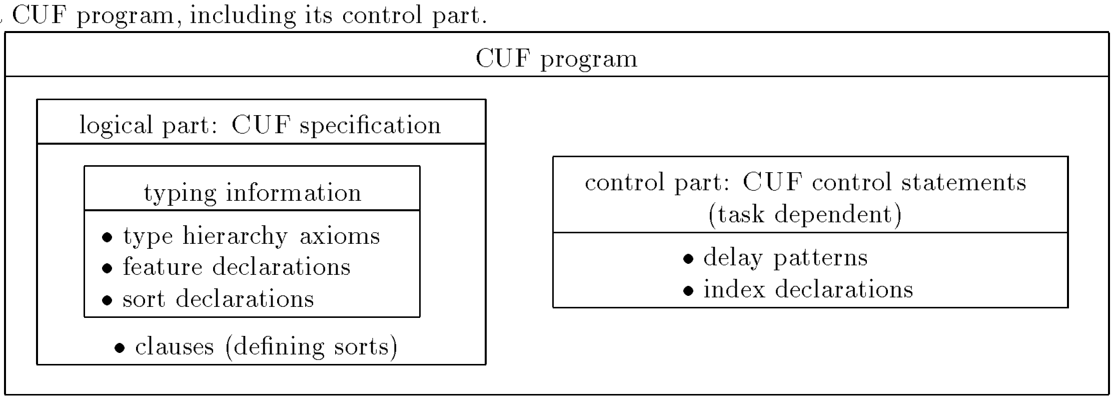


Figure 2.2: Parts of a CUF Program

CUF is an instance of constraint-logic programming (CLP) of the very general Höhfeld/Smolka scheme [Höhfeld and Smolka, 1988]. This provides us not only with a sound and complete proof procedure, but also equips us with the right paradigm to attack the efficiency problems associated with highly modular specifications, as for instance proposed by GB theory. For a more complete description of the CUF language, please refer to [Dörre and Dorna, 1993].

2.4.4 LFG Workbench

See section 6.2.7.

2.4.5 TDL

TDL is a typed feature-based language specifically designed to support highly lexicalized grammar theories like HPSG, FUG, or CUG. TDL offers the possibility to define (possibly recursive) types, consisting of type constraints and feature constraints over the standard connectives $\wedge$, $\vee$, and $\neg$, where the types are arranged in a subsumption hierarchy. TDL distinguishes between *avm types* (open-world reasoning) and *sort types* (closed-world reasoning) and allows the declaration of partitions and incompatible types. Working with partially as well as with fully expanded types is possible, both at definition and at run time. TDL is incremental, i.e., it allows the redefinition of types and the use of undefined types.

TDL is based on UDiNe, a sophisticated feature constraint solver. UDiNe incorporates most of the advanced means that have been described in literature or used in practical system, e.g., distributed disjunctions, negative coreferences, full negation as well as functional and relational constraints.

TDL and UDiNe together provide both a grammar definition environment and a typed run time system which supports lazy type expansion. Efficient reasoning in the system is accomplished through specialized modules.

Motivation

Modern typed unification-based grammar formalisms (like TFS, CUF, or TDL) differ from the early untyped systems like PATR-II in that they highlight the notion of a *feature type*. Types can be arranged hierarchically, where a subtype *inherits* monotonically all the information from its supertypes and unification plays the role of the primary information-combining operation. A *type definition* can be seen as an abbreviation for a complex expression, consisting of type constraints (concerning the sub-/supertype relationship) and feature constraints (stating the appropriate values of attributes) over the standard connectives $\wedge$, $\vee$, and $\neg$. Types can therefore lay foundations for a grammar development environment because they might serve as abbreviations for lexicon entries, ID rule schemata, and universal as well as language-specific principles as is familiar from HPSG. Besides using types as a referential mean, as is also done with templates, types have the following advantages over templates:

- EFFICIENT PROCESSING. Certain type constraints can be compiled into more efficient representations like bit vectors, where a GLB (greatest lower bound), LUB (least upper bound), or a $\preceq$ (type subsumption) computation reduces to low-level bit manipulation. Moreover, types release untyped unification from expensive computation through the possibility of declaring them incompatible. In addition, working with type names only or with partially expanded types, minimizes the costs of copying structures during processing.

- TYPE CHECKING. Type definitions allow a grammarian to declare which attributes are appropriate for a given type and which types are appropriate for a given attribute, therefore, preventing inconsistent feature structures from being written.

- RECURSIVE TYPES. Recursive types give a grammar writer the opportunity to formulate certain functions or relations as recursive type specifications. Working in the *Parsing as Deduction* paradigm enforces a grammar writer to replace the CF backbone through recursive types.

The DISCO Core Engine

The core machinery of DISCO consists of TDL and the feature constraint solver UDiNe. The TDL system is a unification-based grammar development environment and run-time system to support HPSG-like grammars. The DISCO grammar currently consists of more than 700 type specifications written in TDL and is the largest HPSG grammar for German. The UDiNe feature constraint solver is the main processing machinery of DISCO, which has been well-tested in the DISCO environment over years. The typical size of the processed structures reaches more than 1000 nodes and 140 coreferences (which would need up to 185 000 nodes in a PROLOG tree notation).

Both modules communicate through an interface, and this communication mirrors exactly the way an abstract typed unification algorithm works: two typed feature structures can only be unified if the attached types are definitely compatible. This is accomplished by the unifier in that UDiNe handles over two typed feature structures to TDL which returns a simplified form (plus additional information; see Fig. 2.3). The motivation for separating type and feature constraints and processing them in dedicated modules (which again might consist of specialized components as is the case in TDL) is twofold: (i) it reduces the complexity of the whole system, thus making the architecture much clearer, and (ii) leads to a faster system performance because every dedicated module is designed to cover only a specialized task.

Grammars and lexicons can be tested by using the parser of the DISCO system. The parser is a bidirectional bottom-up chart parser, providing a user with parameterized parsing strategies as well as giving him control over the processing of individual rules.

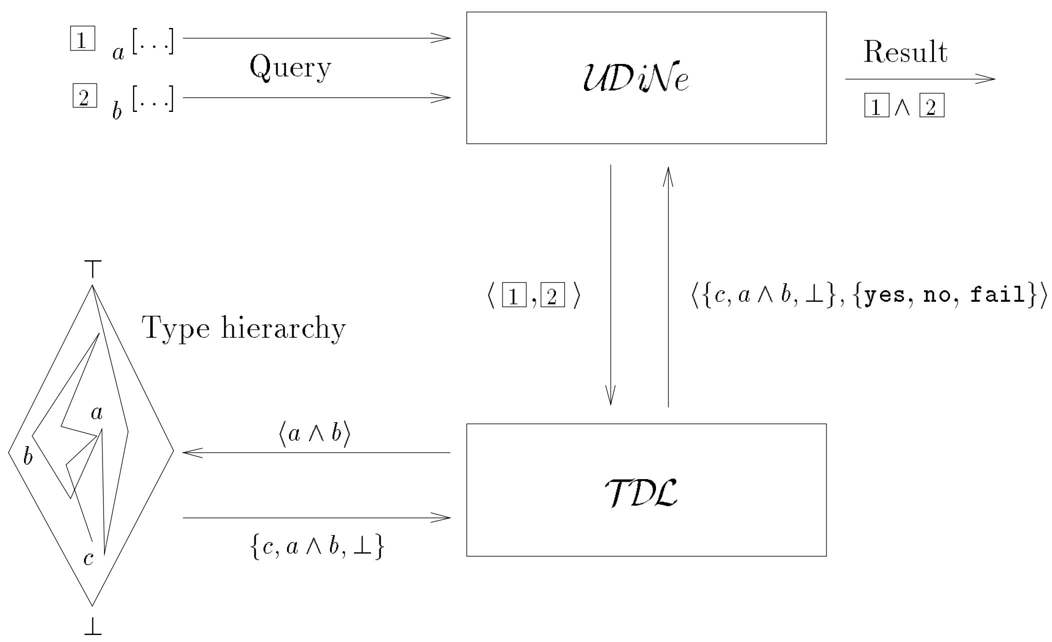


Figure 2.3: **Interface between TDL and UDiNe.** Depending on the type hierarchy and the type of $\boxed{1}$ and $\boxed{2}$, TDL either returns c (c is definitely the GLB of a and b) or $a \wedge b$ (open-world reasoning) resp. $\perp$ (closed-world reasoning) if there doesn't exist a single type which is equal to the GLB of a and b. In addition, TDL determines whether UDiNe must carry out feature term unification (`yes`) or not (`no`), i.e., the return type contains all the information one needs to work on properly (`fail` signals a global unification failure).

The UDiNe Feature Constraint Solver

UDiNe is a modern feature constraint solver that provides distributed disjunctions over arbitrary structures, negative coreferences, full negation and functional constraints. It is the first (and to our knowledge the only) implemented feature constraint solver that integrates both full negation and distributed disjunctions. A relational extension has been implemented, but not yet integrated into the system.

UDiNe works on an internal representation of feature structures, where coreferences are represented using structure sharing. The connection between the internal representation and the (good readable) external one is established by input/output functions. There exists an advanced window-based feature editor called FEGRAMED allowing to define, print and manage feature structures.

During the translation of the external representation into the internal one, several normalization steps are performed. One of these steps is the elimination of full negation in the input structure. We use the method of Smolka (1988), which introduces implicit existential quantification. Using this method, negation can be eliminated if the feature system provides disjunction, negative coreferences and negated atoms/types.

UDiNe uses distributed disjunctions not only as a tool for efficient processing. They are also part of the input syntax, which allows for a very compact representation of the input data. In contrast to other systems using distributed disjunctions, we do not restrict disjunctions to length 2 (neither in input nor during processing). This reduces the size of the representation of a feature structure massively.

UDiNe is a dedicated feature constraints solver that can be connected with different type systems. Unification is done destructively using the lazy copying technique introduced by Aït-Kaci, where only the affected structure must be copied. Non-destructive unification is performed using copy functions. UDiNe has been successfully used for several tasks in the DISCO project, viz. for parsing, generation, extended two-level morphology and surface oriented speech act processing.

The functionality of UDiNe is completed by several auxiliary functions. It is possible to remove inconsistent alternatives, to simplify structures, to extract subterms or to evaluate functional constraints. A general visiting function can be used for constructing user-oriented extensions. Furthermore, one can build the disjunctive normal form of a feature structure. This is needed by other tools used in the application system if they cannot handle distributed disjunction.

Intelligent Backtracking

In 1991 Uszkoreit introduced a new strategy for linguistic processing called *controlled linguistic deduction*. The evaluation of both conjunctive and disjunctive constraints can be controlled in this framework. For conjunctive constraints, the one with the highest failure probability should be evaluated first. For disjunctive ones, a success probability is used instead. The alternative with the highest success probability is used until a unification fails, in which case one has to backtrack to the next best alternative. Besides more complex ones, Uszkoreit also proposed a strategy that uses static values for the success probabilities (called preferences). In the following, we will call unifiers that control the evaluation of disjunctions in this way, unifiers *with intelligent backtracking*.

Because of similarities between this control method and the mechanism of intelligent backtracking in PROLOG, we can formulate the following properties that a unifier with intelligent backtracking should fulfill:

- INDEPENDENCE. Backtracking must be independent from the computation history, i.e. backtracking should not be restricted to the last processed disjunction.
- CONFLICT DETECTION. It must be possible to determine the disjunctive structures that are involved in a unification failure. This is necessary in order to restrict the set of candidates for backtracking.
- CONFLICT DEDUCTION. The conflict information of several unification errors can be used for further restricting the conflicting set of disjunctions. This avoids unnecessary backtracking.
- COMPLETENESS. It must be guaranteed that consistent combination of disjunction alternatives will be detected.

The most promising candidates for implementing intelligent backtracking are unifiers that use distributed disjunctions, since they provide most of the concepts mentioned above. Hereby, the notion of context common to all of these unifiers plays an important role. A *context* is partial function mapping disjunctions to corresponding alternatives. Every node has a unique context that describes under which disjunctions and which alternatives this node can be found. If a unification fails, the context of the node where the inconsistent information has been found is called *inconsistent context*. The inconsistent contexts are stored in order to deduce minimal inconsistent contexts and to detect a global inconsistency. Thus, inconsistent contexts can be used for conflict detection. The calculation of minimal inconsistent contexts corresponds to conflict deduction. The check for global inconsistency can be used for guaranteeing completeness.

We have implemented a prototypical extension of UDiNe that incorporates intelligent backtracking and provides the independence property mentioned above. The implementation works as follows. If a disjunction is encountered, the alternative with the highest preference is chosen, and only this alternative is used for later unifications. If a unification fails, the involved disjunctions are determined by the inconsistent context. Now one of the involved disjunctions has to be selected for backtracking. There are two possibilities: (i) one can use the static preference for this selection; and (ii) the unifier calls a user program in order to select a disjunction. The idea is to use the second selection mechanism for implementing more complex control methods. E.g. to achieve better selection criteria, we can provide the user program with the definition of the disjunction and the conjunctive part the disjunction has be unified with.

The backtracking of the selected disjunction first undoes the unification with the previously chosen alternative. We have modified the existing method for undoing destructive unification in order to guarantee a local undo. Secondly, the cancelled unifications are redone using the new selected alternative. The algorithm guarantees, that unification is restricted to the substructure starting with the disjunction.

The TDL language

TDL supports type definitions consisting of type constraints and feature constraints over the standard operators $\wedge$, $\vee$, $\neg$, and $\oplus$ (xor). The operators are generalized in that they can connect feature descriptions, coreference tags (logical variables) as well as types. TDL distinguishes between avm types (open-world semantics), sort types (closed-world semantics), and built-in types. In asking for the greatest lower bound of two avm types a and b which share no common subtype, TDL always returns $a \wedge b$ (open-world reasoning), and not $\bot$. The opposite case holds for sort types. Furthermore, sort types differ in another point from avm types in that they are not further structured, as are atoms. Moreover, TDL offers the possibility to declare *exhaustive and disjoint partitions* of types, for example $sign = word \oplus phrase$ which expresses the fact that (i) there are no other subtypes of *sign* than *word* and *phrase*, (ii) the sets of objects denoted by these types are disjoint, and (iii) the disjunction of *word* and *phrase* can be rewritten (during processing) to *sign*. In addition, one can declare sets of types as *incompatible*, meaning that the conjunction of them yields $\bot$.

TDL allows a grammarian to define and use parameterized templates (macros). There exists a special instance definition facility to ease the writing of lexicon entries which differ from normal types in that they are not entered into the type hierarchy. Strictly speaking, lexicon entries can be seen as the leaves in the type hierarchy which do not admit further subtypes. This dichotomy is the analogue to the distinction between classes and instances in object-oriented programming languages. Input given to TDL is parsed by a Zebu-generated LALR(1) parser to allow for an intuitive, high-level input syntax and to abstract from uninteresting details imposed by the unifier and the underlying Lisp system.

Type Hierarchy

The implementation of the type hierarchy is based on Aït-Kaci's bit vector encoding technique for boolean lattices (a bit-and/or operation corresponds to a LUB/GLB computation). The method has been modified to open-world reasoning over avm types in that potential GLB/LUB candidates must be verified by inspecting the type hierarchy through a sophisticated graph search. GLB, LUB and $\preceq$ computations have the nice property that they can be carried out in $O(n)$, where n is the number of types. Depending on the encoding method, the hierarchy occupies $O(n \log n)$ (compact encoding) resp. $O(n^2)$ (transitive closure encoding) bits.

The encoding algorithm is extended to cope with the redefinition of types, an essential part of an incremental grammar/lexicon development system. Redefining a type means not only to make changes local to this type. Instead, one has to redefine all dependents of this type—all subtypes, in case of a conjunctive type definition and all disjunction elements for a disjunctive type specification plus, in both cases, all types which mention these types in their definition. The dependent types of a type t can be characterized graph-theoretically via the *strongly connected components* of t.

Conjunctive, e.g., $x := y \wedge z$ and disjunctive type specifications, e.g., $x' := y' \vee z'$ are entered differently into the hierarchy: x inherits from its supertypes y and z, whereas x' defines itself through its elements y' and z'. This distinction is represented through the use of different kinds of edges in the type graph (bold edges denote disjunctive elements, see Fig. 2.4).

TDL decomposes complex definitions consisting of $\wedge$, $\vee$, and $\neg$ by introducing *intermediate types*, so that the resulting expression is either a pure conjunction or a disjunction. The same technique is applied when using $\oplus$ (see Fig. 2.4). $\oplus$ will be decomposed into $\wedge$, $\vee$ and $\neg$, plus additional intermediates. For each negated type $\neg t$, TDL introduces a new intermediate type symbol $|\neg t|$ with the definition $\neg t$ and declares it incompatible with t.

Incompatible types lead to the introduction of specialized bottom symbols (see Fig. 2.4) which

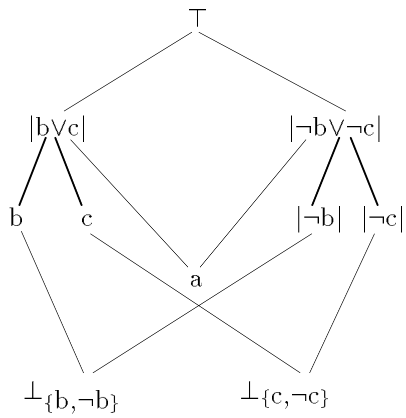


Figure 2.4: Decomposing $a := b \oplus c$, so that a inherits from the intermediates $|b \vee c|$ and $|\neg b \vee \neg c|$.

are, however, identified in the underlying logic. These bottom symbols must be propagated downwards by a mechanism called *bottom propagation* which takes place at definition time.

Symbolic Simplifier

The simplifier operates on arbitrary TDL expressions. Simplification is done at definition time as well as at run time when typed unification takes place (cf. Figure 2.3). The main issue of symbolic simplification is to avoid (i) unnecessary feature constraint unification and (ii) queries to the type hierarchy by simply applying 'syntactic' reduction rules.

The simplification schemata are well known from propositional calculus, e.g., De Morgan's laws, idempotence, identity, absorption, etc. They are hard-wired in Common Lisp in order to speed up computation. Formally, type simplification in TDL can be characterized as a term rewriting system. Confluency and termination is guaranteed by imposing a *generalized lexicographically ordered normal form* on terms (either CNF or DNF). In addition, this order has the nice effects of neglecting the law of commutativity (which is expensive and might lead to termination problems): there is only one representative for a given formula. Therefore, *memoization* is cheap and is employed in TDL to reuse precomputed results of simplified (sub)expressions (one must not cover all permutations of a formula). Additional reduction rules are applied at run time using 'semantic' information of the type hierarchy (GLB, LUB, and $\preceq$).

2.4.6 Overview

A comparison of the formalisms under consideration reveals that, in fact, they are not all equal as far as their expressive power and the number and kind of the formal devices used are concerned. Among the systems, the prototypical version of ALEP (ALEP-0) seems to have been the simplest one at the beginning of EAGLES, offering only a rudimentary type system without multiple inheritance, DNF-like treatment of disjunction (except for atoms), no subsumption check and no functional uncertainty. As for negation, only a restricted treatment (atoms) is possible.

A relative strength of ALEP-0 is the morphology, generation and transfer modules, which were not present in some of the other systems at the beginning of the project. However, the performance of the ALEP-0 generator did not meet the expectations of most users, a weakness that was not corrected until the release of ALEP-1.

The compilation of feature terms into prolog terms also enhances the runtime performance of ALEP.

Compared with ALEP, the 1993 version of ALE offers a wider spectrum of grammar writing tools, including multiple inheritance within the type system (a restricted version), negation of coreferences, extensionality declarations, ISA/ISNOTA operators and others. The handling of morphology by means of unary lexical rules and a DNF compilation of the fullform lexicon is a less satisfactory solution than those offered by ALEP and TDL. Finally, the system available in 1993 was uni-directional due to the lack of a generator.

While ALEP-0 and ALE are systems with fairly similar advantages and disadvantages, TDL and CUF offer the user much more grammar development facilities. In TDL, the type system with GLB/LUB semantics allows for multiple inheritance; full disjunctions can be handled by the system, along with certain types of negation (atoms, coreferences and full feature structures with some restrictions). Furthermore, distributed disjunctions are also supported. A prototype version of weighted disjunctions ("intelligent backtracking") was being tested at the beginning of the project.

As for the system architecture, TDL works with a modular chart parser, semantic-head-driven generator and the extended two level morphology X2 MorF, which is a further advantage over ALE and, in fact, ALEP-0.

The last of the systems in question, CUF, is a universal deduction system in which parsing, generation and other NLP algorithms are not predefined in the way they are in the systems described above. Instead, parsing, generation, morphological analysis, etc. are reduced to solving user-defined constraints on the feature structures describing the lexicon. Of course, some control information may also be needed here. The system supports full negation (and hence implication). The handling of disjunction is more modest than in TDL as only disjunctions between primitive types are handled by the constraint solver. On the other hand, the goal delaying technique allows for compiling out disjunctions into delayed goals, which however makes a weaker form of disjunction. Functional uncertainty is also encodable.

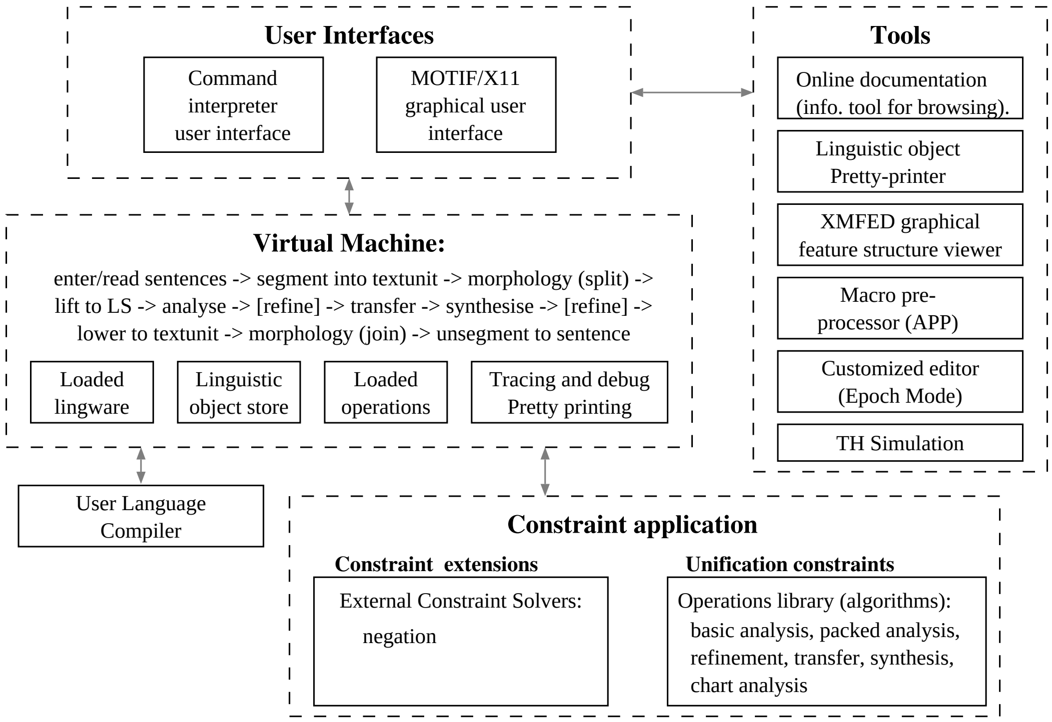


Figure 2.5: Outline of ALEP-0 environment modules

2.5 Development Platforms

2.5.1 Introduction

At the beginning of EAGLES, development platforms were normally only available as a "package deal" with a grammar formalism. The one notable exception from this is the Pleuk system which is described below.

There were huge differences concerning the support provided by a development platform. Some systems had only a very simple interface, which made use of a text-only terminal, while others had very user-friendly graphical interfaces. One of the earliest development platforms with a graphical user interface was the XEROX LFG workbench, which is still one of the most comfortable development platforms. Other formalisms which came with nice development platforms were ALEP and TDL. The development platforms of the LFG workbench, ALEP and TDL are described along with the formalism in sections 2.4.4, 2.4.2, and 2.4.5.

2.5.2 ALEP

The ALEP-0 prototype software is intended for small to medium scale lingware development, debugging and testing. It provides a formalism and tools outlined in Figure 2.5.

The most important application is the **Virtual Machine** (VM) with either graphical or command-line oriented user interface. The VM is implemented using Quintus Prolog Version 3.1.1.

Xalep ([Groenendijk and International, 1993a]) is an experimental, simple to use, graphical user interface to the VM. It consists of a number of 'toolboxes' (analysis, transfer, synthesis, text-, object- and lingware-handling). This is implemented in C using the OSF/Motif widget set and Quintus Prolog Foreign Language Interface. **Xalep** can run on most displays with X-Windows version 11 and makes direct calls to the Virtual Machine.

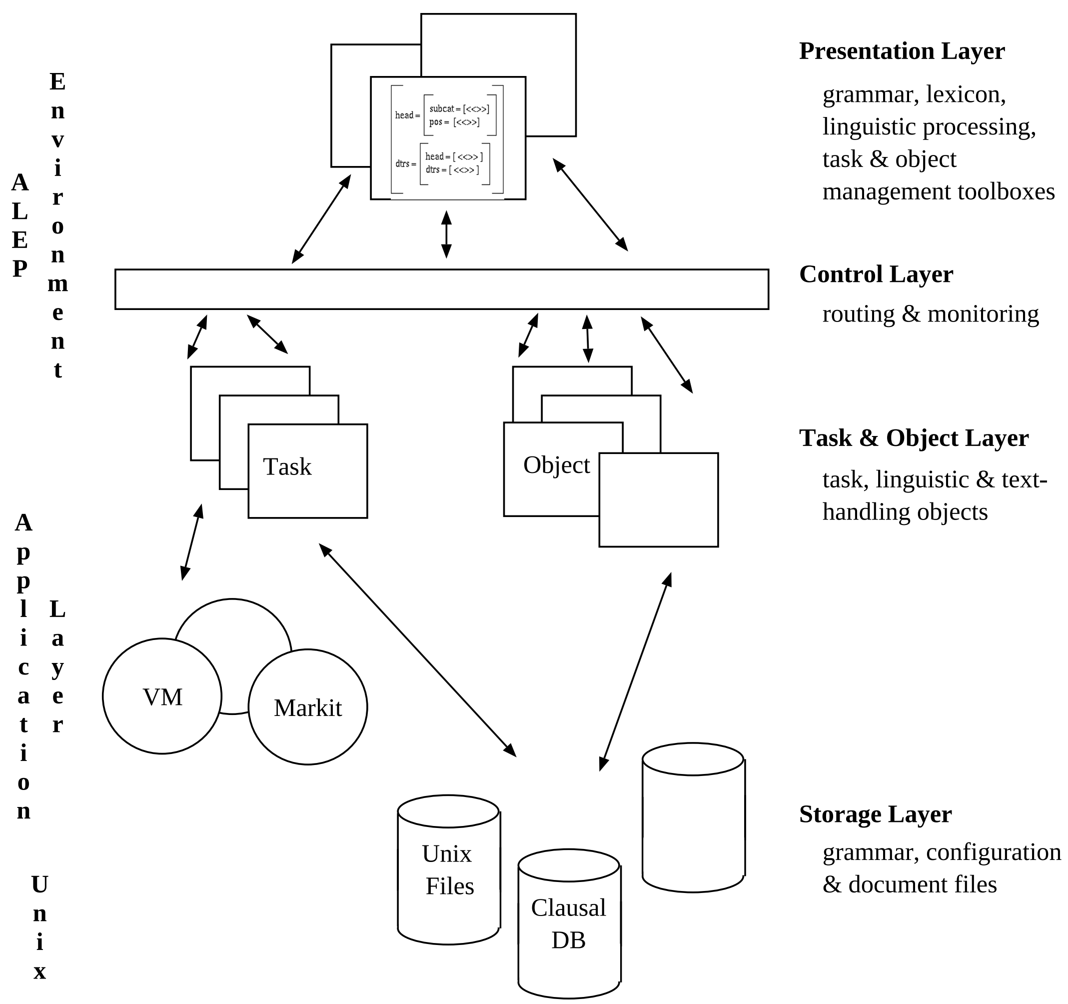


Figure 2.6: Outline of full ALEP Environment

ALEP-0 is not intended as open software as designed in ET-6/2 [IAI *et al.*, 1991]. The software applications are monolithical and only make use of other applications, such as the graphical feature viewer, via simple Unix calls and ICCCM based communication. Integration of other applications requires source code changes, although the VM allows for user contributed algorithms as separate modules and extensions via the *Hooks* and *External Operations Library* mechanisms [Simpkins *et al.*, 1993].

ALEP-0 is distributed with the following additional tools:

- *XmInfo*: a graphical tool for browsing hierarchically structured on line documentation. All ALEP-0 documentation can be used as an on line reference manual for ALEP-0 users.
- *Xmfed*: a graphical feature viewer for viewing large (linguistic) structures, integrated with Xalep and also with the VM tracer to provide graphical feedback during tracing of linguistic operations [Groenendijk and International, 1993b].
- *App*: a customized linguistic macro preprocessor.
- *alepemacs*: a grammar editing mode (elisp) for the GNU emacs editor with dynamic syntax checking and string completion.

ALEP-0 is only the small-scale prototype of the full ALEP platform [Meylemans and Simpkins, 1993]. The first version of the full environment has been designed and implemented by BIM [BIM, 1992], [BIM, 1993]. The environment of ALEP-1 (Figure 2.6) is to a certain extent formalism independent and open to customization and extension.

2.5.3 The LFG Grammar-writer's Workbench

The LFG Grammar-writer's Workbench is a computational environment that assists in writing and debugging Lexical Functional Grammars [Kaplan and Bresnan, 1982]. It provides linguists with a facility for writing syntactic, lexical, and morphological rules, and testing and editing them. For sentences or other strings parsed it provides the following analytic information:

Constituent-structures – whether or not they have valid f-structures.

The chart – containing all complete or incomplete bracketings of the input string that the grammar allows.

Functional-structures – including display of inconsistencies, incompletenesses, and incoherencies.

Functional-descriptions – the instantiated equations corresponding to particular c-structure nodes or particular f-structures

The system was originally written in the early 1980's, but it has evolved since then and also implements most of the later additions to the LFG formalism. It includes functional uncertainty [Kaplan and Zaenen, 1988], functional precedence [Kaplan and Zaenen, 1989], generalization over sets [Kaplan and Maxwell, 1988], and a rich notation for expressing c-structure patterns (including immediate dominance, linear precedence, and other regular predicates). The system allows correspondences between multiple levels of linguistic representation to be defined, as described by Kaplan [Kaplan, 1987] and Halvorsen and Kaplan [Halvorsen and Kaplan, 1988]. As a simple application of this capability, the system can display the properties of a semantic representation that is characterized and associated with a string by lexical and syntactic schemata.

The system provides a powerful interface for defining and manipulating linguistic rules and representations. After installing a collection of syntactic and lexical rules into the Grammar-writer's Workbench (henceforth GWB), you can see whether those rules are sufficient to analyze sentences or phrases in the language in question. You can also easily mix and match different sets of linguistic specifications as you experiment with different versions of particular rules and lexical entries, whether you have written them or they have been provided by other users of the system.

GWB also supports other activities that surround the business of grammar development. It can keep track of a number of editing tasks, essentially making available to the user an entire desktop's worth of memos, files, diagrams, versions of the paper being written about the grammar being developed, as well as the files of an entire working group. System output such as c-structures and f-structures can be copied into papers, and sentences from a paper can be used as input to the parser.

The Grammar Writer's Workbench is implemented in the Medley Lisp programming environment. This descendant of the original Xerox Interlisp-D system is now available from Venue Corporation. Medley runs on the original Xerox AI workstations and on a wide variety of Unix workstations (Sun, DEC, IBM, MIPS, HP, etc.) and on certain PC compatible platforms running MS-DOS. GWB and the documentation try to be self-contained, so that a linguist can successfully use the system without detailed knowledge of the programming environment. However, it may be helpful to obtain some familiarity with Medley's window, menu, mouse, and editing conventions from the more general Medley documentation. Medley supports a 16-bit character encoding so that national characters and special symbols are easy to work with and have their natural appearances.

2.5.4 Pleuk

Pleuk is a grammar development shell within which many different grammatical formalisms can be embedded. It offers publication-quality on-screen and hard copy presentation of linguistic information, via abstract representation and generic routines for processing. Sophisticated tools are provided for the manipulation of derivations.

By 1993, the following formalisms had been ported to Pleuk.

Cfg A simple context-free grammar system, intended for demonstration purposes.

HPSG-PL A system for developing HPSG-style grammars, produced at Simon Fraser University, Canada, by Fred Popowich, Sandi Kodric and Carl Vogel.

Mike A simple graph-based unification system, enhanced with additional operations for the treatment of free word order proposed by Mike Reape in various publications.

SLE A graph-based formalism enhanced with arbitrary relations in the manner of Johnson and Rosner (EACL, 1989) and Doerre and Eisele. Delayed evaluation is used to compute infinite relations. This system has been used for the development of several HPSG-style grammars.

Term A term-based unification grammar system, originally developed for the support of Unification Categorial Grammar (Zeevat, Klein and Calder).

2.5.5 TDL

The DISCO DEVELOPMENT SHELL serves as the basic architectural platform for the integration of natural language components in the DISCO core system, as well as for the COSMA application system [Neumann, 1993]. Following an *object oriented* architectural model a *two-step approach* was taken, where in the first step the architecture is developed independently of specific components to be used and of a particular flow of control. In the second phase the resulting 'frame system' is instantiated by the integration of existing components and by defining the particular flow of control between these components. Using an object-oriented design together with multiple inheritance has been shown fruitful for the system's modifiability, extensibility and incremental usability.

Several editing and visualization tools greatly facilitate the work of the grammar developer. The most prominent of them, FEGRAMED, provides the user with a fully interactive feature editor and viewer. There are many possibilities to customize the view onto a feature structure, such as hiding certain features or parts of a structure, specifying the feature order and many more. The large feature structures emerging in the process of constraint based formalisms make such a tool absolutely indispensable for grammar debugging. Main goals of the development of FEGRAMED were high portability and interfacing to different systems. Written in ANSI-C, it exists in Macintosh and OSF/Motif versions and is already used at several external sites.

There exists a graphical chart display with mouse-sensitive chart nodes and edges directly linked to the feature viewer, thus making debugging much simpler. It also provides a view of the running parser and enables you to inspect the effects of the chosen parsing strategy visually. A browser for the TDL type system permits navigation through a type lattice and is coupled with the feature editor. There are other tools as well, e.g., a TDL2LaTeX utility, an Emacs TDL mode, global switches which affect the behaviour of the whole system etc.

The diagnostics tool (DiTo) [Nerbonne *et al.*, 1993] containing close to 1500 annotated diagnostic sentences of German facilitates consistency maintenance and measuring of competence. The tool has been ported to several sites that participate in extending the test-sentence database.

2.6 Implemented Grammars

2.6.1 Introduction

The development of implemented large-scale grammatical resources is a rather time- and cost-intensive process: because of the inherent complexity of implemented grammars and the (still today) unsolved engineering problem of modularity and interface specification, very often a single grammarian builds up a monolithic resource over years. Accordingly, as of the beginning of EAGLES the majority of available implemented grammars represent resources that are rather specific to the project or research effort that they grew out of; grammar development for these grammars typically had started well before 1990 when widely accepted linguistic formalisms and especially development platforms were not (yet) available.

Two notable exceptions, however, were two HPSG-style grammars for German with (then) medium to substantial coverage, viz. (i) the grammatical component of the LILOG dialogue system developed by the research division of IBM Germany (see [Kiss, 1991a]) and (ii) the DISCO grammar developed at DFKI Saarbrücken (see [Netter, 1993]) which has since been developed into the DFKI German HPSG grammar (see section 3.6.4).

2.6.2 The RELATOR Survey

Shortly after the beginning of EAGLES the EU-funded project RELATOR produced a survey of available language resources, including implemented grammars. The following is a quote from [Bech *et al.*, 1993a] (The RELATOR project report available as a postscript document from `http://de.relator.research.ec.org/l=en/project.mlhtml`)

> For each of the nine EU languages there are grammars which were developed under the auspices of the EUROTRA machine translation project. The EUROTRA grammars for each language are more or less comprehensive (with respect to the text-domain treated). The domain treated in the project consisted of descriptive texts about satellite telecommunications, and the linguistic specifications cover the major constructions for the EU languages for such texts. These grammars are for both generation and analysis.
>
> For the EU languages, then, the status with regard to coverage of grammars is that for all languages at least the EUROTRA grammar exists. But in terms of usefulness, the EUROTRA grammars are probably not totally adequate, since the formalism has been abandoned by the EU, and the grammars will have to be rewritten. [...]
>
> For perhaps the majority of EU languages the EUROTRA grammar represents the largest and most comprehensive description of that language. For those languages which have a longer tradition of theoretical and computational linguistics (including the other Nordic countries surveyed), there is also a large variety of grammatical descriptions and implementations of different syntactic theories, including LFG, HPSG, Constraint Grammar, GB, Categorial Grammar and context free grammars. Nevertheless, in terms of syntactic coverage, the EUROTRA grammars still tend to be the most comprehensive ones available.
>
> The most important grammars for Danish are the EUROTRA grammar and its successor the PaTrans grammar which is part of the MT system PaTrans (developed by CST, Copenhagen). The ValP parser and associated grammar are mentioned in the report.
>
> For Dutch, the METAL grammars are quite comprehensive. These grammars are developed by Siemens Nixdorf, Liege, Belgium. In Leiden, Dutch grammars were developed for grammar checkers.
>
> Helsinki University, Finland, developed Constraint grammar for English and is also working on Finnish and other languages. The SITRA Corporation developed a dependency grammar for Finnish.
>
> For French, many grammars exist. The GRAAL grammar was developed by GSI-ERLI, and the ARIANE grammar by GETA, Grenoble.

For English, there are of course an abundance of grammars, in various formalisms. Here we could mention the Alvey tools and the statistical grammars developed in Lancaster. UMIST developed a large grammar formalism. In the commercial area, METAL (Sietec, Munich, Germany) and Sharp have developed grammars for English in connection with their MTdevelopment activities.

For Italian, the EUROTRA-grammar was developed into DIMA grammar for a grammar checker (DIMA, Torino).

The range of existing tools is also wide, including various taggers, parsers, MT systems, spelling checkers, grammar development environments, and user interfaces. These are both commercial and research oriented products with different levels of availability [...].

2.6.3 Further Pointers

Further pointers on implemented grammatical resources that were already available at the beginning of EAGLES can be found in [Bech *et al.*, 1993b]
(The RELATOR project report available as a postscript document from `http://de.relator.research.ec.org//l=en/project.mlhtml`).

Chapter 3

New Developments since the Start of EAGLES

3.1 Introduction

This chapter summarises the major trends in the area of grammar formalisms that have emerged within the duration of the EAGLES project.

As for the previous chapter, each section or subsection was written by an expert who is active in the respective field, and provides an insider's view of the relevant developments.

We have made no attempt to be complete in the coverage of developments. Different experts have given different weight to various issues, and highlighted the developments which they consider to be the most significant. Since objectivity is not possible with respect to these very new developments, we feel it preferable to present a variety of viewpoints.

A more unified view of the most important developments that cut across different grammatical theories, and different formalisms is given in chapter 4, which points out the major convergences that have become apparent in the course of the project.

3.2 Grammar Formalisms/Models

3.2.1 Introduction

This section provides an overview of the recent developments in various grammatical theories since the beginning of the project, i.e. 1993.

Some radical changes have been made in these theories since then. The most striking example is, of course, Government and Binding theory, which has shifted to an entirely new framework, the Minimalist Program. However, this is also the case in HPSG where the publication of the 1994 book, [Pollard and Sag, 1994b], marks a new orientation in the line of research. In Categorial Grammar the multi-dimensional approach turns out to be a fruitful track, as well as the subsumption-based approach to features. In TAG, XTAGs and LTAGs have been introduced, and in LFG there is a new line of research into the use of linear logic.

The various authors describe these developments from their own perspective, thus providing us with an insider's view of the ongoing research.

3.2.2 Categorial Grammar

Rule-based versus logic-based research

Recent developments in Steedman *et al*'s line of work are mostly either in the stipulation of new rules to account for linguistic phenomena, or in the exploration of the relation between syntactic and semantic dependencies. Examples of the former are Jacobson (syntax/semantics interface [Jacobson, 1994], [Jacobson,], [Jacobson, to appear]), Hoffmann (free word order in Turkish; Multiset CG, [Hoffmann, 1995a], [Hoffmann, 1995b]). An example of the latter is Steedman, who uses CCG to describe intonation and incremental NLP [Prevost and Steedman, to appear]. Both Cremers ([Cremers, 1993b], [Cremers, 1993a]) and Milward ([Milward, 1994b], [Milward, 1994a]) have worked on coordination, and come to the conclusion that it should be considered an independent mechanism.

Milward [Milward, 1995] implemented an incremental interpretation algorithm for CG. Cremers and Hijzelendoorn's parsing system Delilah exploits occurrence properties of bracket-free categorial grammar for the parsing of discontinuous structures in Dutch.

Recent developments in Van Benthem *et al*'s line of work occur mostly in the more adequate formal modelling of linguistic phenomena, either through the introduction of new type constructors, or through the refinement of the different dimensions of the sign (or both). Examples of the former are Carpenter's work on discontinuity, quantification and polymorphic coordination ([Carpenter, 1994], [Carpenter, 1995]), P. Hendriks's work on discontinuity phenomena [Hendriks, 1995b] and Kraak's work on French clitics [Kraak, 1995].

Before we summarize current work in the the latter line of research, let us say a bit more about it. Gabbay's work on Labeled Deductive Systems [Gabbay, 1991] has given rise to exploration of the idea of representing a sign in multiple dimensions. The nucleus of a sign remains its type (hence the term 'type-driven'), but it can be *labeled* with other kinds of information: prosodic, semantic, orthographic, hierarchic, etc. Each of those can have their own kind of operations and their own logic, but the purpose of parsing remains the same: relating a representation of an utterance (*signe*) to a semantic representation (*signifié*).

Examples of this line of work can be found in three flavours: logical, linguistic and computational. Logical research, investigating the fundamentals of sign-based Lambek Categorial Grammar, is carried out by Moortgat, Morrill, Oehrle, Kurtonina and Versmissen ([Moortgat and Oehrle, 1994], [Morrill, 1994a], [Moortgat, to appeara], [Moortgat, to appearc], [Oehrle and Moortgat,], [Oehrle, 1995c], [Moortgat and Kurtonina, to appear], [Versmissen, 1993a]). Emms ([Emms and Leiß, 1993], [Emms, 1993c], [Emms, 1993a], [Emms, 1994b], [Emms, 1994a], [Emms, 1995]) has extensively studied polymorphism in Categorial Grammar.

Linguistic applications are given by Oehrle, in his work on quantification, binding and the interpretation of prosody ([Oehrle, 1995d], [Oehrle, 1995a], [Oehrle, 1995a], [Oehrle, 1995b]) and

Morrill gives an account for tuples, discontinuity and gapping [Morrill, 1995b]. Moortgat presents a modal approach to in situ binding, [Moortgat, to appearb] and together with Oehrle he discusses adjacency, dependency and order [Moortgat and Oehrle, 1993b], and linguistic variation [Moortgat and Oehrle, 1993a], and Versmissen [Versmissen, 1993b] presents a categorial account of Mike Reape's theory of word order domains. H. Hendriks ([Dekker and Hendriks, 1994], [Hendriks, 1993]) studies the syntax/semantics interface, as well as *information packaging* across languages ([Hendriks, 1994], [Hendriks, 1995a]). Heylen works on sorts and features, trying to complement the multiplicative type system with a hierarchical lexicon theory.

Morrill presents applications in logic programming ([Lloré and Morrill, 1995], [Morrill, 1995a], [Morrill, 1995c], [Morrill and Aris, 1993]), Aarts ([Aarts, 1994], [Aarts, 1995], [Aarts and Trautwein, 1996]) discusses parsing algorithms for Lambek Categorial Grammar, and Emms [Emms, 1993b] also applies his ideas on polymorphism to parsing.

Carpenter implemented his ideas in what he calls a *Type-Logical Grammar Interactive Natural Deduction Theorem Prover*, which parses sentences and presents parses in various notations. Morrill's CATLOG, *a type-logical compiler and language processor*, covers most recent developments in Lambek CG, including intensionality, discontinuity and plurals.

Hybrid architectures: type-logical verus feature-logical

A second important division from a more recent date can be made on how to combine Categorial Grammar with features. Several ways in which one can achieve such a marriage have been studied. HPSG represents one extreme here by adopting only the notion of argument cancellation. Akin to the HPSG approach are Zeevat et.al.'s UCG and Uszkoreit's CUG, who consider category as yet another grammatical feature, which decides whether strings can combine into meaningful larger strings. Bouma [Bouma, 1993] studied the use of such a system, combined with the use of macros, for phenomena like agreement and control. Bouma and Van Noord ([Bouma and van Noord, 1994a], [Bouma and van Noord, 1994b]) have worked on lexical rules in relation to polymorphism.

So-called sign-based Lambek Categorial Grammar (Moortgat, Morrill) does not differ significantly in this respect. Application remains a type-driven operation, but the categories now carry features, and an argument is required to be *unifiable* with the functor, instead of *identical*. Instead of like this:

$$\mathrm{C} \rightarrow \mathrm{C}$$

the axiom now looks like this:

$$\mathrm{C}(\mathrm{x}_1,\ldots,\mathrm{x}_n) \rightarrow \mathrm{C}(\mathrm{x}_{n+1},\ldots,\mathrm{x}_m)$$

with x_1 unifiable with x_{n+1}, and so forth.

Van Noord and Bouma's ideas have also been implemented in Van Noord's Hdrug package (see `http://www.let.rug.nl/~vannoord/hdrug/`). König's LexGram system is a hybrid between HPSG and Lambek CG, and is built in in Dörre et al's CUF system, and includes a German grammar which covers movement, relative clauses and complement clauses. It also incorporates Reyle's Underspecified Discourse Representation Structures (UDRS's).

Quite a different approach is presented in recent work by Bayer & Johnson ([Bayer and Johnson, 1995], [Bayer, 1993], [Bayer, 1995]) Dörre, Gabbay and König ([Dörre *et al.*, 1995] and Dörre & Manandhar [Dörre and Manandhar, 1995]), who consider the relation between the requirements of the functor and the properties of the argument as that of *subsumption*, not unification. This approach does away with the destructive notion of instantiation, but only performs a type-check. The change has linguistic motivations—it makes more accurate predictions in crucial cases of agreement and coordination— but fits very nicely in a Lambek categorial framework, which is essentially based on derivability, not unifiability.

3.2.3 Constraint Grammar and Finite-State Syntax

[0] The author wishes to thank Atro Voutilainen for comments and suggestions on this section.

Introduction

Constraint Grammar[1] and Finite-state syntax[2] are closely related. CG provides the general framework and basic concepts of grammatical description which are further developed in FSIG. A major characteristic of both approaches is that grammatical description is entirely based on words and tags related to the single words. CG and FSIG are mere descriptive approaches. In contrast to most other grammar theories no claims about explanation of linguistic phenomena are made. In FSIG increased attention is given to the formalism which is based on the calculus of finite-state automata. While finite-state grammars are currently under development, their CG-based predecessors have by now successfully shown that robust, efficient, and linguistically adequate syntactic analyses of large scale running text can be achieved by an entirely linguistically motivated rule-based approach. Thus they can easily compete with state of the art statistics-based approaches, and are even superior to such approaches due to their explicit linguistic modeling.

Constraint Grammar – A Language-independent Linguistic Framework for Parsing Unrestricted Text

CG is designed according to the following criteria: The framework for linguistic description is geared towards language independence. The major aim is efficient and robust parsing of unrestricted text. Rule-like constraints constitute the kernel of the grammar which is reductionistic in nature, i.e. parsing is viewed as ambiguity reduction.

In the following major characteristics of CG will be elaborated on:

- Language independence

  Language independence is achieved by the clear separation of grammar and formalism. CG-based disambiguation grammars already exist for some languages, such as English ([Karlsson *et al.*, 1995], [Voutilainen, 1993]), French ([Chanod and Tapanainen, 1995a], [Chanod and Tapanainen, 1995b]), Swedish, German, Swahili, Danish, Portuguese and Finnish[3].

- Robustness

  Morphological heuristics account for words left unanalyzed after morphological analysis. Furthermore, heuristic modules can be added to grammar-based disambiguation.

- Orientation towards parsing

  The grammar is entirely geared towards parsing. Lexicon construction, preprocessing, and morphological analysis is optimized wrt. structural disambiguation. A major insight of CG is that the linguistically motivated introduction of ambiguity facilitates reductionistic parsing, and that there is not necessarily a positive correlation between the amount of ambiguity in the input and the speed of analysis.

- Ambiguity reduction

  The notion of ambiguity is central to CG. The parsing problem is viewed as an ambiguity reduction problem. Structural disambiguation already starts during preprocessing. Secondly, the lexicon is constructed such that disambiguation is optimized. Disambiguation principles also apply during morphological analysis. The parser reduces or at least restructures ambiguities that have been left over from prior processing. In the best case only one analysis remains.

- Constraints

  The task of constraints is the reduction of morphological and syntactic ambiguities. Constraints are derived from descriptive grammar and the study of tagged and untagged text

[1] In the following CG. For more details on CG see [Karlsson *et al.*, 1995].

[2] In the following FSIG. For basics on FSIG see [Koskenniemi, 1990], [Koskenniemi *et al.*, 1992], [Voutilainen and Tapanainen, 1993], and [Voutilainen, forthcomming].

[3] Personal communication: Atro Voutilainen

corpora of the language in question. Constraints state bits and pieces of linguistic phenomena, and, thus, are rather large in number (approx. 1 300 in ENGCG). Constraints are required to be maximally independent from each other, and order insensitive. The constraints postulated for a language L constitute the grammar of L.

- Syntax

  CG syntax is flat, and functional. No deep structure or phrase structure is assigned. Syntactic analysis operates entirely on words and tags. Three classes of tags are distinguished: (i) part-of-speech tags, representing the syntactic category of a word; (ii) morphological tags, representing morphosyntactic features of a word; and (iii) so called syntactic labels, i.e. tags indicating the surface syntactic function of a word. Syntactic labels represent the basic dependency relation that hold between words within a clause, specifically, between heads and their modifiers; for instance, nouns and pre- or post nominal modifiers such as determiners and adjectives, prepositions and their complements, etc. Syntactic labels are either introduced in the lexicon or via morphosyntactic mapping. In this case morphological and word order information is mapped into syntactic labels.

- Modularity of the parsing grammar

  The parsing problem is split up into three disambiguation problems:

  morphological disambiguation,

  assignment of intrasentential clause boundaries, and

  assignment of surface syntactic labels.

- Important steps prior to parsing proper

  Preprocessing, morphological and lexical analysis are viewed as basic steps prior to parsing. Text normalization, and sentence boundary detection positively effect parsing. The lexicon is designed such that structural disambiguation is supported. Morphological disambiguation is recognized as essential prior to parsing.

Lexicon

The lexicon plays a major role in disambiguation. Several parts of the lexicon are distinguished: The root lexicon defines a set of lexica that can be accessed at an initial stage. The stem lexicon is a list of stems. Optionally, grammatical features are associated with the stems. Sublexica handle inflection, derivation, and compounding. Besides the master lexicon (core lexicon of a language) several text type specific lexica are defined.

The lexicon is designed such that morphological and syntactic disambiguation are optimized. Thus, the readings for the most frequent words of a language (approx. 5 000 - 10 000 lemmata) are carefully designed, marginal words or word forms are represented such that they do not create unreasonable ambiguities for the more frequent forms. Ambiguities not relevant for parsing are represented by underspecification. Parsing relevant ambiguities are stated explicitly.

The lexicon is based on tagged and untagged corpora, and machine readable lexica. Lexicon construction starts with irregular lexis, and proceeds with regular ones. In addition, a list of fixed syntagms (e.g. idioms), and most frequent compounds (approx. 5 600 in case of ENGTWOL[4]) is provided. The lexicon is redefined iteratively by applying it to new text. Lexicon update is supported by interactive tools.

[4]ENGTWOL is the two-level morphology analyzer coupled with ENGCG, the English Constraint Grammar component, developed at the University of Helsinki.

Morphological Disambiguation

In morphological analysis a word form and all its readings are packed together into a single structure, the cohort. A cohort consists of the word form surrounded by quoted angle brackets, and all possible morphological analyses of the form, i.e. the base form, indicated by double quotes, part-of speech, morphological, and syntactic tags (indicated by a leading @). Figure 3.1 shows the cohort for the English word *stress* as analyzed by ENGTWOL. The analysis was obtained online from `http://www.lingsoft.fi/cgi-pub/engtwol`.

```
"<stress>"
        "stress" <SVO> V PRES -SG3 VFIN @+FMAINV
        "stress" <SVO> V INF
        "stress" <SVO> V IMP VFIN @+FMAINV
        "stress" <SVO> V SUBJUNCTIVE VFIN @+FMAINV
        "stress" N NOM SG
```

Figure 3.1: Cohort for the English word form *stress*

The form *stress* is ambiguous and has five possible analyses. It is either a transitive verb, indicated by <SVO>, or a noun in nominative singular, indicated by N NOM SG. The verb readings are ambiguous wrt. finiteness and mood. The syntactic label @+FMAINV assigned to the finite readings indicates that the syntactic function is "finite main predictor". For the meaning of the other tags in the cohort see figure 3.6.

In order to reduce ambiguity, morphological analysis is guided by two methodological principles, the Compound Elimination Principle, and the Derivative Elimination Principle. These two principles only apply within cohorts (local disambiguation). They are heuristics rather than principles.

- Compound Elimination Principle

  If a cohort contains readings with a different number of compound boundaries, then only the reading with the smallest number of boundaries is kept (cf. [Karlsson *et al.*, 1995], p. 32).

  In languages such as Swedish or German, where compounds are contiguous, segmentation is problematic. See for instance the Swedish word *tonkontroll* which might be segmented either as *ton_kontroll* (tone control) or *ton_kon_troll* (tone cone goblin). Thus the compound elimination principle would predict the first reading which is the correct one.

- Derivative Elimination Principle

  If a cohort contains derived and non-derived readings, all derived readings are discarded. (cf. [Karlsson *et al.*, 1995], p. 32)

The compound elimination principle is specifically useful for languages such as Swedish or German, where the segmentation of compounds is problematic. See, for instance, the strings in the above example.

Modules of Constraint Grammar Parsing

Full scale CG parsing consists of the following five consecutive modules:

- preprocessing,
- lexicon updating,
- morphological analysis,
- local morphological disambiguation,

- CG parsing.

The preprocessor performs case conversion, recognizes fixed syntagms, and specifies sentence delimiters. The lexicon updating module spots unknown words, and inserts them in one or more domain specific lexica. A two-level style morphological analyzer provides every word form with its morphological readings, the cohort. A heuristic component assigns two-level style descriptions to the words not recognized by the morphology. The morphological disambiguator performs local disambiguation. The CG parser performs the following interacting operations:

- context-sensitive morphological disambiguation by applying disambiguation constraints,
- assignment of surface syntactic functions by means of syntactic rules (constraints).

Input to the Parser

The parser takes two input files:

- the constraint file, and
- the text file.

The text file consists of morphologically analyzed and partially disambiguated word form tokens (cohorts), designated inter and intra sentential boundaries, information on capitalization, on fixed expressions, and on compounds. Mark-up information is characterized as such.

The constraint file consists of 13 sections comprising so called set and function declarations, morphosyntactic mappings, and disambiguation constraints.

The Rule Formalism

Set and function declarations are declarations of the following:

- Sentence delimiters

  $ indicates sentence final punctuation such as $.,$!, $?;
  **CLB indicates intrasentential boundaries.

- Sets of grammatical properties required by the constraints (set-declarations)

  These are represented as lists of set names and set elements, with set name as the first list element, and the rest comprising primitives (base forms of a word, angle-bracketed features, parts-of-speech, morphological features) or lists of primitives. Elements within a sublist are conjunctively related; toplevel set elements are disjunctively related. Set names are used in the target, and context conditions of constraints.

- Syntactic functions

  Syntactic functions represent labels for syntactic heads and modifiers. Syntactic labels are prefixed by @. Labels of modifiers indicate the direction the head is to be found, e.g. @DN>, the syntactic label of a determiner indicates that a nominal head is to be found to the right; or @<P-FMAINV characterizing a non-finite verb complement of a preposition that has to be found to the left.

  A special class of syntactic functions are principal functions. They specify:

  - those syntactic labels that occur only once in a simplex non coordinated sentence, such as finite main predicators, and
  - barrier elements, i.e. features characterizing non transcendable boundaries.

Morphosyntactic mappings are mapping relations between morphological features and syntactic labels.

Each mapping relation is a list of the three sublists:

(i) target: list of grouped morphological information,

(ii) context conditions: information about the relevant context,

(iii) syntactic labels: list of syntactic tags.

Disambiguation constraints and **syntactic constraints**
Constraints consist of the following parts:

- domain,
- operator,
- target, and
- context conditions.

A domain can either be a word form, or a variable over any word form that possesses the features specified in the target. The target describes which readings the constraint is about. The target can be a declared set name. The operator defines the operation performed on the readings picked out by the target.

For disambiguation constraints three operators are defined:

=!! discards either all non-target readings, if the context conditions are satisfied, or discards the target reading, if the context conditions are not satisfied;
=! discards all non-target readings, if the context conditions are satisfied;
=0 discards the target reading, if the context conditions are satisfied.

Syntactic constraints are specifically introduced to reduce the number of syntactic labels. The operators for syntactic constraints are:

=s! discards all non target readings, if the syntactic conditions are satisfied, and
=s0 discards the target reading, if the syntactic conditions are satisfied.

Context conditions are triples specifying polarity, position, and set. Set is represented by a set name. Position is a word's distance to the right or left of a target word. A context condition is called unbounded, if position contains a Kleene star, e.g. *2 states that the information represented by set has to be found two or more words to the right of the target. Relative positions refer to positions right or left to the absolute position of an instantiated *-position. Context conditions can be abbreviated with templates. $$-templates represent sequences of context conditions, &&-templates represent alternative context conditions.

In addition to the linguistically motivated constraints heuristic constraints are defined. Especially in the case of unknown words heuristic constraints apply. For example, unclear cases are resolved according to clear readings, levels of predomination are set by the grammarian, probabilities of readings are derived from corpora, heuristics to deactivate constraints and discard readings are specified, constraints are ordered according to their frequency of application. The heuristic part of CG as well as FSIG still awaits investigation

ENGCG parsing results

The sample analysis presented in figure 3.2 is intended to give a flavour of the appearance of CG output. A CG analysis is a sequence of maximally disambiguated cohorts. The star preserves information on original capitalization. For a definition of the tags occurring see figure 3.6.

Finite-State Syntax

Finite-state syntax is, as already mentioned, based on the linguistic framework developed in CG. The FSIG formalism builds on the calculus of finite-state automata, and thus, in contrast to CG, provides a well founded formal machinery.

In the following differences in grammatical description to CG are listed:

```
"<*i>"
        "i" <*> <NonMod> PRON PERS NOM SG1 SUBJ @SUBJ
"<see>"
        "see" <as/SVOC/A> <SVO> <SV> <InfComp> V PRES -SG3 VFIN @+FMAINV
"<a>"
        "a" <Indef> DET CENTRAL ART SG @DN>
"<bird>"
        "bird" N NOM SG  @OBJ
"<$.>"
```

Figure 3.2: ENGCG analysis of *I see a bird*

- Explicit distinction of finite and non-finite clauses

  Non-finite verb chains are provided with lower case syntactic tags, finite verb chains and other information are represented with upper case tags.

- Distinction of syntactic structure and grammatical function

  In addition to tags for syntactic structure (syntactic labels in CG), tags indicating the grammatical function of a word within a matrix clause are introduced. Structural tags are characterized by a leading @, e.g. @MV. Functional tags end with @, e.g. SUBJ@. All words are annotated with structural tags. Main verbs may also have functional tags, in so far as they are main matrix verbs or main verbs of clausal complements, such as sentential subjects (SUBJ@) and objects (OBJ@), or main verbs of adverbial clauses (ADVL@), parenthetical clauses (PAREN@), etc.

- More accurate treatment of intrasentential structure

  Five kinds of word boundaries are introduced: sentence boundary (@@), the plain word boundary (@), beginning of a center embedding (@<), end of a center embedding (@>), and border between two finite juxtaposed clauses (@/).

  Information on deferred prepositions is explicitly represented with tags: @>>P indicates that a deferred preposition is be expected to the right. The deferred element itself is annotated with the tag <Deferred>, stating that there is no complement to the left.

- More consistently than in CG, syntactic information that can be inferred from the tag context is not explicitly represented

  Thus, for instance, tags such as @DN> and @NN>, as used in CG, are changed into @N> in FSIG, only indicating the type of head and the direction in which the head has to be found.

For further illustration of the differences in linguistic description between CG and FSIG see the analysis of the following sentence: *Henry dislikes her leaving so early.* The FSIG sample analysis is taken from [Voutilainen and Tapanainen, 1993]. Note that morphological tags are omitted. The ENGCG analysis is repeated as obtained from http://www.lingsoft.fi/cgi-pub/engcg.

The introduction of grammatical function tags and differentiation between elements of finite and non-finite clauses allows, amongst other things, for a more appropriate treatment of control constructions than is allowed for in a CG analysis. Considering the CG and FSIG analyses of *dislikes her leaving*, we get the following picture. In FSIG (figure 3.4) *her* is analysed as subject of a non-finite verb (@subj), *leaving* is analysed as a non-finite verb (@mv), and predicate of the matrix sentence's object clause (OBJ@), and *see* is analysed as the predicate of the matrix clause (MAINC@). In CG (figure 3.3), in contrast, *her* is analysed as a premodifying genitive (@GN>) awaiting a noun to its right, although in this type of construction there is none. *Leaving* is analysed as non-finite main verb (@-FMAINV) and object (@OBJ), which obscures the fact that according to its syntactic properties *leaving* is a non-finite main verb (@mv), and according to its grammatical function within the specific matrix clause it is the predicate of the clausal object (OBJ@). Those who wonder why the adverbs *so* and *early* in figure 3.4 are not annotated with lower case tags, note that in order to avoid structurally unresolvable ambiguity, no distinction wrt. finiteness is made in case of adverbs. Thus, no lower case tags @A> or @ADVL exist. For a definition of the tags occurring see figure 3.6 and 3.7 respectively.

```
"<*henry>"
        "henry" <*> <Proper> N NOM SG @SUBJ
"<dislikes>"
        "dislike" <SVO> V PRES SG3 VFIN @+FMAINV
"<her>"
        "she" PRON PERS FEM GEN SG3 @GN>
"<leaving>"
        "leave" <SVOO> <SVOC/N> <SVOC/A> <SVO> <SV> <P/for> PCP1 @OBJ @-FMAINV
"<so>"
        "so" ADV @ADVL @AD-A>
"<early>"
        "early" ADV ADVL @ADVL "early" A ABS @OBJ
"<$.>"
```

Figure 3.3: ENGCG analysis of *Henry dislikes her leaving so early*

```
                                             @@
Henry          N         @SUBJ               @
dislikes       V         @MV      MAINC@     @
her            PRON      @subj               @
leaving        PCP1      @mv      OBJ@       @
so             ADV       @A>                 @
early          ADV       @ADVL               @
.              FULLSTOP                      @@
```

Figure 3.4: FSIG analysis of *Henry dislikes her leaving so early*

Finite-State Rule Formalism

Grammar rules and sentence descriptions are represented as regular expressions. Before parsing the regular expressions are compiled into finite-state automata. In the following characteristics of the formalism will be listed:

- The formalism has the full power of regular expressions.
- All types of structural ambiguity are represented by a uniform rule component, based on regular expressions. Regular expressions are used in three types of rules:
  - constants,
  - predicates, and
  - implications.
- Due to the use of implication rules distributional generalizations can be stated more easily in FSIG than in CG. For an English core grammar this means theoretically approximately 200 rules are required in FS syntax vs. 1 300 constraints in CG.

- Each sentence reading is represented separately by the parser. In CG only word readings are distinguished. Sentence readings are hidden in the cohorts.

Constants are abbreviations of regular expressions. Particularly useful constants are ".", "..", and the functor \, where one dot refers to any sequence of tokens within a word, two dots refer to any sequence of tokens within a clause, and the backslash refers to sequences of tokens not containing the sequences of tokens specified as arguments of the backslash, e.g. \[@@|@/] specifies any sequence of tokens not containing a sentence boundary (@@) or an intrasentential boundary (@/). Constant declarations are of the form
NAME = REGULAR_EXPRESSION.

Predicates are predicate-argument structures, where the argument is a regular expression, and the predicate is a name. With the functor \, and a regular expression X we could define a predicate forbid(X) which excludes all sequences containing the sequence specified in X. Predicate declarations are of the form
PREDICATENAME(REGULAR_EXPRESSION) = f REGULAR_EXPRESSION,
where f denotes an operation on the regular expression.

Implications are of the form
EXPRESSION $\rightarrow$ LC$_1$_RC$_1$, $\cdots$, LC$_n$_RC$_n$,
where EXPRESSION, the left context (LC), and the right context (RC) are regular expressions.

Finite-State Parsing

Input to the finite-state parser are morphologically analyzed and enriched sentences represented as regular expressions. A regular expression of a sentence contains information similar to the information of a comparable sequence of cohorts as already known from CG. In a regular expression of a sentence words are separated by <wb>, indicating word boundaries. Disjunctive information is indicated by grouping ([ ]), and the disjunction operator |.

As an example see figure 3.5, representing a regular expression for the sentence *the red car is in the garden*. The example is taken from [Voutilainen, forthcomming]. In the analysis given, *the* is a determiner of a nominal to its right; *red* is either an adjectival prenominal modifier or a predicate complement; *car* and *garden* are nouns in nominative singular which might function as subject, object or complement of a preposition; *is* is a present tense main verb or auxiliary; and *in* is either an adverbial preposition or a preposition modifying a noun to its left, or an adverbial adverb. For a definition of the tags occurring see figure 3.7.

```
The     the      DET       @>N                         <wb>
red     red     [A ABS    [@>N | @SUBJCOMP]]           <wb>
car     car      N NOM SG [@SUBJ | @OBJ | @PREPCOMP] <wb>
is      be       V PRES   [@MAINVERB | @AUX]          <wb>
in      in     [[PREP     [@ADVL | @N<]] |
                [ADV       @ADVL]]                     <wb>
the     the      DET       @>N                         <wb>
garden  garden   N NOM SG [@SUBJ | @OBJ | @PREPCOMP] <wb>
.
```

Figure 3.5: regular expression for *the red car is in the garden*

Syntactic analysis is the intersection of an input sentence automaton with each rule automaton. This can be done in various ways. Several parsing methods have been investigated. Ambiguity reduction by first intersecting the sentence automaton with carefully selected rules, and then applying all the remaining rules in parallel is considered to be rather successful (cf. [Voutilainen and Tapanainen, 1993]). Similar rule automata are merged during compilation. Automata irrelevant for parsing a specific sentence are excluded before parsing.

The whole parsing scheme is sketched in the following:

- preprocessing (text normalization, sentence boundary detection),
- morphological analysis, enrichment with syntactic and clause boundary ambiguities,
- compilation of each sentence reading into a finite-state automaton,
- selection of relevant rules for the sentence,
- intersection of selected rule groups with the sentence automaton,
- parallel application of remaining rules,
- heuristic ranking of resulting analyses.

<Indef>	indefinite determiner, e.g. a
<InfComp>	potential infinitive complementizer
<NonMod>	pronoun with no determiner or modifier
<P/for>	the verb can take a for-PP as complement
<SVO>	monotransitive
<SV>	intransitive
<SVOC/A>	complex transitive verb with adjective complement
<SVOC/N>	complex transitive with noun complement
<SVOO>	ditransitive
<as/SVOC/A>	complex transitive verb with adjective, prepositional verb
-SG3	non third singular
@+FMAINV	finite main predicator
@-FMAINV	non-finite main predicator
@AD-A>	premodifier for a adjective
@ADVL	adverbial
@DN>	determiner
@GN	premodifying genitive
@OBJ	object
@SUBJ	subject
A	adjective
ABS	absolutive form of an adjective
ART	article
CENTRAL	central determiner, e.g. this
DET	determiner
IMP	imperative
N	noun
NOM	nominative
PCP1	present participle
PERS	personal pronoun
PRES	present
PRON	pronoun
SG	singular
SG1	first person singular
SUBJ	pronoun in nominative, always used as subject
SUBJUNCTIVE	subjunctive
V	verb
VFIN	finite verb

Figure 3.6: a sample of CG tags

<wb>	word boundary
@>N	determiner or modifier of a nominal in the right-hand context
A	adjective
@A>	premodifying adverb
@ADVL	adverbial
@MAINVERB	main verb in a finite clause
@MV	main verb
@N<	determiner or modifier of a nominal in the left-hand context
@OBJ	object in a finite clause
@PREPCOMP	complement of a preposition
@SUBJ	subject in a finite clause
@SUBJCOMP	predicate complement
ABS	absolutive
ADV	adverb
DET	determiner
MAINC@	finite matrix verb
N	noun
NOM	nominative
OBJ@	object clause
PREP	preposition
SG	singular
V	verb

Figure 3.7: regular expression for sample tags used in FSIG

3.2.4 Dependency Grammar

Word Grammar

While Hellwig concentrates on the formal aspects of implementing and processing dependency grammars, there have also been new DG-oriented analyses of linguistic phenomena, especially those difficult to handle within Phrase Structure and Categorial Grammar.

Word Grammar (cf. [Hudson, 1984]) rejects the notions constituent and phrase as intermediate structures of language, and postulates that the description of a language should be formulated in terms of relations holding between words. This claim may be viewed as a consequent continuation of the current tendency towards lexicalised grammars with a prominent notion of syntactic head, i.e. a word that bears all the important information that is associated with the constituent. In comparison with other theories of grammar, dependencies between phrases are in Hudson's theory replaced by dependencies between their heads.

Dependency Grammar

In the period since the begin of EAGLES most of the new linguistic research in Dependency Grammar has been devoted to finding convergencies between DG and other linguistic theories.

Covington ([Covington, 1994]) has worked on a reinterpretation of some notions of dependency grammar that would open this theory towards analyses made in the framework of phrase structure grammar, and, conversely, to incorporate a number of dependency-grammatical notions into PSG-based theories such as GB. Such a reinterpretation could also allow for a more direct comparison between both linguistic traditions.

Since dependency structures may be viewed as trees, the whole problem can be reduced to defining a function which would map dependency trees to the corresponding phrase structures and vice versa. The difficult point is that there is no direct correspondence between the different bar levels in X-bar theory and the nodes of a dependency tree. Covington thus hints that representations in dependency grammar usually consist not only of dependency trees, but also contain some more specific information about the sort of the dependencies described (modification, complementation, etc.). In GB, all these notions are expressed structurally, so that the translation mapping will create different trees depending on the additional information of the underlying dependency relations. Another factor to be taken into consideration is the surface ordering, which matters when modifiers are mapped to binary branching X-bar structures.

Accordingly, notions such as c-command and government or constituency can be expressed within dependency grammar in terms of the relation *being a dependent of*, cf. [Covington, 1994].

This development seems promising as it eliminates some essential hindrances to a free exchange of theoretical contributions between the theories. However, the research has only been made on a small number of issues, and to our knowledge there is no large-scale comparison of Dependency Grammar with any other theory or formalism that would concern a substantially greater subset of linguistic knowledge.

3.2.5 Government Binding Theory

By the beginning of the Eagles project the seeds of a paradigm shift had already been sown in the form of Chomsky's underground "least effort" paper (published in 1991 as "Some notes on economy of derivation and representation" [Chomsky, 1991]), and much related work by GB linguists. By the time his "Minimalist Program" paper had been circulated and published ([Chomsky, 1993]), there was already considerable research generated by this framework, and it began to seem unlikely that there would be a return to the "Barriers" framework. While the so-called Minimalist Program is still a "principles and parameters" type approach (i.e., the variety of constructions in natural language being generated by the interaction of universal principles with language specific parameters), and many of the analyses derived from work in the Barriers framework, the theory is quite different in character and detail. Among the major differences are the following:

1. The level of DS had already become both redundant on the one hand, and problematic on the other (non-thematic structure, such as the empty subject for the passivized NP, were hard to license other than by stipulation). Hence, the Minimalist Program proposed to eliminate both it and S-structure, retaining only the "real" interfaces with other cognitive systems, PF and LF.

2. Derivation of a sentence proceeds as follows (simplified): a fixed set of lexical entries is chosen (**N**), and a structure is built by means of two processes: *Merge* and *Move*. *Merge* combines two linguistic elements (either a word from **N** or a previously built structure). *Move* raises an element by attaching a copy of it to the structure it is contained in. At a particular point the built-up structure is mapped to PF (Spell-Out); the derivation continues (now invisible to PF) to the final LF structure.

3. Motivation for movement is solely checking of (formal) features. It is restrained by "economy" principles such as *Greed* (a category moves only to check its own features), *Procrastinate* (wait as long as possible in the derivation before moving), etc.

4. A set of convergent (i.e. successful) derivations is kept (the Reference Set) and the most economical derivation is chosen. This has the flavor of Optimality Theory in phonology, but the members of the Reference Set are dynamic (derivations) as opposed to phonological OT, where they are usually portrayed as static representations.

5. Hence, the entire set of conditions on Move-α in GB has been replaced in the Minimalist framework by the requirement that a derivation converge: using only legal operations and structures, and choosing the most economical derivation (e.g., shortest movements), all (strong) formal features must have been "checked" by Spec-Head agreement.

6. The relation of linear order to hierarchical form is a subject of some controversy: proposals range from Kayne's (1993) Antisymmetry Hypothesis, which maps it directly onto heirarchical structure (in essence eliminating trees from the theory: every structure is a list), to proposals divorcing them entirely, as does HPSG.

As a result of this, it was decided to eliminate further work on "GB"-datatypes in the Eagles project, as the theory had been effectively replaced by the Minimalist Program. Unfortunately, there has been little work on (computational) formalization of minimalist theory even though Chomsky's expositions of the theory are quite formal in character; as an emerging paradigm, it is still in a state of flux, and there is considerable controversy surrounding some of the details. Probably for the same reason there seems to be little implementational work in the *spirit* of the minimalist program. There are, however, implementations of certain aspects of the hypothesis, for example Wu's 1993 dissertation based on Kayne's Antisymmetry Hypothesis. There is, of course, a large outpouring of grammatical analysis in a wide variety of languages in the framework, as it has become the inheritor of post-GB linguistics. However, it should be pointed out that these, of course, are grammatical *analyses* of various constructions (just as under GB) intended to provide empirical evidence arguing for revision or sharpening of the theory, and not complete grammars for any particular language. (A grammar for a language X in the Minimalist framework, as in the previous Barriers framework, would of course be just the Lexicon for X, i.e., the lexical entries and parameters.)

3.2.6 Head-Driven Phrase Structure Grammar

The most important new development in the HPSG grammatical theory (and related formalism) since the beginning of the project is represented by the publication of [Pollard and Sag, 1994b] (henceforth PS94), a volume which has been circulating for a long time in a draft version and which has undergone a number of revisions since its first appearance. Here we will ascribe to it more merits than it really has, since we will accredit innovations in areas such as control and binding theory to this volume, while, actually, they are the result of a revision of work which appeared between 1987 and 1994. Thus some of the issues that we will touch on in this chapter were already known before the beginning of the project.

As in section 2.2.5, we will focuss not on the linguistically most interesting topics, but on those aspects of the linguistic description which introduce devices which are of some interest from a formal point of view.

Phrasal Organization

The view of the grammar as containing a set of descriptions which are distinct from types, i.e. phrases structure rules, clearly disappears in PS94. Firstly, all the various phrases structure schemata are now part of a general principle with many alternations:

> The ID Principle:
> Every headed phrase must satisfy exactly one of the ID schemata
> (p. 399)

Secondly, the whole load of capturing phrase structure information is shifted from phrase structure rules to phrasal types. Consider for instance the Schema 2 of PS94, ch.9:

> Schema 2: a phrase with DTR value of sort *head-comp-struc* in which the HEAD-DTR value is a lexical sign.
> (p. 348)

Here the notation [SUBCAT < [] >], or [SUBJ < [] > & COMPS <>], is replaced by explicit reference to the type *phrase* (which is a subtype of sign imposing the condition [COMPS <>]. Moreover, the condition that the HEAD-DTR value is a lexical sign could be conceived as a restriction over the type *head-comp-struc*, and as such it needs not to be stated explicitly in the schema. As a consequence, Schema 2 reduces to a subtype of the type *phrase*. For instance, if we define the type *phrase* as in (28) (*cons-str* is a supertype of *head-str* and *coord-str*), Schema 2 can be represented as an instance or a subtype of *phrase* with the restrictions which are shown in (29):

(28) $\left[\begin{array}{ll}\text{SYNSEM}\,|\,.....\,|\ \text{COMPS} & \langle\rangle \\ \text{DTRS} & \text{cons-str}\end{array}\right]_{phrase}$

(29) *schema-2* = $\left[\text{DTRS} \quad \text{head-comp-struc}\right]_{phrase}$

This reduction of phrase structure rules to types is a highly desirable theoretical result, even though, from a formal point of view, it introduces a further complication: only few systems, such as TFS, are able to implement such a parsing strategy *directly* as type resolution. Other frameworks, such as TDL, rely on mixed approaches where the daughters of a phrase structure rule appear as members of a special list valued attribute. Finally, the family of the so called "lean" formalisms, such as ALEP and ALE is definitely bound to the use of traditional phrase structure rules. It is obviously possible to *emulate* the type-based phrasal organization in a number of ways, but, especially in the last cases, this would introduce a great deal of redundancy in the grammatical organization.

Control and Raising: Lexical Types

Control and raising are handled in PS94 thorough principles which impose a set of well formedness constraints on the lexicon. Let us start from the Control Principle:

> Control Theory:
> If the CONTENT of an unsaturated phrase is the SOA-ARG in a psoa whose relation is a control relation, then the subject SUBCAT element of that phrase is
>
> - reflexive; and
> - coindexed with the INFLUENCED, COMMITTOR or EXPERIENCER value in that psoa, according as the control relation is of sort *influence*, *commitment*, or *orientation*, respectively
>
> (p. 302)

where a control relation is a type which has as subtypes the relation *influence*, *commitment* and *orientation*, which have in turn subtypes, depending on the different verbal relations by which they are realized. This principle is lexical, in the sense that it is supposed to capture certain generalizations over lexical entries: lexical entries should not contain information in order to state the proper coindexations, but this should follow as a consequence of the control theory. Thus Control Theory is a principle only as far as it makes use of the usual implicational format. Otherwise, it can be seen as a type constraining the information contained in lexical entries (in this case the type *control-theory* has to contain at least two other disjunctions (besides the one between *influence*, *commitment* and *orientation*); namely the one taking care of *non* control relations and the one taking care of control verbs when they do not subcategorize for an unsaturated complement (cf [Bell and llegas, to appear])).

Here it is worth noting that, as a lexical type, *control-theory* should be a subtype of *local*, i.e. the first attribute with visibility over both semantics and syntax. However, such an attribute should be subtyped also to account for other aspects of the grammatical organization. For instance, if a *linking theory* has to be implemented, either along the lines of [Davis, 1995] or [Dini and Di Tomaso, 1995], *local* should also be subtyped to account for connections between syntax and semantics. In a closed world system, such as the one which PS94 seem to assume, this causes the introduction of a huge amount of common minimal subtypes just for the purpose of satisfying the properties of unification under a closed world assumption. This consequence is obviously unsatisfactory, as the multiplication of linguistically unmotivated subtypes makes the management of grammars of a certain size a rather difficult matter. For this reason, we believe that a principled organization of the grammar could be only achieved by introducing devices such as multiple inheritance, as described in [Erbach, 1994b], which allow *partitions* of a given type without requiring an explicit introduction of *ad hoc* subtypes.

Analogous considerations hold for the Raising Principle, whose implementation is made even more difficult by the fact that it is conceived as a *filter* rather than as a positive statement:

> 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>]

Firstly, we note that in order to work properly, this principle requires both relational constraints, such as membership, and negated coreferences. Moreover, it is not clear at all how, under the standard formulation of the HPSG content attribute, the notion of *element with no semantic role assigned* could be captured. Indeed, assuming that relations are typed, and that every relation introduces its own set of attributes to designate the proper semantic roles, a formal implementation of the raising principle should disjunctively mention all the possible relations and, for every relation, all the possible roles it introduces. For this reason we are rather skeptical about the fact that a serious implementation of the raising principle could be provided at the current stage of development of unification based formalisms.

Semantics

The Semantic Principle undergoes a major revision in PS94. This is the final formulation

> Semantic Principle
> In a headed phrase:
>
> **a** the RETRIEVED value is a list whose set of elements forms a subset of the union of the QSTOREs of the daughters; and the QSTORE value is the relative complement of that set; and
>
> **b** (Case 1) if the semantic heads CONTENT value is of sort *psoa*, then the NUCLEUS value is identical with that of the semantics head, and the QUANTS value is the concatenation of the RETRIVED value and the semantic heads QUANTS value; (Case 2) otherwise the retrieved value is empty and the CONTENT value is token identical to that of the semantic head.
>
> (p. 323)

Let us start with condition **a**. We note that, crucially, scope ambiguities of quantifiers are obtained through a special function which changes sets into lists. During this passage, all the possible orders are exploded. The feature structure described in **a** would then be:

(30) $$\begin{bmatrix} \text{QSTORE} & \boxed{1} \\ \text{RETRIEVED} & \textbf{set-to-list}(\boxed{2}) \\ \text{DTRS} \mid \text{X-DTRS} & \boxed{3} \end{bmatrix}$$
Constraint $\bigcup\{X: \exists\, Y : Y \in \boxed{3} \wedge X = \text{QSTORE}(Y)\} = \cup\,(\boxed{1}, \boxed{2})$

The second part of the principle is simpler, and can straightforwardly be implemented through a disjunctive type or an implicational constraint (we ignore for simplicity the case of adjunct structures, which would introduce a further disjunction):

(31) *content-principle*= *predicative* $\vee$ *non-predicative*
$$\begin{bmatrix} \text{SYN} \mid \ldots \mid \text{CONT} & \begin{bmatrix} \text{NUCLEUS} & \boxed{1} \\ \text{QUANTS} & \boxed{2} \oplus \boxed{3} \end{bmatrix}_{psoa} \\ \text{RETRIEVED} & \boxed{2} \\ \text{HEAD-DTR} \mid \ldots \mid \text{QUANTS} & \boxed{3} \end{bmatrix}_{predicative}$$

$$ {}_{\textit{non-predicative}}\left[\begin{array}{ll} \text{SYN}\,|\ldots|\,\text{CONT} & \boxed{1}\ {}_{\neg\, psoa}[\,] \\ \text{RETRIEVED} & \langle\rangle \\ \text{HEAD-DTR}\,|\ldots|\,\text{CONT} & \boxed{1} \end{array}\right] $$

To sum up, the Semantics principle as it is stated makes use of:

- relational constraints
- user defined functions such as **set-to-list**
- universal quantification (which can be emulated through user defined functions)
- negation of types
- set operations
- list operations

There are obviously certain possible simplifications. For instance, a binary branching grammar would dispense with the use of universal quantification. However, given the way in which the principle is formulated, spurious ambiguities would increase, due to the fact that the possible levels of retrieval are multiplied. Moreover, one could challenge the idea that quantifier scoping is a task which pertains to the grammar 'strictu sensu': in fact, as it stands, inadequacies of this version of the semantic principle have been pointed out by [Carpenter, 1992b]. Perhaps, the most desirable solution is having the QUANTS list reflect the superficial linear order of the quantifiers (a *quasi logical form* a la [Alshawi, 1991]), thus delaying the resolution of scope ambiguities to other modules of the linguistic organization.

Binding Theory

The HPSG Binding Theory is standardly formulated as follows:

> **Principle A.** A Locally o-commanded anaphor must be locally o-bound.
>
> **Principle B** A personal pronoun must be locally o-free.
>
> **Principle C** A nonpronoun must be o-free.
>
> (p. 254)

Where 'a *synsem* object is an *anaphor* (respectively, a *personal pronoun*, *nonpronoun*) provided its LOCAL|CONTENT value is of sort *ana*(respectively *ppro*, *npro*)' (p. 401). O-binding, local o-command and o-command are in turn defined in the following way:

> O-binding
> Y (*locally*) *o-binds* Z just in case Y and Z are coindexed and Y (locally) o-commands Z. If Z is not (locally) o-bound, then it is said to be (*locally*) *o-free*.
> (p. 254)
>
> Local O-Command
> Let Y and Z be synsem objects with distinct LOCAL values, Y referential. Then Y *locally o-commands* Z just in case either:
>
> **i** Y is less oblique than Z; or
>
> **ii** Y locally o-commands some X that subcategorizes for Z.
>
> (p.278)
>
> O-Command: Let Y and Z be *synsem* objects, with distinct LOCAL values, Y referential. Then Y *o-commands* Z just in case either:

i Y is less oblique than Z; or

ii Y o-commands some X that subcategorizes for Z; or

iii Y o-commands some X that is a projection of Z (i.e. the HEAD values of X and Z are identical

(p. 279)

First of all, it must be said that the relations of local o-command and o-command are extremely unlikely to be implemented in a declarative way, given their recursive character. Thus, if adopted, they need to be 'plugged' directly in any implementation of HPSG Binding theory. Next question is: assuming we renounce the wish to make explicit use of the relation of o-command, is there any way to provide a declarative implementation of the binding theory? Let us start from Principle A, and let us assume that implicational constraints are allowed in the metalanguage, as well as relational constraints. Then principle A would look like:

(32) $\left[\text{SUBCAT } \boxed{1}\right]$

Constraints: if $\exists$X [X$\in\boxed{1}$ $\wedge$ CONTENT(X)=*ana* $\wedge$ $\exists$Y[*list* $\oplus$ Y$\oplus$*list*$\oplus$X$\oplus$*list*=$\boxed{1}$] $\rightarrow$ $\exists$Z[*list* $\oplus$ Z$\oplus$*list*$\oplus$X$\oplus$*list*=$\boxed{1}$ $\wedge$ INDEX(CONTENT(X))=INDEX(CONTENT(Z))]
and
if $\exists$X Y[X$\in$SUBCAT(CAT(Y)) $\wedge$ Y $\in\boxed{1}$ $\wedge$ CONTENT(X)=*ana* $\wedge$ $\exists$Y[*list* $\oplus$ Y$\oplus$*list*$\oplus$X$\oplus$*list*=$\boxed{1}$] $\rightarrow$ $\exists$Z[*list* $\oplus$ Z$\oplus$*list*$\oplus$Y$\oplus$*list*=$\boxed{1}$ $\wedge$
INDEX(CONTENT(X))=INDEX(CONTENT(Z))

Thus an implementation of principle A is possible, even though it exploits a very powerful formal apparatus. Indeed, the recursion problem is avoided in principle A by adopting the assuption that there are no verbal subject anaphors except from subjects of controlled VPs, which, anyway, can only constitute chains which can be locally checked. The same holds, at least in English, for personal pronouns. Thus, apparently, principle B can be formulated along the same lines:

(33) $\left[\text{SUBCAT } \boxed{1}\right]$

Constraints: if $\exists$X [X$\in\boxed{1}$ $\wedge$ CONTENT(X)=*ppro* $\rightarrow$ $\forall$ Y[*list* $\oplus$Y$\oplus$*list*$\oplus$X$\oplus$*list*=$\boxed{1}$ $\rightarrow$ INDEX(CONTENT(X))$\neq$INDEX(CONTENT(Y))]
and
if $\exists$X Y [X$\in$SUBCAT(CAT(Y)) $\wedge$Y $\in\boxed{1}$ $\wedge$ CONTENT(X)=*ppro* $\rightarrow$ $\forall$ Z[*list* $\oplus$Z$\oplus$*list*$\oplus$Y$\oplus$*list*=$\boxed{1}$
$\rightarrow$ INDEX(CONTENT(X))$\neq$INDEX(CONTENT(Z))]

The problem here is that there is no apparent way to get rid of the universal quantification, as lack of coindexation must hold for *all* the elements locally o-commanding X. In the absence of such a universal quantification no reasonable implementation of principle B is possible, unless one makes the (dubious) assumption that SUBCAT has a limited length.

Finally, it seems to be impossible to implement principle C within a declarative view of a framework for HPSG. Indeed to check if something is o-free (i.e. not o-commanded and coindexed) amounts to span a potentially unbounded structure negating all the possible coindexations. For instance (simplifying a bit and using X<Y as a shorthand for *list* $\oplus$Y$\oplus$*list*$\oplus$Y$\oplus$*list*=$\boxed{1}$):

(34) $\begin{bmatrix} \text{H-DTR} \mid \text{SYN} \mid \ldots \mid \text{SUBCAT} & \boxed{1} \\ \text{F-DTRS} \quad \boxed{2} & \end{bmatrix}$

Constraints: if $\exists$X [X$\in\boxed{1}$ $\rightarrow$ $\forall$ Y[Y$\in\boxed{1}$$\wedge$ X<Y $\wedge$ CONTENT(Y)=*npro*
$\rightarrow$ INDEX(CONTENT(X))$\neq$INDEX(CONTENT(Y))]]
and
if $\exists$X [X$\in\boxed{1}$ $\rightarrow$ $\forall$ Y [$\exists$Z $\exists$ W[Z $\in\boxed{2}$ $\wedge$ W<Y $\wedge$ Y HEAD(W)=HEAD(Z) $\wedge$ Y$\in$ SUBCAT(SYNSEM(Z)) $\wedge$ CONTENT(Y)=*npro*
$\rightarrow$ INDEX(CONTENT(X))$\neq$INDEX(CONTENT(Y))]]
and
if $\exists$X [X$\in\boxed{1}$ $\rightarrow$ $\forall$ Y [$\exists$Z $\exists$ W $\exists$ J [Z $\in\boxed{2}$ $\wedge$ W<Y $\wedge$ Y HEAD(W)=HEAD(Z) $\wedge$ Y$\in$ SUBCAT(SYNSEM(J)) $\wedge$ J$\in$ F-DTRS(Z) $\wedge$ CONTENT(Y)=*npro*

$\rightarrow$ INDEX(CONTENT(X))$\neq$INDEX(CONTENT(Y))]]

and
...

where '...' stands stands for an infinite number of conditional statements. Obviously, this is not to say that the the binding principle it is not implementable at all. Only, in order for it to be implemented, a language is needed where the possibility of defining recursive functions is retained.

3.2.7 Lexical Functional Grammar

See the LFG homepage: http://clwww.essex.ac.uk/LFG.

3.2.8 Tree Adjoining Grammar

Current TAG Formalisms

The TAG formalism was first presented in [Joshi *et al.*, 1975], for a more current introduction, see [Joshi, 1987a]. Its use as a formalism for natural language processing is described, e.g., in [Kroch and Joshi, 1985].

In a lexicalized TAG (LTAG, see [Schabes, 1990]), the core of a grammar is a set of *elementary trees*, describing the possible syntactic structures and a *syntactic lexicon* which contains stem forms associated with a list of elementary trees. The elementary trees might be grouped into *tree-families* which contain all elementary trees for a given subcategorization frame.

The mapping from full forms to stem forms might occur through a morphological lexicon which maps from full form to stem form by adding appropriate features, or through a separate module.

As an example, the architecture of XTAG is shown in figure 3.12.

An LTAG is said to be *fully lexicalized* (see [Srinivas *et al.*, 1994]) since there are no other knowledge sources (e.g. schemata) but for the elementary trees. There are only two simple combination operations, namely *adjunction* and *substitution*. The complete description of a sentence is derived by merging the elementary trees for the lexemes by adjunction and substitution. The operations are shown in figure 3.8 and 3.9.

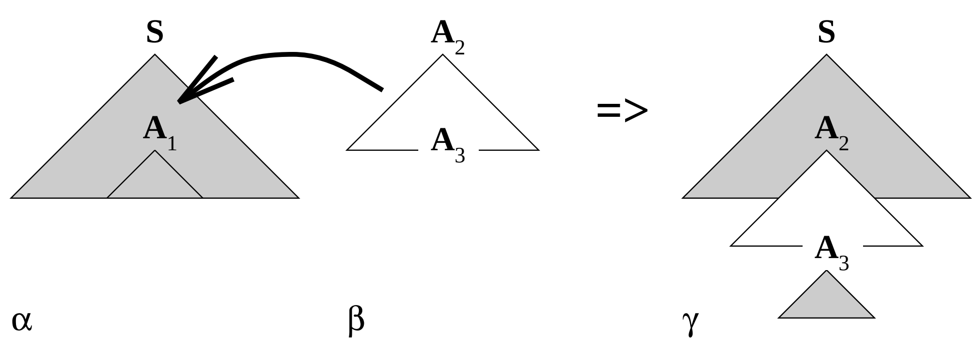


Figure 3.8: The adjunction operation.

In addition to the category labels, the nodes carry information in a feature structure. Such feature-based TAG (FB-TAG) were introduced in [Vijay-Shanker and Joshi, 1988]. To every node two feature structures (FS) are added, a *top* and a *bottom* FS. These FS are restricted in that the list of possible features and values is finite and fixed in advance, allowing no unbounded recursion in the FS. In the course of a derivation, the features are extended monotonically. Figures 3.10 and 3.11 show the unification of features for the adjunction and substitution operations.

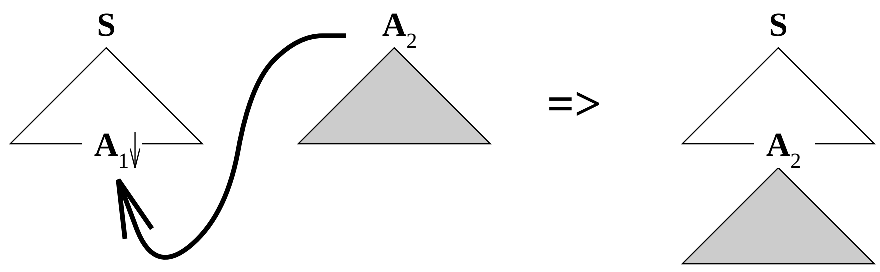


Figure 3.9: The substitution operation.

An alternative approach, named U-TAG (see [Buschauer *et al.*, 1991]), was originally adopted for TAG-GEN. Here, every node is extended only by a single FS. In the course of a derivation, these FSs are broken up and reassembled in a non-monotonic fashion.

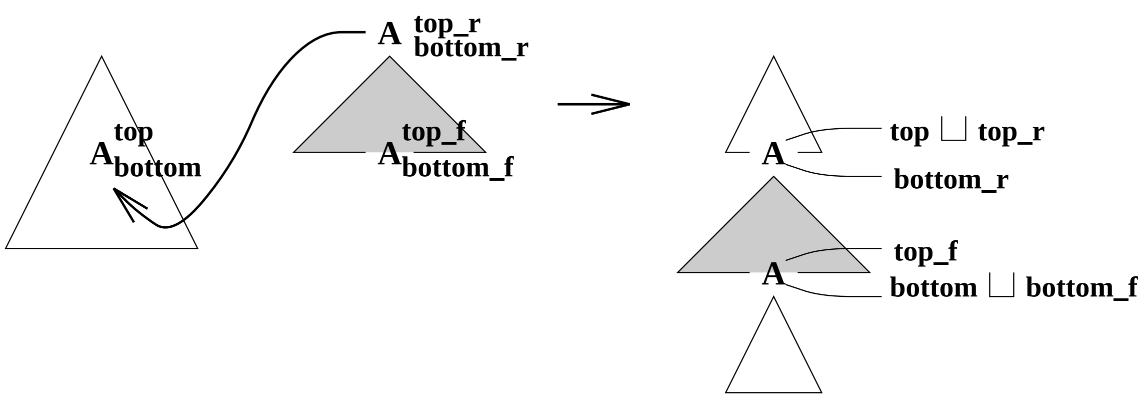


Figure 3.10: Unification with adjunction.

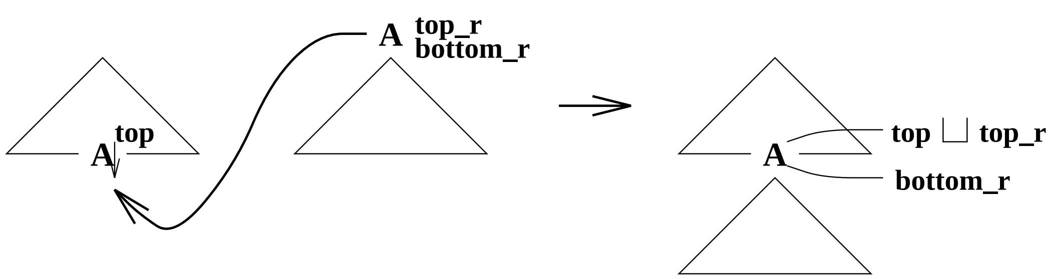


Figure 3.11: Unification with substitution.

There have been a number of approaches to a compact representation of TAG grammars which would also allow the linguistic generalizations implicit in the structure of the elementary trees to be expressed. Such work can be found in [Vijay-Shanker and Schabes, 1992; Becker, 1993; Evans *et al.*, 1995; Becker, 1995]. Even though some of this work is implemented (e.g. as part of the XTAG system), it has not been used extensively yet and the proposed common specification language will not deal with these extensions.

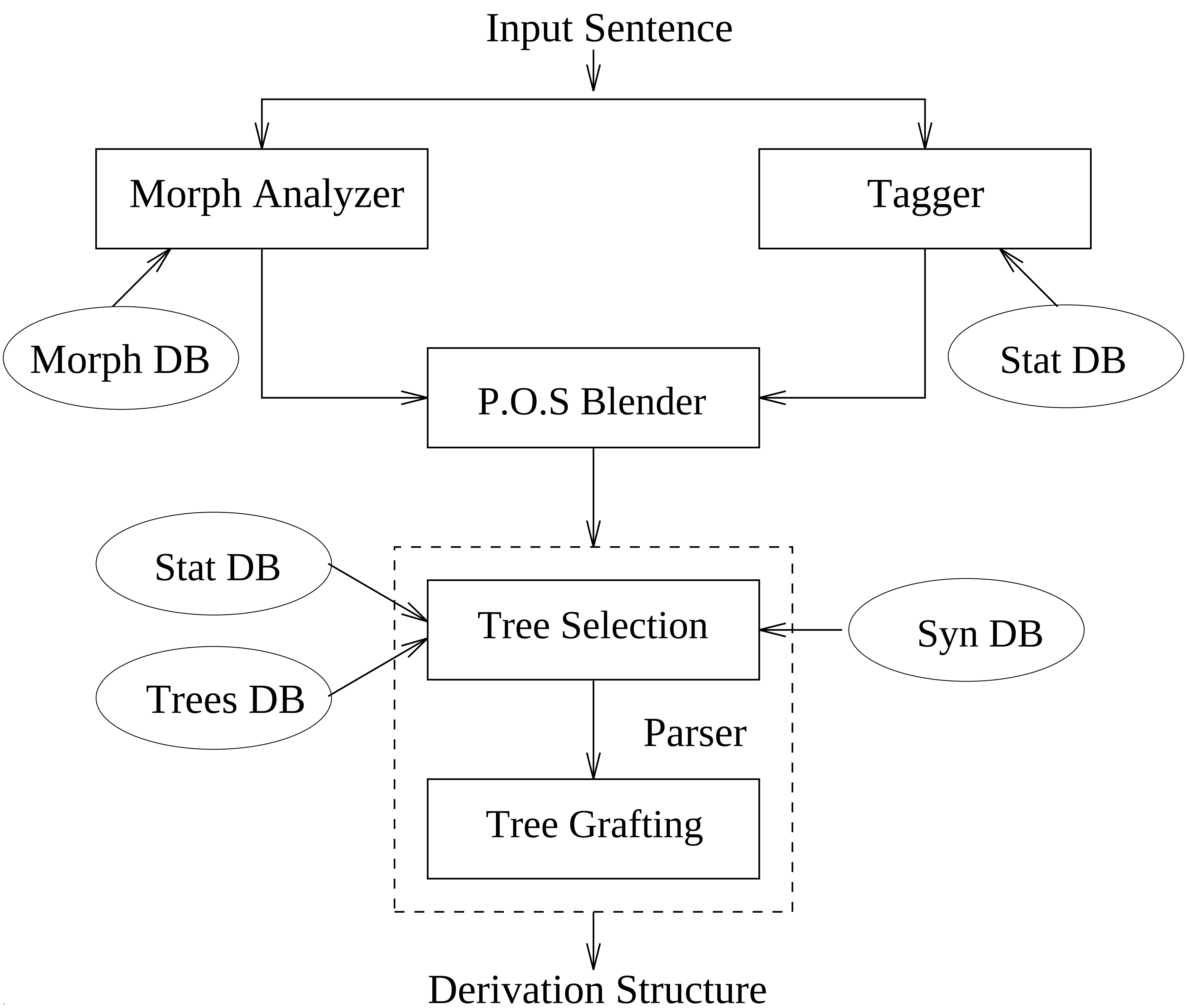


Figure 3.12: Architecture of the XTAG system.

The XTAG and TAG-GEN Projects

XTAG is an on-going project to develop a wide-coverage grammar for English, based on the Feature-Based Lexicalized Tree Adjoining Grammar (FB-LTAG) formalism.

TAG-GEN is a syntactic generator that exploits an incremental and parallel processing scheme. Incrementality is supported by a distributed, parallel model of active cooperating objects. They verbalize the incremental given input in a lexically guided two-level system, first building the hierarchical structure and then computing the serial order of words in the sentence under construction. Tree Adjoining Grammars are used as syntactic representation formalism and have demonstrated their adequacy in supporting incremental processing. TAG-GEN is based on the non-monotonic UTAG formalism.

TAG-GEN is the predecessor of VM-GEN. The current version of VM-GEN as part of VM-GECO (see [Kilger and Finkler, 1995]) has switched to the monotonic feature structures of FB-TAG, facilitating the exchange of grammars between the systems.

3.3 Mathematical Formalisms

3.3.1 Extensions

From 93 until 95, some important additional constraints have been introduced and/or investigated in detail. An important constraint is the *arity constraint*, which allows a finite set of features to be specified that are admissible for a corresponding node [Smolka and Treinen, 1994]. If two different nodes have the same arity and the same elements under the corresponding features, then the two nodes must be equal. This property is referred to as extensionality. For interpreting the feature description using arity constraints, a new interpretation consisting of feature trees has been introduced [Aït-Kaci *et al.*, 1994; Smolka and Treinen, 1994]. It was shown, that this interpretation is canonical for satisfiability of feature description extended by arity constraints [Smolka and Treinen, 1994]. Furthermore, it was shown in this paper that one can translate formulae in the language of constructor trees (as used by PROLOG-III) into the extended language, such that the validity in the canonical models is preserved. Note that this does not hold for the feature graph interpretation of feature logic extended by arity constraints (since the extensionality does not hold in this model), which was one of the main reasons for introducing the new model. Another reason is that feature trees are mathematically simpler, and for this reason more flexible.

The properties of different first-order languages over feature trees has been investigated in detail in [Backofen, 1994a]. One important aspect in this work is the comparison of the expressivity of the different languages. It was shown that the language F [Treinen, 1993], which allows for quantification over features, has a very high expressivity. For example, set, subsumption and functional uncertainty constraints are encodable in this language. It is even possible to encode all relations that are definable via definite equivalences. Since most of the proposed extensions to feature logic are definable via definite equivalences, this language (together with the feature tree interpretation) can be used as a "general" feature description language.

Set constraint systems have been investigated in detail by [Manandhar, 1993a]. He considers a feature description language, where features are not necessarily functional (i.e., the value of a feature can be a set instead of a singleton). As additional constraints he introduces set union and intersection, membership and sets with fixed cardinality. This is done by connecting this language with the KL-ONE like concept languages. He also provided a language that integrates both a KL-ONE like concept language and feature descriptions.

Subsumption constraints together with a weaker form have been investigated in [Dörre, 1993]. By and large, a subsumption constraint $x \sqsubseteq y$ specifies that whenever a path p is defined on x, then p must also be defined in y, and whenever two paths p, q lead to the same node in x, then p, q must lead to the same node in y. A weak subsumption constraint between x and y requires only that all paths defined on x must also be defined on y. This constraint suffices in many cases. However, weak subsumption constraints have much nicer computational properties than subsumption constraints.

More effort was devoted to tree constraint systems. In contrast to the above constraint systems, a tree constraint system is no feature description language. It allows the overall structure of trees to be partially specified using dominance and precedence relations. There have been formalisations of these constraints as first-order languages and as modal languages (e.g., [Rogers and Vijay-Shanker, 1994; Backofen *et al.*, 1995; Blackburn *et al.*, 1993; Blackburn and Meyer-Viol, 1994]). Tree constraint systems are the basis of a theoretical treatment for TAG-like grammar formalisms.

Beside new constraint systems, higher fragments have been consider for extending the expressivity. An important example is the fragment corresponding to the entailment check [Aït-Kaci *et al.*, 1994; Smolka and Treinen, 1994]. A formula ϕ entails a formula ψ in some interpretation $\mathcal{A}$ if every valuation in $\mathcal{A}$ that makes ϕ valid also makes ψ valid. Or equivalently, ϕ entails ψ iff

$$\mathcal{A} \models \tilde{\forall}(\phi \rightarrow \psi),$$

where $\tilde{\forall}$ abbreviates the universal quantification of all free variables of ϕ and ψ. Hence, testing entailment is the same as testing specific formulae in a fragment of feature logic that allows a limited use of universal quantification (note that unification is the same as testing purely existensional quantified formulae). In contrast to satisfiability, there is no canonical model. Hence, we have to restrict ourself to one model for testing entailment (which is usually the model canonical for satisfiability). Since there at least two different models of feature logic that are canonical for satisfiability, the question arises how these models related to each other (and whether they lead to different theories).

3.3.2 Semantic Foundations

In the last two years, there have been two main streams of formalisations, namely the ones using first-order logic, and the ones using modal languages. We have also seen some convergence between these groups. Firstly, many results given for the first-order approaches have their equivalent in the modal approaches and vice versa. Secondly, modal languages can be translated via a standard translation into second-order logic. For many modal languages, this translation even yields a first-order formula, and that is true of many modal approaches to feature logic. For the remaining cases, it is often possible to eliminate the second-order quantifiers, thus yielding first-order formulae having the same first-order theory as the initial second-order formula. There is increasing interest in the modal community in such quantifier elimination methods. This lead to a situation where first-order approaches and modal approaches are seen as two different views of the same thing, which are to some extent convertible.

3.3.3 Results

Here is a list of some important results that have been obtained in the last two years (certainly NOT a complete list):

- quasi-linear algorithms for entailment checking [Aït-Kaci *et al.*, 1994; Smolka and Treinen, 1994]

- decidability of a calculus for set constraints [Manandhar, 1993c]; this calculus is based on a calculus provided for KL-ONE-like languages; for a sub-language, even NP-Completeness was shown [Manandhar, 1994]

- the equivalence of the feature graph interpretation and the feature tree of the basic feature description language; this result was achieved by providing a complete axiomatisation for these theories; the completeness proof exhibits decision procedures for valid formulae (which can e.g. be used for deciding entailment of formulae that use negated coreferences) [Backofen and Smolka, 1995; Backofen, 1994a]

- complete axiomatisation of the theory of feature trees in the language of feature logic extended by arity constraints (which also yield a decision procedure) [Backofen, 1995; Backofen and Treinen, 1994]

- proof of NP-completeness of the existential fragment of feature logic where we allow quantification over features (i.e., where feature are first-class values) [Treinen, 1993]

- partial decision algorithm for type systems (over a restricted constraint system without disjunction) with greatest fixpoint semantics; [Aït-Kaci *et al.*, 1993]

- feature logic and some other constraints can be formulated in a fragment of first-order logic known as the Schoenfinkel-Bernay fragment, which is know to be decidable [Johnson, 1991; Johnson, 1994]

- axiomatisations of trees (fixed and unbounded arity) in first-order logic [Backofen *et al.*, 1995] and (for the fixed arity) in modal logic [Blackburn and Meyer-Viol, 1994];

- decidability of functional uncertainty constraints with cycles [Backofen, 1994b; Backofen, 1994a]
- decidability of weak subsumption constraints (even with negation) [Dörre, 1993]
- complete and sound deduction systems for the combination of feature logic and Lambek calculus [Dörre and Manandhar, 1995]

3.4 Implemented Formalisms

3.4.1 Formal Devices Used in Grammar Formalisms

Grammar formalisms are implementations which allow (largely) declarative specification of grammars together with a collection of tools such as parsers, generators, morphological processors *etc.* that make use of the grammar. Grammar formalisms provide means for detecting inconsistency within grammars at compile-time and means for debugging grammars at run-time.

For grammar formalisms there is a choice between expressing grammars in a general form as Horn clauses or in a special grammar rule notation. For instance, within the CUF formalism, grammar rules are not given any special status and are expressed as Horn clauses. On the other hand, within the ALE formalism, grammar rules are expressed as phrase-structure rules and a chart parser is employed. Usually, the former corresponds to the choice of a general deduction system, whereas the latter corresponds to the use of specialised parsers or generators (e.g. chart parser, semantic-head driven generator).

There is a convergence between these two approaches, because often parsing algorithms can be generalised as deduction algorithms, and because methods from logic programming, such as constraint solving, are being imported into NL parsing and generation.

Also there is a trend in grammatical theory away from construction-specific rules towards general well-formedness conditions (grammatical principles), which can be naturally expressed as constraints that need to be satisfied universally. This can be evidenced within the so called *principle-based* grammatical theories such as the *government-binding theory* [Chomsky, 1981; Chomsky and Lasnik, 1991] and HPSG [Pollard and Sag, 1994b].

Constraint Logic Programming

In logic programming — following early work such as CHIP — and the development of the CLP scheme [Jaffar and Lassez, 1987] [Höhfeld and Smolka, 1988], Constraint Logic Programming has become an important paradigm and an area of much active research. This CLP scheme has been adopted in NLP in systems such as CUF [Dörre and Dorna, 1993]. The CLP scheme also permits the addition of new constraint solvers in addition to feature constraints and this approach is currently being experimented with in formalisms such as CL-ONE [Erbach *et al.*, 1995].

Typed feature formalisms

A lot of effort within the NLP community has been directed towards a class of grammar formalisms known as *typed feature formalisms*. Broadly speaking, typed feature formalisms are Horn-extensions over feature descriptions. They can loosely be understood as specific instantiations of the CLP(FT) paradigm where FT stands for a specific version of feature logic [Kasper and Rounds, 1986] [Smolka, 1992] with types and inheritance. Languages such as LOGIN and LIFE known within the logic programming community and systems such as ALE, CUF, TFS, ALEP known within the NLP community are examples of typed feature formalisms.

Resolution strategies

The search strategy of Prolog (top-down, left-to-right, depth-first, backtracking) has turned out to be problematic for natural language parsing and generation because of termination and efficiency problems.

As a consequence, a number of alternative deduction strategies have been developed for logic grammars, partly as genuine deduction algorithms, and partly as adaptations of existing parsing algorithms (such as *bottom-up* and *left-corner* parsing) for context-free grammars or of existing generation algorithms (such as *head-driven generation*).

For example, CUF is a formalism which follows a CLP scheme and employs the Andorra resolution strategy [Haridi and Janson, 1990], which prefers deterministic goal reduction. On the other hand, systems such as ALE employ a bottom-up chart parser.

Bottom-Up Approaches

Pure top-down resolution is often problematic for NLP because it leads to a large amount of search and to termination problems, especially for grammars which make use of discontinuous constituents. Therefore NL parsers and generators often make use of bottom-up methods, e.g., left-corner parsing and its compilation into a Prolog program (BUP) [Matsumoto *et al.*, 1983], semantic-head driven generation [Shieber *et al.*, 1990], head-corner parsing [van Noord, 1991]. The advantages of bottom-up and top-down processing are often combined by making use of a precompiled *reachability relation* [Kay, 1980].

In logic programming, bottom-up approaches are found, for example, in the area of deductive databases.

Compilation and program (grammar) transformation

Partial deduction (or partial execution) techniques have been used to transform a grammar (or program) into an equivalent, but more efficient, one in which some goals have been replaced by their defining clauses. Explanation-based learning is a partial deduction technique that has been used to speed up NL parsing and generation by replacing sequences of frequently occurring rule applications (inference steps) by one compiled rule [Neumann, 1994a; Samuelsson, 1994a].

Approaches derived from LR-parsing compile a grammar into a condition-action table which has the benefit that different rules (or clauses) with a common prefix don't lead to separate threads in the computation. LR-based techniques have been applied to parsing and generation [Samuelsson, 1994b; Samuelsson, 1995].

Termination problems can arise when the same program (logic grammar) is used for parsing and generation because the termination is dependent on the instantiation of arguments and the flow of information between different goals of a clause.

The *Essential Argument Algorithm* [Strzalkowski, 1991] distinguishes input and output arguments of DCGs, and uses re-ordering of goals within and across clauses in order to derive grammars that are specialised for parsing or generation, in the sense that the input arguments (the semantics in case of generation, and the surface string in case of parsing) are always guaranteed to be instantiated for the respective direction of processing.

Tabulation

Due to the amount of search involved in NLP, Prolog's backtracking search is rather inefficient, and tabulation (memoing) techniques should be used in order to store and re-use partial results.

The earliest application of tabulation in logic programming is Earley Deduction [Pereira and Warren, 1983], which adapts the basic idea of Earley's context-free parsing algorithm as a proof procedure for logic programs, and especially DCG parsing. Refinements include the restriction technique to avoid prediction loops [Shieber, 1985], the use of the algorithm for both parsing and generation through a modification of the indexing scheme [Shieber, 1988; Neumann, 1994b], extensions for the memoisation of coroutined constraints [Johnson and Dörre, 1995], and a bottom-up variant [Erbach, 1994a].

Only recently have memoisation techniques (OLDT resolution) been incorporated into logic programming languages such as XSB Prolog [Warren, 1992]. This is an important development for NLP because traditional Prolog-based implementations of chart parsers suffer from a loss of efficiency due to the copying of structures when items are stored. Further research into more efficient memoisation schemes is probably needed.

Feature Structures

In computational linguistics, feature structures (attribute-value matrices) have for a long time been preferred over Prolog terms (e.g. PATR [Shieber *et al.*, 1983] or Lexical Functional Grammar [Kaplan and Bresnan, 1982]) because they allow a more perspicuous and compact encoding of linguistic knowledge. The encoding is more perspicuous because feature names indicate what the purpose of a particular value is, and it is more compact because values can be left unspecified by omitting features instead of introducing anonymous variables.

Feature structures can be *open* or *closed*. Open feature structures allow more features to be added to a structure, whereas closed feature structures allow only a fixed set of features. Closed feature structures are generally used in typed feature structures, where a fixed set of features is declared to be appropriate for a particular type. A good overview of feature constraint logics for NLP is given in [Smolka, 1992].

In logic programming, feature constraints have become fairly common. Following the pioneering work of Aït-Kaci [Ait-Kaci, 1984; Ait-Kaci and Nasr, 1986a], they have been introduced into various logic programming languages. Efficient implementation methods exist for open and closed feature structures, and have been integrated in the Oz language [Smolka *et al.*, 1995].

Types and Typed Feature Structures

Types and inheritance between types are important devices for the hierarchical structuring of linguistic knowledge, and are widely used in influential linguistic theories such as head-driven phrase structure grammar [Pollard and Sag, 1994b]. However, the notions of type (or sort) can mean different things in different grammatical formalisms, which can be a source of confusion.

Single inheritance is generally considered insufficient since it does not allow cross-classification of linguistic descriptions. Most typed feature systems generally provide some form of multiple inheritance. For instance, Carpenter's formalisation of typed feature structures [Carpenter, 1992a] requires the inheritance lattice to be specified as a BCPO (bounded complete partial order). On the other hand, the ProFIT typed feature system allows either single inheritance or multi-dimensional inheritance which can be employed for cross-classification [Erbach, 1994b].

Some expressive formalisms, such as TDL [Krieger and Schäfer, 1994], use a more expressive type system with relational types, full negation, lazy type expansion etc. A system such as CUF allows type axioms to be stated using the full propositional calculus. However, CUF does not permit variables in type definitions.

The choice of an ideal type system for NLP is subject to debate and will ultimately depend on the needs of the representation of linguistic knowledge. At this stage, it would be premature to decide on one type system.

Higher-Order Unification

A standard approach for treating ellipsis in NLP makes crucial use of higher-order unification [Dalrymple *et al.*, 1991]. For example, in the sentence *Dan likes golf, and George does too* the first conjunct will have the logical form *like(dan,golf)*, and for the second, elliptical, conjunct we know that the property P predicated of Dan in the first conjunct holds of George.

$$P(dan) = like(dan,golf)$$

A possible value for P is the property $\lambda x.like(x,golf)$. Then $\lambda x.like(x,golf).george$ reduces to *like(george,golf)*.

Constraints

Different types of constraints play an important role in grammatical descriptions. Equality or co-reference constraints are most important within computational linguistics, and have led to the designation of a whole class of grammar formalisms as *Unification Grammars*, which are the forerunners of typed feature formalisms.

Inequality constraints (cf. *dif* in Prolog II or Sicstus Prolog) are for example needed to model aspects of the binding theory, e.g. in the sentence *John saw him*, the pronoun *him* cannot be coreferent with the NP *John*.

NL processing demands specialised constraint processing facilities for both high-level declarative encodings of grammars and for efficient processing. However, as opposed to specialised constraint solving in the numeric domain which appears to be the trend within constraint logic programming evidenced by formalisms such as CLP(R), CLP(Q) and CLP(FD), NL processing usually demands specialised constraint processing in the symbolic domain.

Symbolic constraints such as *set descriptions and set operations* [Manandhar, 1994], *linear precedence constraints* [Manandhar, 1995], *guarded constraints over these constraints* [Erbach *et al.*, 1995], *tree descriptions* [Vijay-Shanker, 1992] are among those that have been investigated.

Set descriptions permit multiple values for the same feature symbol in the same way as knowledge representation languages such as KL-ONE [Brachman and Schmolze, 1985][5]. Set constraints are widely used in linguistic theories to model phenomena whose handling involves storing and passing information about possibly large collections of certain linguistic objects such as subcategorisation frames, discourse referents, quantifiers, extraposed or fronted constituents etc. A set constraint language has been implemented which makes use of membership constraints, cardinality constraints, disjointness constraints, and union and intersection constraints. A sound and complete constraint solver for these constraints has been implemented [Manandhar, 1994].

Linear precedence constraints allow an underspecified representation of the order among the elements of a set [Manandhar, 1995]. Unlike lists, where the order is completely fixed, or sets, where the order is completely open, linear precedence constraints allow one to specify which elements must stand in a precedence relation. These constraints are used for modelling partially free word order in languages such as German or Bulgarian, and also for underspecified representation of quantifier scoping possibilities.

With the introduction of attributed variables [Holzbaur, 1992], it has become possible to implement constraint solvers in systems such as Sicstus 3. This mechanism is very useful for NLP because it permits the experimentation with and application of new kinds of constraint solvers (e.g. the handling of grammatical principles as type constraints [Matiasek, 1994], or morphology as a constraint between surface and lexical forms [Trost and Matiasek, 1994]).

Disjunction

Disjunction handling is an important problem for NLP due to the lexical and structural ambiguity inherent in natural language [Trost, 1993]. The creation of choice points for these cases of ambiguity would lead to serious efficiency problems.

There are two approaches other than backtracking to disjunction handling in grammar formalisms: distributed disjunctions and finite domains.

Distributed Disjunction

Distributed disjunctions [Dörre and Eisele, 1990; Böttcher, 1993; Maxwell III and Kaplan, 1991] implement a full disjunction handling by avoiding expansion to disjunctive normal form by pushing disjunctions as deep into the feature structures as possible. In order to deal with the fact that the choice in one disjunct requires a corresponding choice in another disjunct, the disjuncts are given names, and the choice of one named disjunct leads to the choice of the disjunct with the same name in another part of the structure.

An example is the relationship between particular syntactic forms and semantic readings, as in German where the combination of the preposition *in* with an dative NP has a locative reading (English: *in*), whereas the combination with an accusative NP leads to a directional reading (English: *into*).

Compared to the relatively inefficient implementation of distributed disjunctions on top of Prolog, their re-implementation in C, which is used in the XEROX Language Engineering system (XLE), is very efficient.

[5] See [Manandhar, 1993a] for a comparison between feature logics and terminological logics employed in knowledge representation languages such as KL-ONE.

Finite Domains

Finite domains are useful where disjunctions are not distributed over different parts of the structure. A finite domain variable is a variable which can take on a finite set of ground values. As an example, take the following finite domain which consists of the possible combination of the agreement features NUMBER and PERSON in English:

{[1st,sg],[2nd,sg],[3rd,sg],[1st,pl],[2nd,pl],[3rd,pl]}

A word form such as *takes* can simply be described as taking a subject with the agreement value [3rd,sg], but for the form *take*, there are five different possibilities:

{[1st,sg],[2nd,sg],[1st,pl],[2nd,pl],[3rd,pl]}

In this case, the use of finite domains can avoid the creation of choice points and contribute to efficient processing. In Prolog-based systems such as ProFIT [Erbach, 1995], finite domains with N possible values are implemented by compilation into a term with $N+1$ arguments [Mellish, 1988; Mellish, 1992].

Underspecification

Underspecification techniques have been used to avoid disjunctions by constructing type hierarchies or term representations, in which one term can stand for a set of values and a choice among these values takes place by further instantiation of the term. Underspecification techniques are especially important for problems in semantic representation such as the encoding of quantifier scoping possibilities [Alshawi and Crouch, 1992].

Lean Formalisms

The implementation of typed feature structures, constraints, disjunction handling, and alternative deduction strategies on top of a high-level language such as Prolog leads to a serious loss of efficiency, which hinders the development of NLP applications which make use of these formalisms. In order to overcome these problems, some recent formalisms (e.g. the Core Language Engine CLE [Alshawi, 1992], the Advanced Linguistic Engineering Platform ALEP [Alshawi *et al.*, 1991], ProFIT [Erbach, 1995]) have made use of a compilation of typed feature structures into Prolog terms, so that Prolog's term unification can be used at runtime. These systems have been successfully used for representing linguistic knowledge for NLP applications. However, these systems do not allow currently support multiple inheritance.

In order to handle multiple inheritance, the Attribute Logic Engine ALE [Carpenter and Penn, 1994] therefore takes a different approach by compiling typed feature structure unification into a set of primitive instructions [Carpenter, 1993a]. This approach is currently implemented on top of Prolog by making use of open data structures and efficient indexing techniques, but a re-implementation in a lower-level language using an abstract machine approach is planned in order to speed up the system.

Lean formalisms are important for NLP systems to enable implementations of large-scale systems and for downsizing research prototypes into commercial products.

Template Notation

Template definitions provide the possibility of giving names to terms, and of referring to frequently occurring terms by that name. For example, the template *np* can stand for the complex term representing a noun phrase. Templates can have arguments, and an n-place template is equivalent to an $n+1$-place relation. While templates are a logically redundant notation, they are considered important by grammar writers because they permit the definition of complex terms in one place and refer to them by their name on all occurrences, which ensures transparency and facilitates the maintenance of a grammar.

Preference-driven Constraint Satisfaction

In order to deal with the ambiguous readings which occur very often with NL (with which humans seem to have no difficulties), newer processing models make use of a preference-driven best-first search strategy (where the preferences are acquired using a statistical model).

Within CLP based systems, this can be implemented through a dynamic goal selection function which for a selected atom chooses clauses with the highest preference first [Uszkoreit, 1991]. In future language processing systems such best-first selection strategies are likely to play an important role.

An equally important problem is the handling of ill-formed (ungrammatical) input as it occurs in applied systems. In order to handle this problem, where not all constraints of the grammar can be satisfied in parsing a sentence, future constraint-based linguistic processing will require dynamic and preferential constraint satisfaction. Thus research in areas such as *hierarchical constraint logic programming* [Wilson and Borning, 1993] has potential applications for such NLP tasks. HCLP can be thought as an extension of the CLP(X) paradigm to CLP(X,S) where X represents the constraint domain and S represents the *comparator* or *selection function* that can handle preferences in case of conflicts.

3.4.2 Relationship between Logic Proramming and Grammar Formalisms

The early development of Prolog and logic programming was strongly influenced by NLP applications (Colmerauer's Q-Systems and Metamorphosis Grammar), and DCGs have been part of Prolog from the very beginning. Today, logic programming is the most important programming paradigm for NLP, and the strong relationship between the two fields is evidenced by regular conferences (e.g., *Natural Language Understanding and Logic Programming*) and workshops (e.g., *Computational Logic and Natural Language Processing*, jointly organised by the European Network in Computational Logic and the European Network in Language and Speech).

A large part of the work in NLP is based on Prolog. A key idea is to express a grammar as a set of statements in a logic (e.g. Horn clauses), and to use a theorem prover (e.g. resolution) in order to parse or generate NL sentences. The original slogan *parsing as deduction* [Pereira and Warren, 1983] is nowadays extended to *bidirectional linguistic deduction* in which the same grammar can be employed both for parsing and generation. However, Prolog has turned out to be insufficient for the purposes of NLP for a number of reasons (inefficient search strategy, unexpressive constraint language). As a consequence, research in grammar formalisms has come up with a number of alternatives, extensions or specialisations of the design choices made in Prolog. One important point is that most grammar formalisms make a clear distinction between declarative specification (grammars) and execution (NL parsing or generation). The following section gives an overview of the state of the art in grammar formalisms, with special emphasis on those developments that are different from developments in logic programming (although there is a large overlap, where the fields have either influenced each other or the same kinds of approaches have been developed independently).[6]

3.4.3 Requirements of NLP for a Programming Language

In this section, we outline the requirements for a declarative successor to Prolog from the perspective of the needs of Natural Language Processing. Within NLP, there are two separate kinds of activities:

1. The development of formal models of NL syntax and semantics (Computational Linguistics)
2. The implementation of NLP components for applied systems (Language Engineering)

[6] Due to space limitations, this paper can only give a very broad overview and references to the relevant literature. A more extensive overview of grammar formalisms is in preparation. Also, we concentrate the bibliographic references on the work in grammar formalisms, since we expect the reader to be familiar with the relevant literature in logic programming.

Both kinds of activities are closely related, and make use of the same declarative core, but need different kinds of support services from a programming language:

For *Computational Linguistics*, it is important to have the best support for developing, testing, debugging grammars and for formalising and evaluating linguistic theories. The emphasis on developing linguistic descriptions, expressing generalisations about linguistic phenomena, and experimenting with new linguistic theories and new descriptive devices imposes the following requirements on a logic programming language, which can support the building of tools for these activities:

- declarativeness
- an expressive constraint language with a convenient notation
- the possibility to add new kinds of constraint solvers
- tools for debugging and visualisation (e.g. Tcl/TK interface, as it is provided in Sicstus and Oz).

For *Language Engineering*, on the other hand, computational efficiency is a very important consideration because systems must either respond with a minimum of waiting time (dialog systems) or be able to handle large amounts of data (text understanding, message extraction).

A second important consideration for Language Engineering is the need to make a selection among the large number of possible solutions of a particular parsing or generation task. The chosen solution should be the preferred (most probable) reading in the case of parsing, or the sentence that is optimal for achieving the desired communicative effect in the case of generation.

While it is not yet clear what the optimal model for ranking the possible solutions is, there is currently a strong trend towards statistical language models (which often still use very primitive models such as bigram or trigram statistics). In Computational Linguistics research, this new trend is also reflected in the shift from competence models to performance models [Uszkoreit, 1991]. A programming language for natural language must support research into the integration of declarative logical models and statistical language models.

In summary, a programming language which supports Computational Linguistics research and the development of applied NLP systems must satisfy the following requirements:

- **Declarative Core**
  1. feature constraints (in addition to terms), cf. LIFE, Oz.
  2. finite domains (not only over integers)
  3. higher-order unification and higher-order programming
  4. support for different type systems
- **Resolution Strategies**
  1. support for preference-driven constraint satisfaction
  2. efficient disjunction handling
  3. the possibility to add new types of constraints (such as sets, trees, linear precedence, regular expressions)
  4. support for different search strategies (bottom-up, head-driven, best-first etc.)
  5. tabulation support
- **Support Services**
  1. support for 'programming in the large' (e.g., modules, integration with imperative, object-oriented and functional programming)

2. embeddability into other systems (e.g., database support for large lexicons, compilation to machine-code which can be linked with other system components)

As a whole, the programming language should provide efficient implementations of the basic services, such as feature constraints, tabulation, disjunction, or search engines, so that a grammar formalism can be built on top of it without a lot of computational overhead, and on the other hand be flexible enough to offer the possibility to pursue interesting research directions, e.g., in the combination of declarative with statistical language models, or the addition of new types of constraints.

3.4.4 ALE

Since 1993 the version 2.0 has replaced the beta version of ALE. As far as the functionality of the system is concerned, there have not been any substantial changes to the kernel of the formalism. The main achievement in this period has been an increase in the robustness of the system, and the development of interfaces to other linguistic modules, for example, a new morphology component developed in Edinburgh, or the extended version of the formalism used in the area of machine translation (SNALE i.e. Sharp New ALE, developed at Sharp). An integration within the PLEUK grammar development system (2.5.4, 3.5.4) has also been made possible.

A substantial revision of the formalism that introduces the following features is being prepared:

- Maximality and bottom defaults: all types not occurring on LHS of a sub declaration are assumed to be maximal; all types not occurring on RHS of a sub are assumed to be immediately subsumed by bottom.
- More compiler-time error messages
- Edge subsumption option: if any edge in the parsing chart is subsumed by another edge spanning the same nodes, then it is removed.
- Most general satisfier compiler: this makes type inference and unification faster.
- Various other improvements to Prolog code have been made to speed up the system.
- 'Functional' constraints: i.e. one can define an append(X,Y) that evaluates to a feature structure for Y appended to X, rather than having to define a boolean append(X,Y,Z), where Z is Y appended to X.

Complete documentation (running to 100 pages, with examples of everything, programming advice, and sample grammars), is available as:

> Bob Carpenter (1992) ALE User's Guide. Carnegie Mellon University Laboratory for Computational Linguistics Technical Report. Pittsburgh.

ALE can be run in either SICStus or Quintus Prolog, and with other compatible compilers doing first-argument indexing and last-call optimization. The system and its documentation are available without charge for research purposes.

The release of the new version is planned for May.

3.4.5 ALEP

Since the beginning of EAGLES the ALEP formalism has not undergone any major changes. For information about the ALEP formalism please refer to section 2.4.2.

3.4.6 CL-ONE

The formalism is designed for the direct representation of linguistic theories such as HPSG by providing set descriptions and set constraints and linear precedence constraints.

CL-ONE is a fully implemented grammar formalism based on the ProFIT extension to Prolog. ProFIT extends Prolog with features, multiple inheritance, and templates. ProFIT follows the tradition of the Core Language Engine and the ALEP formalism in compiling typed feature terms into a Prolog term representation.

CL-ONE adds sets and linear precedence constraints, as well as a parser and a generator. Set constraints include:

- maximal cardinality set description
- fixed cardinality set description
- set membership
- union
- intersection
- disjointness
- forall constraints

Other set constraints can be defined from the constraints given above; for example, disjoint union can be defined by the conjunction of disjointness and union constraints. To our knowledge, this is the first implementation of set constraints which handles all of the above mentioned set constraints and does not generate redundant solutions.

Linear precedence constraints make it possible to underspecify the actual constituent order. They can be used to implement analyses of constituent ordering like Reape's [Rea, 1994] domain union operator without enumerating all possible orders.

In addition to a parser, the system includes an interpreter for Prolog programs with typed feature terms and set descriptions.

HPSG grammars making use of sets and linear precedence constraints have been implemented in this formalism.

The development of the formalism has been supported by the Commission of the European Communities through the project LRE-61-061 "Reusable Grammatical Resources" which aims at providing implementations of datatypes for linguistic descriptions for integration into the ALEP grammar development environment.

In order to use CL-ONE you need Sicstus Prolog 2.1 #9 or higher, preferably on a Unix workstation. More information about CL-ONE is available through the web:

`http://coli.uni-sb.de/claus/claus62.html`

3.4.7 CUF

The current CUF system (Version 2.28) consists of a compiler, a runtime evaluator and an ASCII and a graphical user interface (GUI)[7] with several development tools such as a debugger, a data base inspector, and a feature structure browser. The implementation runs under Quintus and SICStus PROLOG under UNIX and X11.

The incremental compiler is used to translate the CUF desriptions into an interpretable format. Type checking and inference is used to eliminate errors very early in the development phase of a description. The most distinguishing features of CUF's type system are:

[7] Currently, the GUI is still under development and not delivered yet. However, the ASCII interface provides main functionalities of the GUI.

- the fact that type interdependencies can be stated in full propositional logic, allowing to state all kinds of type hierarchies
- features may be fully polymorphic (no restrictions on multiple feature declarations)
- complete type checking during compilation
- runtime type checking is reduced to a minimum

CUF makes a clear distinction between the purely declarative logical specification and the control statements which are used to guide the proof procedure without compromising the logical semantics of the specification. The runtime evaluator is an SLD-resolution engine whose selection strategy can be customized by the user. By default the strategy selects deterministicly expandable literals first, or else the leftmost (nondeterministic) literal. By use of delay statements the user can change this behaviour. Another type of control statement is the declaration of predicates for which the system should build an index.

The system CUF is freely available. Simply fetch it via anonymous ftp from

`ftp://ftp.ims.uni-stuttgart.de:/pub/cuf`

or write to cuf-request@ims.uni-stuttgart.de or to:

Jochen Dörre
Institut für maschinelle Sprachverarbeitung
Universität Stuttgart
Azenbergstr. 12
D-70174 Stuttgart, GERMANY

email: Jochen.Doerre@ims.uni-stuttgart.de

3.4.8 ProFIT

The ProFIT system is an extension of Prolog with sorted feature structures (including multi-dimensional inheritance), finite domains, feature search, cyclic terms, and templates. The acronym ProFIT stands for Prolog with Features Inheritance, and Templates.

ProFIT works as a pre-processor, which takes a file containing a ProFIT program as input, and gives a file with a Prolog program as output. Sorted feature terms and finite domains are compiled into a Prolog term representation, and the usual Prolog term unification is used at runtime, so that there is no slowdown through a unification algorithm, and no meta-interpreter is needed.

ProFIT uses the same techniques for compiling sorted feature terms and finite domains into Prolog terms as the Core Language Engine of SRI Cambridge and the Advanced Linguistic Engineering Platform (ALEP 2.2) by the European Community, BIM, and Cray Systems.

ProFIT is not a grammar formalism, although it is motivated by NLP and provides some ingredients that are considered typical of grammar formalisms. The goal of ProFIT is to provide the datatypes listed above without enforcing any particular theory of grammar, parsing or generation. ProFIT can be used to extend your favourite Prolog-based grammar formalism, parser and generator with the expressive power of sorted feature terms.

In order to develop, test and use ProFIT programs, a number of utilities are provided:

- The solutions of a query are displayed as ProFIT terms, not in the (hard to read) Prolog term representation
- A pretty-printer for feature terms is provided
- The output of the pretty-printer can be configured by the user
- Cyclic terms can be printed out
- Queries with ProFIT terms are accepted

- ProFIT terms can be output as LaTeX (sorted feature terms are printed as attribute-value matrices)

ProFIT has been implemented in Sicstus Prolog (2.1 #9) and is available free of charge by anonymous ftp in the directory:
`ftp://ftp.coli.uni-sb.de/pub/profit`
The documentation is available as a text file:
`ftp://coli.uni-sb.de/~erbach/formal/profit/profit-doc.txt`
or can be browsed through the web:
`http://coli.uni-sb.de/~erbach/formal/profit/profit_doc.html`

3.4.9 TDL

The development of TDL in the years 1993-1995 partly continued the research done previously and described in section 2.4.5. "Intelligent backtracking" via preference-driven (weighted) unification, already tested before 1993, has been fully implemented (cf. [Weyers, 1996]).

The denotation of recursive/cyclic types has been formalised by means of *greatest fixpoint semantics* (GFP, cf. [Krieger, 1995]). A type expansion algorithm approximating the construction of the greatest fixpoint was implemented in 1994. In connection with this problem, it has been proved that nonmonotonically defined types can be transformed into definite equivalences. A transformation algorithm is given in ([H. U. Krieger, 1995]).

Further new development involved experiments with *distributed parsing*, cf. [Abdel Kader Diagne, 1995]. The idea is to allow multiple parsers to process a grammar in parallel, each one accessing a (user-specified) part of the information encoded in the lexicon and type system. For instance, a semantic parser and a syntactic one can work in this fashion. Another application is a distribution of processing between a top-down and a bottom-up parser. In the experiments, a 'semantic' (SEM) and a 'syntactic' (SYN) parser were tested. The parts of linguistic representation serving as the input for the parsers were each compiled to a subgrammar.

The interaction between the parsers is based on the exchange of hypotheses concerning the chart. According to the typical flow of information, the SYN module sends bottom-up hypotheses, and SEM top-down ones. The latter hypotheses can be thought of as the verification of the former, and thus the SEM parser is controlled by the SYN module. With this restriction, two parsing modes are possible:

- non-autonomous parsing, in which the SEM parser works as an additional filter to the syntactic rules used,
- quasi-autonomous parsing, in which the semantic parser extends the chart on its own if no syntactic hypotheses are present.

3.4.10 XLE

The Xerox Linguistics Environment (XLE) is a joint project between the Natural Language Theory and Technology group at Xerox PARC and the Multi-Lingual Theory and Technology group at the Grenoble Rank Xerox Research Centre. Begun in late 1993, it is a logical follow-on to the Medley Grammar Writer's Workbench. It is intended to be an efficient implementation of LFG with the following features:

- implements the full LFG notation (currently everything except functional uncertainty and set distribution)
- provides a point-and-click grammar development environment that allows you see trees and feature structures and rapidly determine what caused a parse to fail
- uses a new parsing algorithm based on contexted unification to handle disjunctions efficiently

- is integrated with an industrial-strength two-level morphology
- is written in C and Tcl rather than Medley Lisp for efficiency and portability

The XLE system has been in use by the PARGRAM project for developing grammars in English, French and German since January 1996. The grammar writers are very happy with the system, and report that it has dramatically improved their productivity in developing grammars.

3.4.11 Overview

Since 1993 there has been some progress in the development of ALEP, CUF, ALE and TDL. The changes in ALEP and CUF primarily concern the implementation, and to a much lesser extent their functionality. The user interfaces have been redesigned, the robustness increased and new debugging facilities have been integrated into the system (CUF). The ALEP generation module, which caused some problems in 1993, has been improved.

New developments within ALE (version 2.0) include constraints on types and a new morphology component developed in Edinburgh. There has also been an attempt to use an extended version of the formalism in the area of machine translation (SNALE i.e. Sharp New ALE, developed at Sharp).

In TDL, some essential work has been done on the formal foundations of the type system and constraint solver (cf. [Krieger, 1995], [H. U. Krieger, 1995]). The research into preference driven unification, initiated earlier, has been continued. As far as new functionalities are concerned, distributed analysis with multiple parsers has been made possible (cf. [Abdel Kader Diagne, 1995]).

As for new formalisms, only two of them have appeared since 1993, ProFIT and CL-ONE. The former is a PROLOG-based feature formalism with sort system and multiple inheritance implemented in the form of the so-called multidimensional sorts. The compilation of feature structures into PROLOG terms allows for the efficient internal PROLOG unification to be used. The system proved much faster than ALE, cf. [Erbach, forthcoming].

CL-ONE has been used as an extension to ProFIT, yet it can be integrated with other feature formalisms (for example, an ALEP extension was initially planned) thanks to its modular architecture. It provides the user with a number of formal tools often needed in theoretical linguistic work, but absent in all previous formalisms. These new tools are set operations (union, difference, subset) and linear precedence constraints. The so-called *guarded constraints* (cf. [G. Erbach and Thiersch, 1995]), corresponding to delayed goals in CUF, are also implemented.

3.5 Development Platforms

3.5.1 ALEP

The first version of the ALEP system, referred to as ALEP-1, was released in mid-1993 to a few pilot sites, for early assessment and feedback. The results of this assessment have been taken into account in the second development cycle.

The basic objectives of the second development cycle undertaken by BIM were:

- development of multi-user capabilities and further lingware management facilities;
- porting of the ALEP application software on to other widespread hardware platforms;
- development of a more sophisticated text-handling component;
- implementation of a more intuitive and user-friendly man-machine interface.

We describe the ALEP platform as it appeared when version 2.3 was released. The system can be characterised by the following key aspects:

- **general purpose**: Suitable for development of a range of applications, both mono- and multi-lingual. The system provides a framework for access to linguistic processing and text handling tools and applications.
- **open, customisable, configurable**: The system's modular architecture allows for the extension of the tool box and for the replacement of individual tools. Tools are themselves easily configured in functionality.
- **based on 'standards'**: The system builds on a number of standards (SGML, ISO character sets and Prolog draft, OSF/MOTIF).
- **standard formalism**: The system comes with an efficient (conservative) linguistic formalism which is itself extensible. This formalism is a conservative 'lean' formalism, which is intended to be 'neutral' w.r.t. different linguistic schools, for example HPSG, LFG.
- **formalism independent (environment)**: While the system comes with a standard formalism, the platform itself is formalism independent and can be used with other linguistic formalisms.
- **graphical interface**: The system has a graphical customisable interface based on MOTIF and Emacs.
- **multi-user**: Linguistic and environment resources can be shared in public or private databases.
- **support re-use and exchange**: The clean 'neutral' declarative formalism allows linguistic knowledge to be imported and exported. ALEP also has import/export tools for packaging lingware. Individual applications and components of the system can also be used outside of ALEP.
- **lingware resources**: ALEP comes with demonstration lingware (German analysis, transfer and generation in English) and ongoing projects are developing a methodology for producing large-scale grammars and resources in each of the nine community languages.

A number of tools are available:

- **Editing tools**: for editing declarations, PS rules, transfer rules, TLM rules, TH declarations, abbreviation tables, S&R rules etc.
- **Debugger**: a tool for analysing traces of linguistic processing

- **Graphical feature structure viewer**: a tool for viewing the large structures commonly produced by linguistic processing (zoom, hide, filter, co-indexing etc.).
- **Processing tools**: tools for linguistic processing and text handling:
  - analysis/generation of texts (TH)
  - linguistic. operations (analysis, transfer, synthesis etc)
  - combined TH/Linguistic operations (text to text)
  - saving of operation events (for use with the debugger)
- **Object editor**: browsing and editing object descriptions, maintaining object databases that described the resources (data, lingware) used in the system.
- **User language tool**: for writing, compiling and executing tasks.
- **Console tool**: a command line query interface to the ALEP user language.
- **Task theory tool**: a tool for editing, compiling and maintaining ALEP user language descriptions.
- **On-line help tool**:

The Hardware Prerequisites are the following:

- Sparc with operating system Solaris 2.X or Sparc running SunOs 4.1.x.
- Swap: At least 70 Mb should be reserved for ALEP.
- Memory: Recommended 32 Mb
- Source and compiled software: ALEP-2 required approximately 100 Mb.
- Around 56 Mb were to be reserved for ProLog by BIM, ClauseDB and Emacs.

The software Prerequisites are the following:

- ProLog by BIM 4.0.10
- ClauseDB 2.0.12
- GNU Emacs 19.x
- OSF/Motif 1.2.x

Recently ALEP-3 has been made available. There is little change in ALEP-3 fucntionality w.r.t. ALEP-2.3. The principle change is that the ALEP-3 software is built with Quintus Prolog whereas the ALEP-2.3 software is built with Prolog by BIM.

As for Performance, Runtime behaviour of ALEP-3.1 is almost the same as of ALEP-2.3. However, in some circumstances (large grammars, large lexicon) performance of linguistic operations may degrade.

Contact Point

Mr Paul Meylemans
Commission of the European Communities
DG XIII/E/4 Linguistic Research and Engineering
JMO Building, B4/120A L-2920 Luxembourg
Tel: +352 4301 32711/32886 Fax: +352 4301 34999
email: paul_meylemans@eurokom.ie

Partners

Dr David Sedlock BIM sa/nv
Leuvensesteenweg, 510 B-1930 Zaventem
Tel: +32 2 719 26 11 Fax: +32 2 725 47 83
email: david@sunbim.be

Dr Neil Simpkins Cray Systems
11b, Boulevard Joseph II
L-1840 Luxembourg
Tel: +352 25 08 90/91 Fax: +352 25 08 92
email: neil@cray_systems.lu

3.5.2 Hdrug

Hdrug is an environment to develop logic grammars / parsers / generators for natural languages. The package is written in Sicstus Prolog and uses the ProTcl as an interface between Sicstus Prolog and Tcl/Tk. Tcl/Tk [Ousterhout, 1994] is a powerful script language to develop applications for the X-windows environment.

The package comes with a number of example grammars, including a Categorial Grammar, a Tree Adjoining Grammar, a Unification Grammar in the spirit of Head-driven Phrase Structure Grammar, an Extraposition Grammar and a Definite Clause Grammar.

Each of the grammars comes with a set of parsers, such as Earley-like chart parsers, left-corner parsers and head-driven parsers. Some grammars come with variants of the head-driven generator.

Hdrug allows for easy comparison of different parsers/generators; it has extensive possibilities to compile feature equations into Prolog terms; it can produce graphical (Tk or LaTeX), and ordinary Prolog output of trees, feature structures, Prolog terms (and combinations thereof), plotted graphs of statistical information, and tables of statistical information.

Using simply menus and buttons it is possible to parse sentences, generate sentences from logical form representations, view the parse trees or feature-structures that are derived by the parser or generator, change a particular version of the parser on the fly, compare the results of parsing the same sentence(s) with a set of different parsers, etc.

It should not be difficult to adapt an existing NLP system written in Sicstus to Hdrug. As an example the Ale system is ported to Hdrug (see below).

Applications that come with the Hdrug distribution

Currently the following example applications are shipped with the Hdrug distribution.

Ale: Ale 2.0 HPSG grammar, written by Bob Carpenter and Gerald Penn. As of August 1995 the Ale 2.0 HPSG grammar is now fully integrated in the Hdrug system and provides one of the key examples of using Hdrug. Parse trees are browsed in such a way that clicking on a node gives the corresponding feature structure in a separate window. Clicking on an attribute of a feature-structure hides the substructure (a really useful feature for the large feature-structures produced in this application). Clicking with a different mouse-button brings the hidden substructure back on the display.

Apart from parse-trees you can also browse type definitions, predicates, lexical entries, macro's, lexical rules, definite clauses, and grammar rules. Furthermore you can also browse the partial results, as maintained in the chart.

HPSG grammar for Dutch using delayed evaluation techniques to implement recursive lexical rules (directory LexRules), implementing a technique described in [van Noord and Bouma, 1994]. This application also heavily uses the possibilities of the Hdrug system. As a special feature it has the possibility to browse the lexicon as an inheritance hierarchy.

Another interesting feature of this application (and most of the others) is the possibility to browse Prolog definitions in a Tcl/Tk window. Clicking on a predicate produces the definition of that predicate in the window also. If desired the Prolog definitions can be browsed in such a way that all arguments are interpreted as feature structures.

TAG: Small Tree Adjoining Grammar + 9 related head-corner parsing algorithms for headed Lexicalized and Feature-based TAG's (based on my paper on TAGs). Based on techniques presented in [van Noord, 1994]. For this application the tree browser is especially useful. In one of its incarnations this tree browser allows trees in which the nodes are feature-structures to be printed.

DCG: DCG for Dutch, originally used as illustration for semantic-head-driven generation ([Shieber *et al.*, 1990; Shieber *et al.*, 1991; van Noord, 1990]. Furthermore, some of the parsers were used for the timings of the paper co-authored with G. Bouma on the potential efficiency of head-driven parsing [Bouma and van Noord, 1993]. Plotted graphs in which the parsers are compared with respect to milliseconds of cputime for each of the available sentence lengths can be built automatically. Alternatively the same kind of information can be formatted automatically as a LaTeXtable.

Constraint-based CG: Constraint-based Categorial Grammar for English written by G.Bouma, slightly adopted by G. van Noord for inclusion in Hdrug. [Bouma and van Noord, 1994a].

Small DCG: The smallest possible DCG. Illustration of what you need to do minimally to adapt a grammar / parser to Hdrug.

Extraposition Grammar: Extraposition grammar based on the paper by Fernando Pereira [1981].

Comparison with Pleuk

This package might be compared with Pleuk (2.5.4, 3.5.4), and it provides for a large part overlapping functionality.

An improvement over Pleuk is the use of ProTcl (instead of GM). For this reason the GUI can be said to be more flexible, nicer and more robust. For example, in Hdrug you can still use the ordinary Sicstus prompt, while at the same time having the option to use your mouse to press buttons, and menu-items, etc.

Pleuk contains some functionality that is not available in Hdrug; most notably a nice graphical derivation checker. Hdrug contains some functionality that is not available in Pleuk; most notably extensive possibilities to compare the speed and memory requirements of different parsers.

Current Availability

The Hdrug package is available through anonymous ftp in directory:

`ftp://ftp.let.rug.nl/pub/prolog-app/Hdrug/`

The newest version can be found as the gzipped tar-file hdrugSUF.tar.gz where SUF is the version number. Alternatively the same file is accessible through World Wide Web:

`http://www.let.rug.nl/ vannoord/Hdrug/`

Hdrug comes with extensive documentation. This documentation can also be browsed through the web:

`http://www.let.rug.nl/ vannoord/Hdrug/hman/hman.html` The Hdrug program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.

In order to use Hdrug you need SICStus Prolog version 2.1 #8 or #9. A port to SICStus 3.1 is foreseen (using the Tcl/Tk interface that comes as a standard library in Sicstus 3.1). Furthermore, you need ProTcl 1.4, including the standard Tcl/Tk distribution with the BLT and TkTree extensions.

Current Status and Future Plans

Hdrug is still under development. The author is maintaining it and new releases are uploaded to the ftp site on a regular basis. Given that the system is now used by the Dutch NWO Priority Programme on Language and Speech Technology [Boves *et al.*, 1990], it can be expected that this situation will continue for at least a few years.

During the last months Hdrug has undergone only minor and cosmetic changes. The most important change that is foreseen for the coming year is a port to Sicstus 3.1. This should hopefully improve the portability of the system that is currently somewhat problematic (many people report problems installing ProTcl).

3.5.3 PAGE: Platform for Advanced Grammar Engineering

Background and Functionality

The PAGE system developed and maintained in the computational linguistics department of the German National Research Center on Artificial Intelligence (DFKI GmbH) is an advanced NLP core engine that especially facilitates the development of grammatical resources building on typed feature logics (e.g. for HPSG-style frameworks).

PAGE comprises a fair number of various (and often independent) modules and linguistic resources that allow for a flexible configuration according to different user requirements; for grammar engineering purposes (at least) the following set of modules is typically incorporated:

- a unifier providing full boolean feature logic including full and distributed disjunction, negation, and path inequations;
- a powerful type system implementing full boolean type logic including disjunctive and recursive types, open and closed world reasoning, and highly parameterizable type expansion;
- a very efficient bidirectional chart parser allowing for incremental processing as well as for various flexibly parameterized processing regimes;
- an interactive graphical feature structure browser and editor facilitating the navigation (zooming, implosion and hiding of parts of a structure, coreference expansion, and search facilities) in large feature structures;
- a mouse-sensitive chart display allowing the tracing of the parser and the inspection of intermediate results; and
- an ASCII-based command shell.

Optionally, PAGE has been seamlessly integrated with a systematic and comprehensive test suite for English, French, and German (some carefully chosen 5000 test items per language together with rich and highly structured linguistic annotations) developed in the European *Test Suites for Natural Language Processing* (TSNLP) project. The test suites cover about a dozen core syntactic phenomena (using maximally small vocabulary) and are designed to be customized and extended by PAGE users.

PAGE has been successfully deployed (inside and outside of DFKI) for both grammar development and application building; it has proven a sufficiently mature and robust platform for large-scale engineering (e.g. in the DFKI DISCO and PARADICE projects and part of the VerbMobil initiative) as well as for experimentation and rapid prototyping (e.g. in teaching and developing toy grammars).

Availability

The PAGE system is implemented in Common-Lisp and ANSI C and builds on standardized graphical tookits (the X Window System and Motif) in order to achieve a great degree of portability. In general, it should be possible to port the system to most Unix workstation for which a suitable

Common-Lisp implementation, C compiler, and X and Motif are available; the following combinations of hardware platforms and Common-Lisp implementations (and versions) are known to work and are more or less actively maintained:

	Allegro CL	Lucid CL	Macintosh CL
SparcStation; SunOS 4.1.x	4.1 and 4.2	4.1	–
SparcStation; Solaris 2.3 or 2.4	4.2	–	–
HP Prism Architecture; HP-UX 9.x and 10.x	4.2	4.0	–
SGI Indigo; Irix 5.2	4.2	–	–
IBM RS6000; AIX 4.1.3	–	4.0	–
Apple Macintosh; MacOS 7.x	–	–	2.0

As system development at DFKI primarily focusses on Sun workstations and Allegro CL, these are most likely to allow for an easy out-of-the-box installation. For Unix environments the use of GNU Emacs 19.x is recommended as a frontend to the Common-Lisp systems; Allegro CL comes with a proprietary, fully integrated Emacs interface while Lucid CL (at least as of version 4.1) is best embedded into Emacs by virtue of the `ilisp` package available from the public domain.

The PAGE resource consumption and performance heavily depends on, among others, grammar and lexicon size. For serious grammar engineering purposes a powerful cpu (> 60 SPECint), at least 50 MByte of disk space, and a minimum of 64 MByte of main memory are required. Processing times (e.g. using the DFKI grammar for German or the VerbMobil HPSG of English) on a SparcStation 5 range from a few seconds to around one minute for average length sentences (10 words).

Documentation and Support

As of January 1996, limited documentation is available on the following topics:

- **type system:** overview, theoretical background and complete user-level documentation;
- **feature editor:** complete user and interface documentation;
- **unifier:** reference documentation on functional interface and feature structure description language;
- **parser:** reference documentation on parameterization and tuning;
- **installation:** basic instructions for initial setup and startup.

It is foreseen to complete the set of user-level documentation and complement it with a tutorial on grammar development and running the system.

Technical support is available from the DFKI computational linguistics department with the restrictions that naturally apply to non-commercial academic software. Because of the lack of complete documentation, instructions and assistance on installing, operating, and porting the PAGE system are provided. Additionally, DFKI aims to supply bug fixes, integrate future developments (see below), and — with some limitations — assistance in adapting the software to future operating system or Common-Lisp versions as well as in porting it to additional platforms.

PAGE User Group and Mailing List

As the number of PAGE installations outside of DFKI increases (currently there are close to half a dozen active user sites), the mailing list:

`page-users@cl.dfki.uni-sb.de`

maintained at DFKI is expected to facilitate the communication and exchange of information between the PAGE development team and the user group. Ideally, the mailing list can serve as a forum for both technical and content-wise discussion and allow PAGE users to receive information and assistance from other user sites as well as from the developers.

For subscription to the PAGE mailing list send email to:
`page-users-request@cl.dfki.uni-sb.de`

(as `page-users` is an unmoderated forum, please make sure to send administrative requests — subscription and unsubcription — to the *request* address rather than to the entire list). It is foreseen to archive messages posted to the PAGE users list at DFKI and make them available through the World-Wide Web.

Licensing

The PAGE system is made available upon request, free of royalties, to interested parties; DFKI reserves the software copyright and intellectual property rights and requires recipients of the PAGE system to sign a non-commercial license agreement. PAGE users agree not to make the software available to third parties without prior consultation with DFKI.

Although the standard license agreement does not foresee the commercial use of the PAGE system, there is no principled obstacle to deploying PAGE for commercial grammar development. However, prior consultation with DFKI and an extended license agreement will be necessary.

Future Development

System development and maintenance continue in the DFKI computational linguistics laboratory where PAGE is actively used in several application-oriented research projects and integrated with various demonstrators and prototype applications.

Future developments of the PAGE core engine are expected to address the following issues:

- **User Interfaces** The current ASCII-based command shell plus the graphical feature editor and chart display already provide the grammar writer with a flexible and powerful development and debugging environment. However, the PAGE user interface yet lacks robustness, built-in help and a homogenious look and feel for the graphical components
- **Processing Efficiency** Though the overall system performance is sufficient for comfortable grammar development, the PAGE processing efficiency needs substantial improvement for time-critical or even interactive applications. Performance modelling building on compilation, learning, and control strategy approaches are all pursued in the DFKI computational linguistics laboratory and shall become available in integration with PAGE in the near future.
- **Documentation** Besides completing the user-level and reference documentation on the individual components in PAGE, more detailed instructions on installing and operating the system and a tutorial on grammar engineering in PAGE are required.

3.5.4 Pleuk

Formalisms supported

The period 1993–1995 saw the porting of several important grammatical formalisms to Pleuk. These were:

Ale The Attribute Logic Engine by Bob Carpenter and Gerald Penn of CMU (version 2.0.1)

vNTag Gertjan van Noord's TAG system

DCG Definite clause grammars

CUF Dorna et al's Comprehensive Unification Formalism.

Concurrent work

At the same time as the above formalisms were being ported, minor maintenance work was done of the system as a whole. In addition, as part of ESPRIT BRA 5665 DANDELION, the system was used as the basis for a "Discourse Analyst's Workbench", permitting example texts to be stored and annotated, and their interaction with theories of discourse to be studied.

Current availability

Pleuk is available via anonymous FTP from:
`ftp://ftp.cogsci.ed.ac.uk/pub/pleuk`

containing version 1.1beta, and a more recent revision 1.2a. There are no licensing restrictions for non-commercial use. The system is known to run on Sun workstations, under Sun OS 4.1 and later versions, as well as Sun OS 5.2 and later versions. It also runs on Hewlett Packard 9000s, under HPUX.

A version of SICStus Prolog between 2.1.6 and 2.1.9 is required. Plans are being made for a port to later versions of SICStus Prolog.

3.5.5 TAG

The XTAG Workbench

XTAG is an on-going project to develop a wide-coverage grammar for English, based on the Feature-Based Lexicalized Tree Adjoining Grammar (FB-LTAG) formalism.

XTAG also serves as grammar development system consisting of

- a predictive left-to-right parser,
- an X11 based interface to all components, including the graphical display and editing of trees and feature structures,
- a morphological analyzer,
- morphological and syntactic databases,
- a part-of-speech tagger based on trigrams and
- a supertagger.

The software is written in Common Lisp and C. XTAG and the morphological components are available via ftp. Instructions and more information can be obtained by mailing requests to `xtag-request@linc.cis.upenn.edu`. More information can also be found on the World-Wide Web at `http://www.cis.upenn.edu/˜cliff-group/94/xtag.html`.

VM-GECCO

VM-GECO encompasses the GEneration COmponents of the VerbMobil project. VM-GECO provides a Tcl/Tk based graphical interface, including the graphical display (but not editing) of trees and feature structures.

TAG-GEN is the predecessor of VM-GEN. The current version of VM-GEN as part of VM-GECO (see [Kilger and Finkler, 1995]) has switched to the monotonic feature structures of FB-TAG, facilitating the exchange of grammars between the systems. Information about VM-GEN can be found at `http://www.dfki.uni-sb.de/verbmobil/tp/tp9/`

TAG-GEN is a syntactic generator that exploits an incremental and parallel processing scheme. By handling incremental input and producing incremental output, efficiency and flexibility are improved. The goal is to build a system that both runs in real-time and produces output that is highly adaptive to expansions and changes in the input. Incrementality is supported by a distributed, parallel model of active cooperating objects. They verbalize the incrementally given input in a lexically guided two-level system, first building the hierarchical structure and then computing the serial order of words in the sentence under construction. Tree Adjoining Grammars are used as the syntactic representation formalism and have demonstrated their adequacy in supporting incremental processing. Information about TAG-GEN can be found on the World-Wide Web at `http://cl-www.dfki.uni-sb.de/cl/registry/generation/tag-gen.html`

3.6 Implemented Grammars

3.6.1 Introduction

Providing a representative survey of implemented grammatical resources raises two non-trivial issues, viz. (i) which grammars to include and (ii) how to quantify and compare size grammatical coverage.

Firstly, in the selection of implemented grammars presented presently we follow the focus of chapter 7 (see there for the motivation) on the frameworks of HPSG, LFG, and TAG. Additionally, the following two criteria were applied:

- significant coverage (though difficult to quantify) and
- availability in an actively maintained formalism.

Secondly, in aiming for comparability in the presentation of grammatical resources a template-style questionaire was compiled from (i) a classification of syntactic phenomena (see below) with coarse-grained four value coverage judgements and (ii) a set of size measures (e.g. the numbers of lexical entries, phrase structures rules, type definitions et al.) where applicable. Naturally, the numbers given are often argueable measures (especially the numbers of types and macros) and not all of them necessarily apply to all of the frameworks (or, within one framework, to all formalism implementations). However, if nothing else, they can at least indicate a tendency of how many statements are required for a certain degree of coverage; besides, it often is interesting to compare the ratio of types vs. macros for formalisms that provide both.

Following is the list of *core syntactic phenomena* compiled by the LRE project TSNLP (Test Suites for Natural Language Processing) that was used as a template in judging the coverage of the implemented grammars studied in this section.

- complementation;
- agreement;
- modification;
- diathesis;
- modality, tense, and aspect;
- sentence and clause types;
- topology and word order;
- coordination;
- negation; and
- extragrammatical (e.g. parentheticals and temporal expressions).

Since many of these phenomena manifest themselves in different contexts, a further sub-classification of phenomena relates to the relevant *syntactic domains* in which a phenomenon occurs; for the above list these are sentences or clauses (C, noun phrases (NP), adjectival phrases (AP), prepositional phrases (PP), and adverbial phrases (AdvP). Cross-classified phenomena names are composed by attaching the syntactic domain as a prefix to the phenomenon name: for example *C_Agreement* refers to clausal (or subject verb) agreement and *NP_Agreement* to NP-internal agreement respectively. *Diathesis* and *Sentence and Clause Types*, on the other hand, naturally are restricted to the sentential domain (see [Oepen *et al.*, 1996] for details).

For most of the grammars, the template was submitted to (one of) the responsible implementors of the grammar in question along with some guidelines on coverage measurement. All of the coverage judgements and numbers given were sampled between December 1995 and January 1996; several of the grammars presented are still under active development and, presumably, have evolved since.

3.6.2 Implemented LFG Grammars

The LFG PARGRAM Project

The LFG PARGRAM project is a collaborative effort involving researchers from Xerox PARC in Palo Alto, the Rank Xerox Research Centre in Grenoble, France, and the University of Stuttgart in Stuttgart, Germany. The aim of the project is to produce large grammars for English, French, and German which are written collaboratively, based on a common set of linguistic principles, with a commonly-agreed-upon set of grammatical features. The English grammar is being written at Xerox PARC by Tracy King, María-Eugenia Niño, and Mary Dalrymple; the French grammar is being written at the Rank Xerox Research Center by Frédérique Segond and Caroline Brun; and the German grammar is being written at the University of Stuttgart by Christian Rohrer, Miriam Butt, and Christian Fortmann.

The PARGRAM project is currently conducted within two different grammar development environments. At the start of the project, all three sites used the Xerox LFG Grammar Writer's Workbench, a grammar development environment written in Medley Lisp, a variant of Interlisp ([Kaplan and Maxwell, 1993]). The Grammar Writer's Workbench is a complete parsing implementation of the LFG syntactic formalism, including various features integrated since the introduction of LFG theory by [Kaplan and Bresnan, 1982] such as functional uncertainty, functional precedence, generalization for coordination, and multiple projections. It includes a very rich c-structure rule notation, plus various kinds of abbreviatory devices (including parameterized templates, macros). More recently, the Xerox sites have moved to a newer platform, the Xerox Linguistic Environment, which is similar in spirit to the Grammar Writer's Workbench but operates within Unix and Tcl/Tk instead of the Medley programming environment. The core of the project is an efficient unification-based parser based on "contexted unification" ([Maxwell III and Kaplan, 1991]). The Xerox Linguistic Environment is still under development; currently (March 1996) it implements all of the features that the Grammar Writer's Workbench does except functional uncertainty and generalization for coordination. A unification-based generator is also under development.

The grammars consist of phrase-structure rules and abbreviatory rule macros; LFG allows the right-hand side of phrase structure rules to consist of regular expressions (including the Kleene Star notation) and arbitrary Boolean combinations of regular predicates (including immediate dominance and linear precedence). Thus the rules in the grammar actually abbreviate a large set of rules that might be written in a more conventional notation ([Kaplan and Maxwell, 1993]). The lexicons used by the sites consist of entries for stems, template definitions, and lexical rules. The Xerox Linguistic Environment allows for an interface to an external finite-state morphological analyzer, and so the sites using that development environment supplement their lexicons with entries for the information about morphological inflection supplied by the analyzer ([Karttunen, 1994]). Currently the Medley Grammar Writer's Workbench does not allow for the use of an external morphological analyzer; instead, suffix stripping rules are introduced, and so inflectional endings for words in languages like English can be recognized. Sites using that development environment use morphological tables to encode these suffix endings and the information they carry.

A broad range of core constructions is covered by the English, French and German grammars. All grammars cover declaratives and imperatives, including negation. The French and German grammars cover second-person imperatives as well as the imperative infinitive construction. The German grammar covers standard interrogatives. The English grammar covers standard interrogatives as well as matrix and embedded interrogatives, and its coverage of declarative constructions includes existentials and negative inversions.

All three grammars cover a range of nominal modification: determiners, adjectives, noun compounding, relative clauses, and prepositional phrase nominal modifiers. The English and German grammars also cover measure phrases and modification of nouns by verb phrases and deverbal adjectives. The complicated interaction between case and nominal agreement (adjective and determiner inflection), as well as the occurrence of headless NPs (e.g., "der blaue") is accounted for in the German grammar as well. All grammars cover various coordinate constructions involving all phrases and lexical heads, as well as specialized number agreement with NP coordination.

The three grammars include parallel templates for a wide range of verbal subcategorization and diatheses, including passive, dative shift, particle verbs, various types of control, predicative constructions, subcategorized prepositional phrases, and auxiliary sequencing and selection. The German grammar also allows for "quirky" case constructions (e.g., "helfen"- 'to help') and their interaction with passivization.

All three grammars cover a range of verbal modification, including adverbs and adverb placement, purpose clauses, and subordinate conjunctions. The French and German grammars provide various kind of tense resolution (composed tenses), reflexive constructions, and clitic pronouns (French) or expletives (German). The English grammar also covers various specialized constructions (e.g. parentheticals and time expressions).

Finally, the German grammar covers the various word order and topicalization possibilities found in German. In particular, the distribution of VP-topicalization is fully implemented. Scrambling possibilities are modeled in terms of the traditional Vorfeld, Mittelfeld, Nachfeld partition while simultaneously (through the use of macros) producing constituent structures which are consistent with X'-theoretic views of phrase structure.

LFG PARGRAM English Grammar

LFG PARGRAM English Grammar	
Phenomenon	**Coverage**
C_Complementation	○
C_Agreement	+
C_Modification	+/○
C_Word-Order	○
C_Coordination	○
NP_Complementation	+/○
NP_Agreement	+
NP_Modification	+
NP_Word-Order	○
NP_Coordination	○
AP_Complementation	+
PP_Complementation	+
Diathesis	+
Tense Aspect Modality	+
Sentence and Clause Types	○
Negation[1]	+/○
+ full; ○ reasonable; ∼ rudimentary; − no	

# of lexicon entries	925
# of phrase structure rules	56 + 11 macros[2]
# of lexical rules	1
# of type definitions	0
# of template (macro) definitions	85 + 21 morph.

[1] Double negation is not handled.

[2] Phrase structure rules in LFG have regular expressions on their right hand side. Converting to Chomsky Normal form would result in a grammar with more than 4000 rules.

LFG PARGRAM French Grammar

LFG PARGRAM French Grammar	
Phenomenon	**Coverage**
C_Complementation	∘
C_Agreement	+
C_Modification	∘/+
C_Word-Order	∘
C_Coordination	∘
NP_Complementation	∘/+
NP_Agreement	+
NP_Modification	+
NP_Word-Order	∘
NP_Coordination	∘
AP_Complementation	+
PP_Complementation	+
Diathesis	∘
Tense Aspect Modality	+
Sentence and Clause Types	∘
Negation	∘/+
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	949
# of phrase structure rules	25
# of lexical rules	1
# of type definitions	0
# of template (macro) definitions	20 + 2 morph.

LFG PARGRAM German Grammar

LFG PARGRAM German Grammar	
Phenomenon	**Coverage**
C_Complementation	+
C_Agreement	+
C_Modification	∘
C_Word-Order	+
C_Coordination	∘
NP_Complementation	∘
NP_Agreement	+
NP_Modification	+
NP_Word-Order	+
NP_Coordination	∘
AP_Complementation	+
PP_Complementation	+
Diathesis	+
Tense Aspect Modality	+
Sentence and Clause Types	+
Negation	∘
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	1334
# of phrase structure rules	58 + 14 macros
# of lexical rules	2
# of type definitions	0
# of template (macro) definitions	114

University of Sevilla LFG

Our work on LFG has been guided by a primary interest in Machine Translation, and how LFG can be used as the linguistic framework upon which an MT system can be implemented. Following that idea we propose to merge the classical transfer-based MT architecture with LFG's two-level representation. Thus, the MT system firt produces c- and f-structures as the result of parsing, then transfer operates from source f-structure to target f-structure and, finally, generation operates from target f-structure into target c-structure. According to these criteria, not much effort has been devoted to accommodate our prototype to the 'latest' developments in LFG.

The work described above started in 1992, with JULIETTA [Amores, 1992], an LFG-based MT prototype written in DCG-Prolog. Originally, JULIETTA was designed to translate medical abstracts from English into Spanish. In addition, it has been used to test difficult translation problems from English into Spanish. The current linguistic coverage of the system includes full translation of the following phenomena:

- NP agreement, modification and complementation
- VP, S and NP coordination
- Head-shifting phenomena such as resultative and causative constructions.
- Lexical redundancy rules, diathesis and predicate composition
- Reflexive and reciprocal constructions
- Question formation
- Hyphenated compounds

Size of the lexicon: 1000 stems (roughly).

Due to the good results we obtained with JULIETTA we decided to build a C shell for the development of LFG-based MT systems. The main goal was to achieve a high level of efficiency while maintaining the excellent linguistic performance of JULIETTA. During the past two years we have developed a very efficient tool for the implementation of transfer-based MT systems in the spirit of LFG. The overal system consists of the classical MT modules: analysis, transfer and generation [Quesada and Amores, 1996].

The main components of Episteme are:

Its goal is to obtain a logical model which permits highly efficient storage and retrieval of very large knowledge bases that use feature structures as their theoretical backbone [Quesada and Amores, 1995].

¿From a computational point of view, the implementation of Vtree makes use of two techniques: a tetra–dimensional organizatioon of logical records, and binary trees with vertical cut.

The computational complexity of Vtree is of the order *O(log(log(N))*, where N is the number of items in the lexicon.

A specification language designed for rich lexical descriptions based on attribute-value pairs, which may include:

- Macros and Meta Predicates
- Optionality
- Negation

- Co–reference
- List–valued features
- Disjunction over atomic values
- Lexical redundancy and diathesis rules
- Morphological rules
- Lexical Ambiguity (more than one category associated with a lexical unit and more than one feature description associated with a syntactic category)

Vtree and Mph are multi-protocol. That is, they can be linked to other NLP applications written, for example, in Prolog. With artificial tests of lexical databases containing more than one million records, both systems make the lexical analysis and retrieve the feature structure associated with any inflected form in less than 1.5 milliseconds.

Includes the set of modules that manipulate the languages involved in the translation process: parser, unifier, transfer and generator.

The parser of NLEngine is a bidirectional, event–driven and chart–based model that incorporates strong top–down predictions over the general bottom–up strategy. The unification algorithm is based on constructive unification with post–copy. Both systems perform syntactic parsing with interleaving unification. They cover LFG requirements for well-formedness such as completeness, coherence and consistency. Functional uncertainty has not been implemented yet. The specification language is easy to understand to anyone familiar with LFG. The following would be an example of a PS rule in Episteme. Note that the right–hand side of the rule may be augmented with control functions and user–defined functions. Also, Completeness and Coherence may be tested over a subset of grammatical functions or over all grammatical functions previously defined.

```
(22: VP -> V NP)
{
 UP = SELF-1;
 IF (MEMBER(ncomp,SELF-1.ggf)) THEN {
        UP.ncomp = SELF-2
 } ELSE {
        IF ( MEMBER(obj,SELF-1.ggf) )
               THEN {UP.obj = SELF-2 } };
 COMPLETENESS(GF-[subj]);
 COHERENCE(GF-[subj])
}
```

The memory organization model of NLEngine is based on a new technique named *FastMem* that compresses the representation of grammar relations and avoid the slow string comparison operations, improving the overall efficiency of the system.

NLEngine includes also transfer and generation modules, both using the *FastMem* memory model.

The system described above has been tested in a Speech–to–Speech Machine Translation Prototype developed jointly with Telefonica I+D (a subsidiary of the Spanish Telephone Company).

The task [Amores *et al.*, 1994] involved Spanish–to–English translation of a corpus of 600 sentences in the domain of banking transactions. Lekta worked as the MT module within the overal speech–to–speech MT system. The MT system showed a performance of 98% of accurate translations at a speed of 200 words/second.

3.6.3 Implemented TAG Grammars

DFKI TAG Gramamrs

In the TAG-GEN project at the DFKI, an English and a German TAG grammar have been developed. They are currently used mainly in the Verbmobil and Effendi projects. The English grammar now uses a monotonic feature-based TAG formalism while the German grammar still uses the non-monotonic UTAG formalism.

Although grammar development has not been the main focus of the projects, work on the grammars has been continuous over the last five years, leading to a fairly comprehensive grammar. The main contributors are Wolfgang Finkler, Anne Kilger, Peter Poller and Karin Harbusch.

The grammars are developed with respect to an incremental generation system, covering small, specific domains. Therefore, the lexicon remains limited, although the syntactic coverage is fairly wide.

DFKI TAG Grammars		
Phenomenon	**Coverage**	
	GERMAN	ENGLISH
C_Complementation	∘	∘
C_Agreement	∘	∘
C_Modification	∘	∘
C_Word-Order	∘	∘
C_Coordination	∼	∼
NP_Complementation	∼/∘	∼/∘
NP_Agreement	∘	∘
NP_Modification	∼	∼
NP_Word-Order	∘	∘
NP_Coordination	∘	∼
AP_Complementation	∼	∼
PP_Complementation	∘	∘
Diathesis	∘	∘
Tense Aspect Modality	∘	∘
Sentence and Clause Types	∘	∘
Negation	∘	∘
Time&Date Expressions	∘	∘
+ full; ∘ reasonable; ∼ rudimentary; − no		

# of lexicon entries (stems)	1606	2737
# of phrase structure mobiles	185	225
# of lexical rules		
# of type definitions		
# of template (macro) definitions		

Note that the second table (numbers of entries) does not allow an immediate comparison with, for example, the tables for HPSG grammars. There are no lexical rules or type definitions in the TAG grammars, and even though there are template (macro) definitions they are of a very different type. Instead of the number of phrase structure rules, the number of *mobiles* is listed. These are immediate–dominance structures (i.e. trees) which represent more than 10 (word order) variants on average for verbs. For other categories this number is smaller. Finally, the lexical entries are stems, rather than fully inflected forms.

French TAG grammar

The French TAG grammar developed by Anne Abeillé's group at the University of Paris VII is based on the work in [Abeillé, 1991]. The development effort is now continuing for a several years and uses the XTAG system. As all TAG grammars, it is completely lexicalized. It uses only atomic-valued feature structures. See [Abeillé *et al.*, 1994] for a more detailed description.

It currently covers the following phenomena:

Subcategorization of verbs, nouns, adjectives, prepositions, adverbs, conjunctions
Auxiliaries, modals, aspectuals and other "raising" verbs
Clitic pronouns for arguments and some frozen ones
Infinitival and sentential arguments
Passive
Impersonal constructions
Relatives
Interrogatives
Cleft sentences
Unbounded dependencies
Complex subject inversion
Some word order variation
Some light verb constructions
Idiomatic and semi-frozen expressions
Complex determiners and predeterminers
Complex or frozen adverbials
Complex prepositions, frozen conjunctions
Coordination of arguments, S and "VP" coordination

There are over 20 tree families for verbs with an average of 15 trees each, plus 50 single elementary trees. The morphological lexicon contains over 50,000 (inflected) entries, and the syntactic lexicon has over 6,000 entries.

XTAG English TAG Grammar

As part of the XTAG project at the University of Pennsylvania, Philadelphia, a wide-coverage grammar for English has been developed. The development has been going on for about eight years now and the current state of the grammar is described in great detail in [XTAG, 1995].

The morphological database contains about 317,000 inflected entries which have been extracted from the Collins English Dictionary and the Oxford Advanced Learner's Dictionary. The information stored encompasses inflectional features, the stem and part of speech information. The syntactic database has been extracted from the same dictionaries and contains about 37,000 entries (stems) which are mapped to trees and tree families. There are 569 trees which compose 38 tree families plus 67 single trees.

For a complete description of the syntactic coverage of the English XTAG grammar, see [XTAG, 1995]. An attempt to characterize its coverage and enable a comparison with other grammars is given below:

UPENN English XTAG grammar	
Phenomenon	**Coverage**
C_Complementation	+
C_Agreement	+
C_Modification	+
C_Word-Order	+
C_Coordination	∘
NP_Complementation	+
NP_Agreement	+
NP_Modification	+
NP_Word-Order	+
NP_Coordination	∘
AP_Complementation	+
PP_Complementation	+
Diathesis	+
Tense Aspect Modality	+
Sentence and Clause Types	+
Negation	+
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	317,000
# of elementary trees	569
# of lexical rules	
# of type definitions	
# of template (macro) definitions	

Note that the elementary trees extend to more than one level of phrase structure, although unlike the mobiles of the DFKI grammars, they have a fixed structure. The XTAG system also uses templates for sets of feature equations.

There also exist small grammars for Korean, Chinese and Spanish, developed using the XTAG system.

3.6.4 Implemented HPSG Grammars

DFKI German HPSG Grammar

The DFKI German HPSG grammar grew out of the nationally funded DISCO (Dialogue System for Cooperating Agents) project and its successor PARADICE (Parameterizable NL Discourse Core Enginge). Continuous grammar development for over more than five years now has made the DFKI HPSG grammar one of the largest implementations of the HPSG theory of grammar; the basic grammar design, syntax, and interface to morphology were carried out by Klaus Netter, while the integrated semantics components of the grammar were contributed by John Nerbonne and Walter Kasper (see [Busemann and Harbusch, 1993]).

Grammar development at the DFKI is closely coupled with the implementation and maintenance of the DFKI HPSG system, a comprehensive and comfortable grammar development and NLP core platform that is in active use at several international research institutions. Additionally, in the COSMA (Cooperative Schedule Management Agent) project the DFKI HPSG grammar has been successfully deployed in a prototype NLP application in the domain of distributed appointment scheduling by email ([Busemann *et al.*, 1994]) and, at the beginning VerbMobil was delivered to IBM Germany as an initial input for grammar building within VerbMobil.

Presently the DFKI HPSG grammar by and large covers all of the standard construction types of German, including among others the different types of complementation and of adverbial constructions, all major types of sentence mood, a broad range of word order phenomena including different forms of verb cluster formation (involving, for example, modals, temporal auxiliaries, passive constructions etc.), as well as some specialized subgrammars, e.g., for time and date expressions.

The architecture of the grammar comprises run-time executable lexical rules, multi-word lexemes, a typed interface to morphology, and an interface to a speech act recognition component. The grammar consists of approximately 2.000 type definitions, defining a set of about 20 different instances of rule schemata and 1.000 lexical instances, which mainly comprise closed class items (prepositions, determiners, and other functional categories) and representative instances of open class items (nouns, adjectives, and verbs). Through default entries a lexical data base of about 120.000 open class items can be accessed.

Besides integrating a number of linguistic assumptions from both Categorial Grammar (CG) and Lexical Functional Grammar (LFG) into the HPSG framework, the implementation of the DFKI grammar for German has contributed a large number of innovative concepts and methods to the active development of the HPSG theory (e.g. see [Netter, 1992], [Nerbonne *et al.*, 1994] and [Netter, 1996]), the most important of which are (i) the treatment of functional categories as heads and (ii) a revised (basically LFG-style) encoding of subcategorization allowing a simplified treatment of optionality as well as the lexicalization of unbounded dependency constructions, thus dispensing with empty terminals for `SLASH` termination.

Like the English Resource Grammar developed at CSLI (see below), the DFKI HPSG grammar implements a slightly modified top-level feature geometry and strict binary phrase structure.

DFKI HPSG Grammar	
Phenomenon	**Coverage**
C_Complementation	∘
C_Agreement	+
C_Modification	+
C_Word-Order	∘
C_Coordination	−
NP_Complementation	∼
NP_Agreement	+
NP_Modification	+
NP_Word-Order	∘
NP_Coordination	−
AP_Complementation	∘
PP_Complementation	+
Diathesis	+
Tense Aspect Modality	+
Sentence and Clause Types	+
Negation	∘
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries[8]	490
# of phrase structure rules	22
# of lexical rules	4
# of type definitions	1995
# of template (macro) definitions	46

English Resource Grammar (ERG) Initiative

The English Resource Grammar (ERG) Project is a consortium of research groups in several countries all working within the Head-Driven Phrase Structure Grammar framework ([Pollard and

[8] The relatively small number refers to the lexcion entries for closed classes that are part of the grammar proper; through the interface to an external lexical database the actual lexical coverage for the DFKI HPSG grammar is larger by far.

Sag, 1994b]). The goal of the consortium is to enable the exchange of NL software development tools and implementations of English grammar fragments, thereby dramatically enhancing the effectiveness of each group in doing its own focused research.

Currently the project has established the necessary links for this kind of exchange with several institutions, including the German AI Research Institute (DFKI) in Saarbrcken, the Korean Advanced Institute for Science and Technology (KAIST), Simon Fraser University, Osaka University, and Ohio State University. The consortium is also likely to include groups at the Beckman Institute and Carnegie Mellon University. Research and implementation both at CSLI and at these collaborating sites includes work on the morphology, lexicon, syntax, and semantics, along with the necessary processing and analysis tools needed to build a broad-coverage, precise, reusable grammar.

Part of the ERG grammar building at CSLI is carried out within the VerbMobil project and aims to produce an HPSG grammar of English for use in the VerbMobil spoken language translation setup. The English grammar being developed at CSLI for the Verbmobil prototype will be used for generation only; though, in time, it may also be used to help track the ongoing English dialogue to provide some context for the particular expressions that need translation.

While the implementation of the English grammar for Verbmobil will be used only within that project, the grammatical foundations which are application-independent will be reusable both at CSLI and in the consortium, as will the development tools employed on the project. In addition, related grammatical development at other sites is expected to enhance the linguistic coverage of specific applications such as the Verbmobil prototype.

Current ERG work at CSLI is primarily carried out using the HPSG development platform developed at the DFKI ([Uszkoreit *et al.*, 1994]); other ERG members are in the process of obtaining and installing the system. However, because of the specific VerbMobil requirements CSLI and DFKI investigate means of transforming the ERG grammar into (i) another typed feature formalism (viz. the CUF system as one of the VerbMobil standard formalisms) and (ii) another grammatical framework (viz. Tree Adjoining Grammars) for generation. Chapter **??** reviews both of these transformation activities in more detail.

Although CSLI is one of the two sites leading the theoretical development of HPSG, the ERG grammar, nevertheless, is not a one-to-one implementation of the HPSG theory as it is articulated in [Pollard and Sag, 1994b], where some of the diversions are primarily motivated by linguistic considerations while others reflect implementational and technical decisions. Two of the more significant differences between the ERG implementation and the current state of HPSG grammar theory are (i) the choice of the top-level feature geometry and (ii) the adoption of a binary branching phrase structure treatment.

English Resource Grammar	
Phenomenon	**Coverage**
C_Complementation	∘
C_Agreement	+
C_Modification	+
C_Word-Order	∘
C_Coordination	∼
NP_Complementation	∼
NP_Agreement	+
NP_Modification	+
NP_Word-Order	∘
NP_Coordination	∘
AP_Complementation	∼
PP_Complementation	+
Diathesis	∼
Tense Aspect Modality	+
Sentence and Clause Types	∘
Negation	+
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	2600
# of phrase structure rules	18
# of lexical rules	12
# of type definitions	1500
# of template (macro) definitions	13

Large Scale Grammatical Resources (LS-GRAM)

The LS-GRAM (Large-Scale Grammars for EC languages) project aimed for the development of extensible, well-designed, documented and tested lingware for the nine EC languages based on a common mainstream software platform (ALEP). A strong emphasis has been put on the reuse of existing grammatical descriptions where especially experiences and resources originating from the Eurotra machine translation project were migrated into an HPSG framework and ported to the ALEP formalism.

The project took a two-step approach to the execution of the language-specific work: in late 1993 a core consortium of four partners covering three languages (German, English, and Spanish) started providing a certain amount of definition work. This served as input for the other participants who joined the project some 6-7 months later. For German, English, and Spanish, coverage has been determined on the basis of corpus analysis. For the six other languages (Danish, Dutch, French, Greek, Italian, and Portuguese), a more limited scope was envisaged; basically the aim here was to develop core grammars, which may serve as bootstrapping material for future initiatives. This extended part of the project is going to finish in late summer 96.

Since the different grammars have been implemented by different groups, it is not always easy to find a 'common implementational strategy'. Nevertheless, a number of common guidelines can be identified:

- **Formal Restrictions** Two apparently conflicting choices have been made in LS-GRAM in the adoption of the HPSG theory and the use of the ALEP formalism. The conflict arises from the fact that ALEP, as a so-called 'lean ' formalism, does not provide all the formal tools which would be needed for a 'strict' implementation of an HPSG grammar. Among the formal devices which are missing, the following seem to play a fundamental role: multiple inheritance, definition of types as feature structures where embedded attributes can be constrained, set values, separation of ID and LP constraints, implicational and relational constraints, and lexical rules. Under these premises, the principled organization of standard

HPSG is definitively lost. However, it must be said that the basic view of a grammar as driven by a small set of linguistic principles is somehow preserved (at least for the majority of modules analyzed) by the use of macros. Thus, even in a strictly rule based organization, linguistic statements which are shared by more than one rule are encoded by building a (possibly parametrized) macro. Thus, a clear division of linguistic information is kept without introducing additional formal devices.

Whenever the use of macros cannot supply to the limited formal power of ALEP, changes in the structural organization are introduced which try to preserve the original spirit of the HPSG source as much as possible. This is, for instance, the case for the `append()` function used (among others) in the `SUBCAT` principle: since such a function is missing in the ALEP formalism, the effect of the `SUBCAT` principle is emulated through the adoption of a binary branching phrase structure where `append()` is replaced by `cons()`.

- **Efficiency** One of the leading ideas of LS-GRAM was that efficiency of processing should be privileged over theoretical purity. This *desideratum* had, in general, the following consequences over the organization of the various grammars:
  - Besides the purely linguistic specifications, additional, non linguistic, control features are used. Their main goal is either to drive the parsing process directly (e.g. in *head declarations*), or to constrain the grammar in such a way that a more efficient parsing regime can be achieved (direction of branching, blocking of recursion, more compact encoding of the information which is directly relevant for parsing, such as bar level, category et al.).
  - The same HPSG ID principle is usually split into several category-sensitive rules. This is not a move which is strictly enforced by the formalism (for instance, certain grammars, such as the Italian one, still preserve the small HPSG inventory of phrase structure rules), but reasons of efficiency of parsing urge towards this approach.
  - A division of tasks is enforced between *analysis* and *refinement*, where the former is a structure building level, and the latter is a level which works on the analysis output of the analysis, monotonically adding information (i.e. further decorating the analysis structure(s)). Since the two levels share the same formal apparatus, the choice of what pertains to the analysis and what pertains to the refinement is entirely driven by efficiency reasons. In general, a choice has been made to employ the analysis module to handle syntactic information, whereas the main goal of the refinement is to built proper and complete semantic representations. In this way, all the ambiguities which are introduced by multiple meanings of the same words have less consequences for the efficiency of the whole grammar.
- **Homogenity** One of the goals of the project was to achieve a common semantic representation format. Thus, a semantic representation mildly inspired by Situation Semantics was centrally designed, to which all LS-GRAM grammars conform. The main departures from the standard HPSG interpretation of Situation Semantics are that (i) no quantifier storage mechanism is assumed and generalized quantifiers appear directly as arguments in verbal relations; (ii) the roles introduced by predicative relations mirror the ones which were previously used in the Eurotra project; and that (iii) sets are systematically replaced by lists.
- **Concern for Corpus Data** One of the innovations of the project was that grammar development should not be primarily driven by linguistic text books but by rather by an investigation of real texts. This fact has led to a ranking of grammatical phenomena that, in some cases, assigns different priorities to individual phenomena than do textbooks on grammar. As, for instance, unbounded dependency constructions turned out to be relatively infrequent in the corpus under analysis (descriptive newspaper articles from the domain of economy), they will presumably only be included into some of the LS-GRAM grammars towards the end of the project and at best partially treated.

Conversely, phenomena which are sometimes neglected in theoretical investigation, but which have a great frequency rate in real texts (such as compounding) have received a great deal of attention. However, among the phenomena which have significant corpus relevance but, yet, have not received a detailed treatment in all LS-GRAM grammars are coordination and parenthetical constructions.

- **Text-Handling** ALEP provides the user with a Text-Handling component. Basically this component transforms the input text into a SGML marked text. This will be the input of the linguisitc processing. A user-derfined tag is also foreseen and it has been made an extensive use of them in order to treat some of the phenomena evocated just above and all kind of "messy details". Preprocessing thus playing an important role in order to keep the grammar working in a efficient way. Most text handling modules developed within LS-GRAM are now able to deal with numbers, dates, proper names or fixed phrases, which are encoded in the lexicon by the mean of generic entries, thus helping to keep the lexicon small but also allowing to deal with a potentially infinite linguistic material.

  Some of the groups have also used the Text-Handling component in order to access already existing linguistic information. External lexicons, data-bases or results of corpus-anlaysis can be the input of the lingusitic processing: in this way the coverage of the grammars has been considerably extended, while still keeping the number of internal lexicon entries small[9]Efficient Integrated Tagging of Word Constructs, Proceddings of Coling96.

Figures from Some of the LS-GRAM Linguistic Modules

Numbers given for LS-GRAM grammars (phrase structure rules + lexicon) are for the analysis and refinement components respectively. Whenever only one number is found in the relevant cell, this means that either the grammar make no use the refinement module ore the data about refinement were not provided by the relevant sites.

English LS-GRAM	
Phenomenon	**Coverage**
C_Complementation	+
C_Agreement	+
C_Modification	~
C_Word-Order	∘
C_Coordination	~
NP_Complementation	+
NP_Agreement	+
NP_Modification	∘
NP_Word-Order	+
NP_Coordination	−
AP_Complementation	+
PP_Complementation	+
Diathesis	∘
Tense Aspect Modality	~
Sentence and Clause Types	~
Negation	−
+ full; ∘ reasonable; ~ rudimentary; − no	

# of lexicon entries	1231
# of phrase structure rules	19
# of lexical rules	8
# of type definitions	114
# of template (macro) definitions	172

[9]See A. Bredenkamp, T. Declerck, F. Fouvry, B. Music,

German LS-GRAM	
Phenomenon	**Coverage**
C_Complementation	∘
C_Agreement	+
C_Modification	∘
C_Word-Order	∘
C_Coordination	∘
NP_Complementation	∼
NP_Agreement	+
NP_Modification	+
NP_Word-Order	+
NP_Coordination	+
AP_Complementation	∘
PP_Complementation	∘
Diathesis	∘
Tense Aspect Modality	∼
Sentence and Clause Types	∘
Negation	−
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	556|481
# of phrase structure rules	47|47
# of lexical rules	7
# of type definitions	122
# of template (macro) definitions	189

Spanish LS-GRAM	
Phenomenon	**Coverage**
C_Complementation	∘
C_Agreement	+
C_Modification	∘
C_Word-Order	∘
C_Coordination	∘
NP_Complementation	−
NP_Agreement	+
NP_Modification	∘
NP_Word-Order	∘
NP_Coordination	−
AP_Complementation	−
PP_Complementation	−
Diathesis	∘
Tense Aspect Modality	∼
Sentence and Clause Types	∘
Negation	∼
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	762|404
# of phrase structure rules	3611
# of lexical rules	3
# of type definitions	85
# of template (macro) definitions	146

French LS-GRAM	
Phenomenon	**Coverage**
C_Complementation	+
C_Agreement	+
C_Modification	~
C_Word-Order	+
C_Coordination	–
NP_Complementation	+
NP_Agreement	+
NP_Modification	~
NP_Word-Order	+
NP_Coordination	–
AP_Complementation	+
PP_Complementation	+
Diathesis	+
Tense Aspect Modality	~
Sentence and Clause Types	~
Negation	–
+ full; ∘ reasonable; ~ rudimentary; – no	

# of lexicon entries	410|390
# of phrase structure rules	43|43
# of lexical rules	3
# of type definitions	104
# of template (macro) definitions	190

Italian LS-GRAM	
Phenomenon	**Coverage**
C_Complementation	~
C_Agreement	+
C_Modification	∘
C_Word-Order	∘
C_Coordination	–
NP_Complementation	~
NP_Agreement	+
NP_Modification	+
NP_Word-Order	+
NP_Coordination	–
AP_Complementation	∘
PP_Complementation	∘
Diathesis	~
Tense Aspect Modality	∘
Sentence and Clause Types	~
Negation	–
+ full; ∘ reasonable; ~ rudimentary; – no	

# of lexicon entries	640|35
# of phrase structure rules	11|5
# of lexical rules	0
# of type definitions	154
# of template (macro) definitions	142

CMU ALE HPSG Grammar

The development of an HPSG grammar for the ALE system at CMU was originally carried out in order to demonstrate the utility of ALE in describing and processing HPSG grammars. It was, thus, not developed with an eye toward either coverage or robustness.

The architecture of ALE is basically that of a definite clause grammar, with Rounds-Kasper type logical descriptions of feature structures (extended with variables and inequations) replacing first-order terms. Grammars are processed using a bottom-up chart parser. Definite clauses are processed using Prolog's depth-first execution strategy. The type system is basically that of totally well typed feature structures as proposed in [Carpenter, 1992a]. In addition, general constraints on types are allowed. Lexical rules are preprocessed. ALE works by compiling a type signature, grammar, and definite clauses to low-level Prolog clauses.

As part of his MS project, Gerald Penn extended the ALE 1.0 system to ALE 2.0 and extended Carpenter's HPSG grammar from a toy covering the introduction of [Pollard and Sag, 1994b], to one covering all of HPSG 2.0, as represented by Chapters 1–8 of [Pollard and Sag, 1994b], excluding only the binding theory, the quantifier binding condition (quantifier storage and retrieval was implemented), and the raising control principle, which was instead encoded by hand for raising verbs.

As part of a very small development grant from CMU's College of Humanities and Social Science, Kathy Baker extended Penn's grammar to the Chapter 9 feature geometry, and removed traces. She then extended the type hierarchy for phrases to implement the analysis of clause types proposed by Ivan Sag in his (as yet unpublished) paper on relative clauses. The grammar is still not intended to have wide coverage, but rather to demonstrate the ease of representing and processing an HPSG grammar in ALE. Thus, the size of the grammar amounts to only one or two lexical entries of each of the major types, although the rules are completely implemented. Development is another story, since it was only recently, after receiving new equipment, that we were able to implement the Pleuk (2.5.4, 3.5.4) and HDrug (3.5.2) development environments within ALE.

CMU ALE HPSG Grammar[10]	
Phenomenon	**Coverage**
C_Complementation	+
C_Agreement	+
C_Modification	+
C_Word-Order	+
C_Coordination	−
NP_Complementation	+
NP_Agreement	+
NP_Modification	+
NP_Word-Order	+
NP_Coordination	−
AP_Complementation	+
PP_Complementation	+
Diathesis	∘
Tense Aspect Modality	+
Sentence and Clause Types	+
Negation	+
+ full; ∘ reasonable; ∼ rudimentary; − no	

# of lexicon entries	200
# of phrase structure rules	8
# of lexical rules	10
# of type definitions	300
# of template (macro) definitions	50

VerbMobil

The grammar employed in the spoken language translation system Verbmobil is a variant of HPSG, covering basic syntactic and semantic phenomena such as complementation, modification, agreement, case assignment, the verb-second phenomenon pertinent in German (cf. Kiss/Wesche 1991), as well as special phenomena such as the grammar of time and date expressions. The grammar also covers certain phenomena which only occur in spontaneous speech. The parser is designed to handle sentences, all kinds of subsentential phrases, as well as suprasentential units. The semantic part of the grammar features an implementation of HMRS, a new semantic formalism for HPSG.

The grammar has been implemented in CUF. Since CUF employs only a very restricted type system, the grammar may be ported easily into other formalisms which make use of type inheritance. In particular, the implementation makes no use of recursive constraints. The grammar is processed by a lattice chart parser specially tailored for spoken language. In order to separate suprasentential units (turns) into sentences and phrases, the parser makes use of prosodic information which is also employed to determine the position of head traces (cf. Batliner et al. 1996).

The lexicon is based on two huge type hierarchies for syntactic and semantic types and an intricate set of linking schemata which combine syntactic and semantic information. Currently, the lexicon is extended to cover about 2.500 entries, comprising both syntactic and semantic information.

For more information, contact tibor@heidelbg.ibm.com

VerbMobil HPSG	
Phenomenon	**Coverage**
C_Complementation	$\sim$
C_Agreement	+
C_Modification	+
C_Word-Order	$\sim$
C_Coordination	$\sim$
NP_Complementation	$\sim$
NP_Agreement	+
NP_Modification	+
NP_Word-Order	$\sim$
NP_Coordination	$\sim$
AP_Complementation	$\sim$
PP_Complementation	+
Diathesis	$\sim$
Tense Aspect Modality	+
Sentence and Clause Types	$\sim$
Negation	+
+ full; $\circ$ reasonable; $\sim$ rudimentary; $-$ no	

[10] When discussing coverage, we are discussing only the coverage of the grammar, not necessarily the coverage of the lexicon. And given that we implemented an English grammar, issues such as word order are greatly simplified.

# of lexicon entries	2600
# of phrase structure rules	18
# of lexical rules	12
# of type definitions	1500
# of template (macro) definitions	13

Chapter 4

Trends towards Convergence

4.1 Introduction

Throughout the 1980's, there was a lot of innovation in the area of grammar formalisms, and a steady stream of new formalisms, embodying new ideas and concepts, came out of the very fertile "unification underground". DCG, LFG, GPSG, HPSG, TAG, and various new strands of categorial grammar all emerged during this very productive period. Much of the theoretical groundwork was laid in that period through the formalisation of feature logics by Aït-Kaci, Carpenter, Johnson, Kasper, Moshier, and Rounds.

A similar wave of activity took place in the late 1980s and early 1990s in the area of implemented formalisms when the traditional untyped formalisms such as DCG, PATR and the LFG workbench were being replaced by typed feature formalisms such as STUF, TFS, CUF, TDL, ALEP, ALE, and ProFIT.

When EAGLES began, this development was still in full swing, and formalisms appeared to be developing in various directions. By the mid-1990s, however, the situation had become more consolidated. There are still new developments, for example in the area of word order or new types of constraints, but, on the whole, the field of grammar formalisms has stabilised. There are strong convergences among existing systems, as far as some important properties, such as the fundamental data structures, are concerned.

Progress in Constraint Logic Programming has enabled a more uniform understanding of seemingly diverse additions to grammar formalisms as instances of a general CLP schema.

In other areas, such as control, combination with statistical methods, or word order regularities, research activities are just beginning to gain momentum, and more change can be expected in the future.

In this chapter, we will focus on the convergence that has been achieved over the last few years, and which can form the basis of recommendations and standardisation efforts.

4.2 Grammatical Model/Theory

Increased Lexicalisation

While Dependency Grammar and Categorial Grammar have always been strongly lexicalised theories, the trend towards lexicalisation has become very strong within phrase-structure based grammar models such as HPSG, LFG, TAG, and Chomsky's Minimalist framework.

Lexicalisation means that more of the syntactic and semantic information is coded in the lexicon, and that phrase structure rules become fewer in number and reduced in information content. The increased weight put on the lexicon makes it necessary to employ additional mechanisms to organise the lexicon according to linguistic principles and free of redundancy. Such mechanisms are the structuring of the lexicon through multiple inheritance in type hierarchies, and the use of lexical rules.

Levels of Representation

While previous theories have tended to stress only isolated aspects (such as phrase structure, dependency etc.), recent developments tend to integrate more levels in one formalism. The previous period was very innovative with the addition of new datatypes and constraints to formalisms in order to handle specific phenomena. The current trend in grammar models is to focus on the integration of the analyses of various phenomena into one coherent theory.

Transformations

The use of transformations for relating different levels of linguistic description has fallen into disuse for grammatical theories. From the point of view of processing, the reason is that it is very hard to compute the inverses of transformations, which would be required for parsing.

Principles

In many grammar models, a move away from construction-specific rules and towards general, universal and language-specific principles can be observed. Principles are generally formalised as implications or as constraints on types. This shift in grammar models has also had an influence on implemented formalisms, but formalisms which provide full support for principles are still in the research stage.

Discontinuous Constituency

An increasing number of grammar models give up the basis of context-free phrase structure rules and allow discontinuous constituents and more powerful operations for the combination of constituents, such as adjunction or domain union.

This trend has led to research in grammar formalisms, but has yet to find its way into practically usable implemented formalisms that go beyond research prototypes.

4.3 Mathematical Formalisms

Feature-based Representation

Feature structures have come into almost universal use in all formalisms in use today, along with unification as the operation for combining feature structures.

Feature-based representations are a central part of LFG and HPSG, and have been integrated into Categorial Grammars, Dependency Grammars, Tree Adjoining Grammars. Even formalisms based on DCGs often use a feature structure representation which is compiled into Prolog terms. Finite automata have been augmented with feature structures in systems that have to deal with morphological information, such as the DFKI message extraction system.

Hierarchical Structuring

There is a tendency in some formalisms towards hierarchical structuring of linguistic competence, which is often based on type inheritance hierarchies, but sometimes also on templates. There is broad agreement that multiple inheritance is needed for the type system, but it is still a matter of debate whether a greatest lower bound (GLB) semantics is needed, and whether an open-world or closed-world reasoning over types is to be preferred. There is also some debate whether complex types with variables are needed or whether a simple ALE-style appropriateness is enough.

Constraint-based Approaches

Constraint-based approaches have become more important in mathematical formalisms, drawing on the foundational work done in Constraint Logic Programming (CLP). It has become customary to view feature structures and types as special kinds of constraints on variables.

New kinds of constraints, such as set constraints, linear precedence constraints, finite domain constraints, regular expressions, and tree constraints have been investigated and formalised in recent years, and are being added to implemented grammar formalisms.

4.4 Implemented Formalisms

In implemented formalisms there are two tendencies: on the one hand, there is the trend to view NLP as an instance of CLP, and design a grammar formalism as a specialised instance of a CLP language; on the other hand, there are formalism developments which are not based on close relationship to logic programming, for example XLE.

Less Procedural Attachments

Compared to earlier systems, such at ATN, Slot Grammars etc., there is less reliance on external routines (procedural attachments) in today's formalisms. This trend is motivated by two issues: the reusability of grammars in systems which may provide other kinds of external routines, and the bidirectional use of grammars for parsing and generation. The move away from external routines is facilitated by the provision of a richer set of data structures and constraints within the core formalism.

HPSG-inspired formalisms

There are more implemented formalisms which are inspired by HPSG than by any other grammatical theory. This is due to the fact that HPSG makes use of a rich set of formal devices, most of which are present in implemented formalisms. However, there is no one formalism which covers HPSG completely. For this reason, and also because of its principle-based nature, HPSG has provided a good deal of inspiration and challenges to developers of grammar formalisms.

Trend towards Typed Feature Formalisms

The use of types in grammar formalisms has become more important, which is, on the one hand, inspired by HPSG, and, on the other hand, in line with a trend towards (weak or strong) typing in functional (Haskell, Curry) and logic (LIFE, Escher) programming languages.

Implementation in Prolog and LISP

Most formalisms are still being developed in Prolog and LISP. The prediction made some years ago that more efficient low-level implementations of feature formalisms would be done in languages such as C has not become a reality, with the exception of XLE. The continued use of Prolog and LISP is partly due to the fact that formalisms are — despite the convergence — still an area of active research, so that a consolidated low-level implementation would run the risk of quickly falling behind the state of the art. On the other hand, "lean formalisms" have been discovered as an efficient Prolog-based implementation method, which ensures that typed feature structure unification can be done efficiently as Prolog term unification by choosing appropriate encodings. Advanced CLP languages such as LIFE or Oz have not yet had a significant impact on the field of grammar formalisms.

4.5 Development Platforms

Required Functionality

The necessary functionality of a development platform is more or less agreed upon by grammar writers. It must contain a viewer for charts or trees and for feature structures, which should make it possible to focus on only one particular part of a large structure.

They should have a parser for testing the developed grammars with respect to an input string, and possibly also a generator to test for overgeneration. Ideally, there should be automated testing of the grammar against a test suite corpus of sentences.

In order to manage grammar development professionally, they should have a version management facility, and support for multi-linguality.

Due to resource limitations, these requirements are often only incompletely implemented in existing formalisms.

Open Systems

There is an emerging tendency towards open grammar development shells into which different formalisms and different parsers/generators can be embedded.

Such an approach not only saves development time of these components, but also makes it possible for a grammar writer to work with different formalisms while using the same environment.

4.6 Implemented Grammars

HPSG-inspired Grammars

Many of the grammars being implemented today are more or less strongly inspired by HPSG. This is true for many EU-funded projects, such as LS-GRAM, for the German machine translation *Verbmobil* project, and others. A list of such projects is given in table 4.1.

However, large-coverage grammars are also implemented in LFG, TAG and a number of independently developed theories such as TUG.

Grammar Migration

There is a trend to re-use grammars developed in one formalism for other formalisms. The EU projects ET 10/52 and REFMAN worked towards the reuse of grammars developed in the machine translation project EUROTRA. In Verbmobil, HPSG grammars are being translated to CUF, and to TAG for generation.

Systems for Large Grammar Development

The development of large grammars is only done in a few of the implemented formalisms. For HPSG these are mostly in TDL, CUF, ALE and ALEP, for Tree Adjoining Grammars the XTAG system, and for Lexical Functional Grammars the LFG workbench and XLE.

The reason is that these formalisms are reasonably efficient even for larger grammars, and that they provide comfortable development platforms.

Shorter Development Times

The development times for medium-sized grammars have shortened from close to a decade to a few years. This can be attributed to a number of reasons, the most important of them are that

- a clearer structuring of the linguistic knowledge is possible in typed feature formalisms,
- there are better facilities for the inspection, consistency checking, debugging and validation of grammars,
- the gap between the linguistic theory and the formalism has become narrower, and
- there is more experience with the writing of computational grammars, and training in grammar writing has become part of computational linguistics curricula.

Today, there is a trend to employ theoretical linguists as grammar writers, instead of the linguist-programmer of the 1980s.

General-purpose Grammars

There is still a trend to write general-purpose grammars rather than specialised ones that are tailored for a specific application. Given the significant investment in grammar development, it makes sense to make grammars as general and re-usable for different applications as possible.

However, the methods for deriving grammars for specialised applications from these general-purpose grammars through corpus-based methods are still in their nascent stages.

Project	Topic	Formalisms	KECU
ALEP	*Advanced Language Engineering Platform*		*2755*
RGR	*Reusable Grammatical Resources*	*HPSG, LFG, CG, GB*	*790*
LS-GRAM	Large Scale Grammars for EC Languages	TFL, HPSG (ALEP)	1574
DELIS	Lexical Specification, Lexicon Building	TFL	1350
CRISTAL	Retrieval of Information	MI concept hierarchies	1612
DISCOURSE	Declarative Theory of Discourse	ALE, ALEP, HPSG	710
FraCaS	Framework for Computational Semantics	DRT, QLF, HPSG	975
MULTEXT	Multilingual Text Tools and Corpora	ALEP	3210
ANTHEM	Multilingual Generation in Health Care	TFL	1194
TAMIC	Information Access for the Citizen	HPSG	500
RefMan	Eurotra Grammar Migration etc.	TFL (HPSG)	1500
DYANA	*Dynamic Approaches to NL*	*CUF, TFL, HPSG, CG*	*2000*
Verbmobil	Speech Translation	HPSG, TAG, TUG, STUF	45000
PARADICE	*NL Core Engine / Performance Research*	*HPSG, TDL*	*450*
COSMA	Cooperative Schedule Management	HPSG, TDL	730

Table 4.1: Selected Projects Involving Advanced Linguistic Formalisms

Chapter 5

Needs for Industry and Research

It is not possible to give one fixed answer for the needs which industry and/or research have for grammar formalisms. There are probably as many needs as there are NLP projects. A project for grammar checking has other needs than a machine translation project, which, in turn, has different needs from a project researching the latest ideas in grammatical theory.

Therefore, it is unlikely that there can be one single formalism that fulfills all the needs of different user groups.

In the following we will list some needs that may or may not be relevant to a particular project, and discuss the influence of these needs on the requirements of a formalism.

Expressivity

Sufficient expressivity for a given task is the most important requirement of a grammar formalism, since a formalism that does not allow a grammar writer to express the essential concepts of his grammar is next to useless.

However, the question of expressivity must be considered more closely: Regarded from the point of view of formal language theory, almost all grammar formalisms have the generative capacity of Turing machines or Type-0 grammars. In this sense, a very "lean" formalism such as DCG is expressive enough.

However, the formal generative capacity is not a very good criterion for measuring the adequacy of a grammar formalism because grammars should not only show the right input/output behaviour, but should also be (relatively) easy to write, understand, debug and maintain. In order to achieve this, a grammar formalism must not just allow some encoding of grammatical knowledge, but one that conforms to the grammar writer's intuitions about linguistic structures.

This is not to say that a grammar formalism must directly support all the data structures that are used in a grammatical theory. It is often possible to encode one data structure by clever use of another one. Very important in this respect are the investigations by Chris Mellish about term encodings [Mellish, 1988; Mellish, 1992] and by Steve Pulman about the expressive power of lean formalisms [?]. Case studies in this respect are the encoding of linear precedence constraints in feature formalisms [Engelkamp *et al.*, 1992] and the use of quasi logical forms to represent underspecified quantifier scope [Alshawi and Crouch, 1992].

A formalism should also provide means for the transparent and non-redundant structuring of grammatical knowledge. This can either be achieved by means of types (if the type system is expressive enough to allow the specification of complex types), or by macros, or by a combination of both.

The need for expressivity can conflict with other needs, such as the need for computational efficiency. However, although it is often true that less expressive formalisms are more efficient, and vice versa, this is not a necessary correlation. In fact, it may be that using a less expressive formalism leads to more search (due to the fact that some form of underspecification or disjunction is not supported) or to huge data structures (because of the encodings employed), both of which can result in a degradation of performance. In contrast, a formalism which offers more expressivity, such as disjunction or different kinds of constraint solvers may be more efficient because non-deterministic search is replaced by constraint propagation.

The need for expressivity depends very strongly on the use to which the formalism is put in a particular project. A project which works with controlled languages may need less expressivity than a project which tries to validate leading-edge grammatical theories.

Convenient Notation, Support for Modularity

If grammars need to be written and/or maintained, it is crucial to provide a good notation that is compact and readable. The sweeping success of (typed) feature structures over first-order terms attests to this need.

While macros may be regarded as pure abbreviatory devices, they often play a crucial role in the structured development of grammars and lexicons. The use of macros makes it possible to state definitions of data structures only once in one place, and refer to them only by name on all other occurrences. In this way, grammar and lexicon become modular, and datastructures can be changed by changing the definition in one place. Parameterised macros were very successful in this respect.

Availability of Grammars

Now that NLP is moving from a basic research field to a more mature applied discipline, most projects have neither the inclination nor the resources to develop new grammars from scratch. Therefore, the availability of grammars for a formalism is of great importance, and may strongly influence the choice of a formalism.

The same holds for basic research projects, which often want to make use of existing grammars in order to use them for other kinds of research, such as the syntax-semantics interface or discourse models.

Unfortunately, it is not easy to port grammars from one formalism to another. Not only is an automatic translation process sometimes made hard by ideosyncracies of notation, but it is also not clear whether the translated grammars will have an acceptable runtime since grammars are often 'hand-tuned' for the formalism in which they were developed, and include a certain amount of control information.

Attractive Development Environment

For the development of large grammars, there is a definite need to have adequate tools for the inspection of grammars, type hierachies, lexical entries, and processing results. After a grammar has reached a certain complexity, it becomes virtually impossible to understand and manage it without a proper set of tools.

Of equal importance for grammar development are test suites for measuring the coverage of a grammar.

Interfaces

The question of interfaces becomes important when grammars and grammar formalisms are integrated into larger systems. The interfaces should be specified in an open fashion, so that it becomes possible to connect a grammar to an existing lexicon database, for example, or to use other parsers or generators than those provided with the system.

Support for Multilinguality

It can often be observed that formalisms which were developed with machine translation in mind support the need for multilingual grammatical specifications, whereas most others sadly neglect this issue.

It may be possible to augment a formalism to support multilinguality, as it was done by Sharp for the ALE formalism [?].

Modifiability

Often, it is desirable to modify certain aspects of a formalism in order to adapt it to one's needs. Therefore, it is important to have access to the source code, and the source code must be well-documented.

There is, however, the problem of upgrading to newer versions of the formalism. In this case, any modifications made to the formalism must either be replicated in the newer version, or one must continue to use the older version. Modification and adaptation of formalisms can thus lead to a proliferation of incompatible versions of a formalism.

Availability on Different Platforms

For applied systems, the availability of a formalism on different platforms (e.g. different programming languages/dialects, different operating systems, different hardware) can be an important factor.

In order to build a formalism into real consumer applications, it would be necessary to compile either the formalism or the grammar plus formalism into object code, which can be linked to applications. As far as we know, there is currently no grammar formalism which supports this directly.

Support and Maintenance

In case of problems with a grammar formalism, there is the need for technical support. With the exception of the ALEP formalism, there is currently no formalism for which professional support services are offered. In the case of formalisms developed within research projects, the level of support will often decline drastically when the project is over, or when the developer of the formalism leaves the institution. On the other hand, researchers who developed a formalism may be more willing to make changes to a formalism than a professional support organisation.

Usability for Different Purposes

There is often a need to use a formalism for slightly different purposes than those for which it was originally developed. Most formalisms were developed with syntactic processing in mind, but they have become used for different purposes such as discourse representation, semantic representation and processing or knowledge representation and processing.

For some of these purposes, it is desirable to mix the datastructures of the formalism and those of the underlying programming language, as in ProFIT.

Availability of Tools

In order to make use of a formalism, there must be tools for processing, such as a parser and/or a generator. Since different approaches to processing may be required for different applications, it is preferable to provide a framework such as Hdrug, in which different formalisms and different processing engines can be combined through clean interfaces.

Efficiency

In practically applied systems, speed is of great importance to avoid waiting time in interactive systems, or to achieve a better throughput in applications that must handle large amounts of text, such as message understanding.

The time needed for processing depends very much on the complexity of the grammar, the size of the input, and on processing strategies. Lean Formalisms, which use the data structures and operations of the underlying programming language at runtime have very good prospects of achieving an acceptable efficiency. The same is true of formalisms which are implemented in a lower-level programming language such as C by making use of an abstract machine approach.

Software Engineering Tools

A grammar formalism should contain many of the tools that are useful for software development, such as a version control system, and facilities for systematic testing of the grammar against a test suite of well-formed and ill-formed example sentences, which illustrate syntactic phenomena.

Chapter 6

Recommendations Concerning Formalisms for New NL Projects

Chapter 5 discussed the criteria which are relevant for the choice of a formalism for a particular project. This chapter provides some practical information how to find more information about particular grammar formalisms, and how to obtain and install formalisms and grammars.

In recent years, more and more of this information has become available through the internet, especially through the World Wide Web. In this chapter, we give some starting points, which will lead to a wealth of information about current grammar formalisms.

6.1 Where to get help and information

6.1.1 World Wide Web

EAGLES
The EAGLES webserver contains information from all the EAGLES working groups, including an online version of the document you are reading now, which contains links to all the mentioned sources of information is available on the WWW through the EAGLES webserver.
URL `http://www.ilc.pi.cnr.it/EAGLES/`

NL Software Registry
The Natural Language Software Registry has collected information about NL tools, including formalisms.
URL `http://cl-www.dfki.uni-sb.de/cl/registry/draft.html`

European Language Resource Association
The RELATOR server contains information about NL resources (including formalisms and grammars):
URL `http://www.relator.research.ec.org/`

Consortium for Lexical Research
The Consortium for Lexical Research has an archive of NL software and resources; however, it is not updated due to lack of funding.
URL `http://clr.nmsu.edu/Tools/`

HPSG-related Formalisms
A list of implementations of grammar formalisms which are relevant for HPSG is maintained at Ohio State University:

URL `http://julius.ling.ohio-state.edu/HPSG/Implementation.html`

6.1.2 Helpdesks

The Language Technology Group at the University of Edinburgh provides a helpdesk service for NL software, especially grammar formalisms. It can be reached through the following addresses:

Language Software Helpdesk
c/o Language Technology Group
Human Communication Research Centre
2 Buccleuch Place
Edinburgh EH8 9LW
UNITED KINGDOM
e-mail: `Language.Software.Helpdesk@ed.ac.uk`
fax: +44 131 650 4587 (marked "Language Software Helpdesk")

6.1.3 Introductory Reading

Unfortunately, it is not easy to find a set of readings that will give a good introduction and overview of the field of grammar formalisms. The overview given in this report is the most comprehensive one that is avialable. In this subsection, we will provide references to a small selection of good introductory readings, rather than a comprehensive bibliography.

A good general overview of the problems involved in developing, testing, and maitaining large-scale grammars is found in the report of the first grammar engineering workshop [Erbach and Uszkoreit, 1990].

One widely accepted view of the mathematical foundations which underly typed feature structures is found in Bob Carpenter's book *The logic of Typed Feature Structures* [Carpenter, 1992a].

For grammar models, there are often very good introductions available:
HPSG: [Pollard and Sag, 1994b]
LFG: [Kaplan and Bresnan, 1982; Dalrymple and Zaenen, 1995]
TAG: [Joshi, 1987b]
GB: [Webelhuth, 1995]
CG: [Morrill, 1994b]

The following are the key readings for several implemented formalisms:
Core Language Engine: [Alshawi, 1991]
CUF: [Dörre and Dorna, 1993]
TDL: [Krieger and Schäfer, 1994]
ProFIT: [Erbach, 1995]
TFS: [Zajac, 1992a]

For implemented grammars and development environments, there are no introductory readings, but it is possible to refer to the technical reports referenced in chapters 2 and 3.

6.1.4 Other Information Sources

The following resources are not concerned exclusively with grammar formalisms, but offer a broader range of information, and links to such information, about all apects of computational linguistics and natural language processing.

ACL CL/NLP Universe
For other information about the field of NLP, the *CL/NLP Universe* of the ACL is an excellent starting point.
URL: `http://www.cs.columbia.edu/~radev/cgi-bin/universe.cgi`

Index of Computational Linguistics Institutes
The University of Stuttgart maintains a fairly complete list of links to computational linguistic institutes all over the world, which offer online information through WWW and ftp servers.
URL: `http://www.ims.uni-stuttgart.de/info/FTPServer.html`

ELSNET
The European Network in Language and Speech (ELSNET) comprises industrial and academic NLP groups, and offers a WWW server.
URL: `http://www.cogsci.ed.ac.uk/elsnet/goals.html`

6.2 Information on Formalisms, Grammars and Development Environments

The following sections summarise the information throughout the report on existing formalisms, grammars and development environments. The systems are listed alphabetically.

6.2.1 ALE – Attribute Logic Engine

System type: Implemented formalism

Theoretical basis: ALE was developed to handle HPSG grammars. It can also execute PATR-II grammars, DCG grammars, Prolog, Prolog-II, and LOGIN programs, etc.

Envisaged use: A freeware logic programming and grammar parsing system.

Licence: the system and its documentation are available without charge for research purposes.

System requirements: ALE can be run in either SICStus or Quintus Prolog, and with other compatible compilers doing first-argument indexing and last-call optimization.

Source code: `http://macduff.andrew.cmu.edu/ftp/ale/`

Documentation: Bob Carpenter and Gerald Penn (1994) ALE 2.0 User's Guide. Carnegie Mellon University. Laboratory for Computational Linguistics Technical Report. Pittsburgh.
`http://macduff.andrew.cmu.edu/ale/guide/guide.html`
`http://macduff.andrew.cmu.edu/ftp/ale/`

Homepage: `http://macduff.andrew.cmu.edu/ale/index.html`

Contact: Bob Carpenter

Address: Bell Laboratories, Room 2D-329, 600 Mountain Avenue, Murray Hill, New Jersey 07974

Tel.: (908) 582-5790

Fax.: (908) 582-3306

Email: `carp@research.bell-labs.com`

6.2.2 ALEP – the Advanced Language Engineering Platform

System type: Implemented formalism and development environment.

Theoretical basis: Theory-neutral formalism to a certain extent.

Envisaged use: A versatile and flexible general purpose NLP platform, the Advanced Language Engineering Platform (ALEP) is suitable for a wide range for mono/multi-lingual applications, is open and highly customizable.

Licence: Research and commercial (specific licence for commercial use).

System requirements: SPARC stations of Sun Mircosystems with at least 32 MB of RAM.
SunOS 4.1.3(4) or Solaris 2.4(5). Motif 1.2. Quintus Prolog. Some GNU utilities which are included in the ALEP. distribution package.

Source code: See contact address below.

Documentation: The ALEP Project, Cray Systems
User documentation: ALEP-1 User Manuals, ALEP-2 User Manuals.
System documentation: ALEP-1 System Documentation, ALEP-2 System Documentation.
`http://www.anite-systems.lu/alep/doc/index.html`

Homepage: `http://www.cray-systems.lu/alep`

Contact: Mr Neil Simpkins

Address: Cray Systems, ALEP Support, 11b Bvd Joseph II, L-1840 LUXEMBOURG

Tel.: +352 25 08 90/91

Fax.: +352 25 08 92

Email: `neil@cray_systems.lu`

6.2.3 CL-ONE

System type: Implemented formalism (under development).

Theoretical basis: An NLP formalism with sets and linear precedence constraints.

Envisaged use: The formalism is designed for the direct representation of linguistic theories such as HPSG by providing set descriptions and set constraints and linear precedence constraints.

Licence: Public domain (on completion of system).

System requirements: Sicstus Prolog 2.1#9 or higher, preferably on a Unix workstation.

Source code: `ftp://ftp.coli.uni-sb.de/pub/coli/proj/rgr/cl-one/`

Documentation: Not yet available.

Homepage: Not yet available.

Contact: Gregor Erbach

Address: DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken

Tel.: +49 681 302 5288

Fax.: +49 681 302 5341

Email: `erbach@dfki.uni-sb.de`

6.2.4 CUF – Comprehensive Unification Formalsim

System type: Implemented formalism.

Theoretical basis: Theory neutral.

Envisaged use: CUF provides a formalism for declarative description of linguistic phenomena independent of the linguistic area and a system for processing CUF descriptions.

Licence: Freely available.

System requirements: The implementation runs under Quintus and SICStus PROLOG under UNIX and X11.

Source code: `ftp://ftp.ims.uni-stuttgart.de/pub/cuf/`
or write to `cuf-request@ims.uni-stuttgart.de or contact the address given below.`

Documentation: Diploma Thesis and small paper in German. Report and short documentation in English.
Papers on CUF and its applications:
`ftp://ftp.ims.uni-stuttgart.de/pub/cuf/english_papers/`

Homepage: `http://www.ims.uni-stuttgart.de/cuf/`

Contact: Jochen Dörre

Address: Institut für maschinelle Sprachverarbeitung, Universität Stuttgart, Azenbergstr. 12, D-70174 Stuttgart, GERMANY

Tel.: +49 0711 121 1357

Fax.: –

Email: `Jochen.Doerre@ims.uni-stuttgart.de`

CUF Users Group: write to `cuf-request@ims.uni-stuttgart.de` for subscribing to the CUF Users Group and for further questions concerning CUF.

6.2.5 ENGCG – English Constraint Grammar Parser

System type: Implemented grammar.

Theoretical basis: Constraint Grammar.

Envisaged use: ENGCG, the Constraint Grammar Parser of English, performs morphosyntactic analysis (tagging) of running English text.

Licence: see contact below.

System requirements: See contact below.

Source code: See contact below.

Documentation: See contact below.

Homepage: `http://www.lingsoft.fi/doc/engcg/intro`

Contact (1): Lingsoft, Inc.

Address: Museokatu 18 A, FIN-00100 HELSINKI, FINLAND

Tel.: +358 0 499 556 (+358 9 499 556)

Fax.: +358 0 440 602 (+358 9 440 602)

Email: `info@lingsoft.fi`

Contact (2): Atro Voutilainen

Address: Department of General Linguistics, P.O. Box 4, FIN-00014 University of Helsinki, Finland

Tel.: +358 0 191 23 507

Fax.: +358 0 191 23 598

Email: `Atro.Voutilainen@Helsinki.FI`

Online analysis: You can input one or more English sentences to ENGCG for a demo analysis via the following web page: `http://www.lingsoft.fi/cgi-pub/engcg`

6.2.6 Hdrug

System type: Development environment.

Theoretical basis: Theory neutral.

Formalisms supported: ALE: ALE 2.0 HPSG grammar
HPSG: HPSG grammar for Dutch
TAG: small Tree Adjoining Grammar
DCG: DCG for Dutch
Constraint-based CG: Constraint-based Categorial Grammar for English
Small DCG: the smallest possible DCG
Extraposition Grammar: Extraposition grammar based on the paper by Fernando Pereira.

Envisaged use: Hdrug is an environment to develop logic grammars / parsers / generators for natural languages.

Licence: The Hdrug program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.

System requirements: In order to use Hdrug you need SICStus Prolog version 2.1 #8 or #9. A port to SICStus 3.1 is foreseen (using the Tcl/Tk interface that comes as a standard library in Sicstus 3.1). Furthermore, you need ProTcl 1.4, including the standard Tcl/Tk distribution with the BLT and TkTree extensions.

Source code: The Hdrug package is available through anonymous ftp in directory:
`ftp://ftp.let.rug.nl/pub/prolog-app/Hdrug/`
Alternatively the same file is accessible through World Wide Web:
`http://www.let.rug.nl/~vannoord/Hdrug/`

Documentation: Hdrug comes with extensive documentation, which is accessible through the World Wide Web:
`http://www.let.rug.nl/~vannoord/Hdrug/`

Homepage: `http://www.let.rug.nl/~vannoord/Hdrug/`

Contact: Gertjan van Noord

Address: Oude Kijk in 't Jatstraat 26, NL-9700 AS Groningen, Niederlande

Tel.: +31 50 3 6359 35

Fax.: –

Email: `vannoord@let.rug.nl`

6.2.7 The LFG Grammar-writer's Workbench

System type: Implemented formalism and development environment.

Theoretical basis: Lexical Functional Grammar.

Envisaged use: The LFG Grammar-writer's Workbench is a computational environment that assists in writing and debugging Lexical Functional Grammars. It provides linguists with a facility for writing syntactic, lexical and morphological rules, and testing and editing them.

Licence: Available at no cost from Xerox for research and educational use. It is no longer necessary to obtain a licence from Venue.

System requirements: Original Xerox AI workstations and a wide variety of Unix workstations (Sun, DEC, IBM, MIPS, HP, etc.) and certain PC compatible platforms running MS-DOS. The Medely Lisp programming environment. It does not have a teletype interface–it only runs as a graphical program. It requires at least 16MB of ram on UNIX and 8MB under DOS, plus 40 or more MB of disk (for program storage and swapping).

Source code: `ftp://parcftp.xerox.com/pub/lfg/`

Documentation: Postscript: `ftp://parcftp.xerox.com/pub/lfg/lfgmanual.ps`

Homepage: `http://clwww.essex.ac.uk/LFG`

Contact: Ron Kaplan

Address: Xerox Parc, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA

Tel.: 0 01 (4 15) 4 94 43 48

Fax.: –

Email: `kaplan@parc.xerox.com`

6.2.8 PAGE – Platform for Advanced Grammar Engineering

System type: Development environment

Theoretical basis: Typed feature logics

Envisaged use: An advanced NLP core engine that especially facilitates the development of grammatical resources building on typed feature logics (eg. for HPSG-style frameworks).

Licence: The PAGE system is made available upon request, free of royalties, to interested parties; DFKI reserves the software copyright and intellectual property rights and requires recipients of the PAGE system to sign a non-commercial license agreement.

System requirements: The PAGE system is implemented in Common-Lisp and ANSI C and builds on standardized graphical tookits (the X Window System and Motif) in order to achieve a great degree of portability. In general, it should be possible to port the system to most Unix workstation for which a suitable Common-Lisp implementation, C compiler, and X and Motif are available.

Source code: See contact below.

Documentation: See homepage and contact below.

Homepage: `http://cl-www.dfki.uni-sb.de/cl/systems/page/page.html`

Contact: Klaus Netter

Address: DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken

Tel.: +49 681 302 5283

Fax.: +49 681 302 5341

Email: `netter@dfki.uni-sb.de`

PAGE mailing list: For subscription to the page mailing list send email to `page-users-request@cl.dfki.uni-sb.de` (as `page-users` is an unmoderated forum, please make sure to send administrative requests — subscription and unsubcription — to the *request* address rather than to the entire list).

6.2.9 Pleuk

System type: Development environment.

Theoretical basis: Theory neutral.

Formalisms supported: HPSG-PL: a Prolog implementation of the HPSG formalism, developed by Fred Popowich, Sandi Kodric and Carl Vogel
Cfg: a simple context-free grammar system, intended for demonstration purposes
Mike: a simple graph-based unification system
SLE: a graph-based formalism enhanced with arbitrary relations
Term: a term-based unification grammar system
Ale: the Attribute Logic Engine by Carpenter and Penn
vNTag: van Noord's TAG system
definite clause grammars

Envisaged use: Pleuk is a grammar development shell within which many different grammatical formalisms can be embedded

Licence: There are no licensing restrictions for non-commercial use

System requirements: The system is known to run on Sun workstations, under Sun OS 4.1 and later versions, as well as Sun OS 5.2 and later versions. It also runs on Hewlett Packard 9000s, under HPUX. A version of SICStus Prolog between 2.1.6 and 2.1.9 is required. Plans are being made for a port to later versions of SICStus Prolog.

Source code: `ftp://scott.cogsci.ed.ac.uk/pub/pleuk`

Documentation: User documentation: "Pleuk Overview", "Interface", some formalisms (manuals in TeXinfo format, both hard copy and on-line)
System documentation: "Functional backbone" (Core code, same format)

Homepage: `http://fas.sfu.ca/0/cs/research/groups/NLL/3.html#3.2.2`

Contact: Jo Calder

Address: University of Edinburgh, Centre for Cognitive Science, 2 Buccleuch Place, Edinburgh EH8 9LW, Scotland

Tel.: +44 131 650 46 56

Fax.: –

Email: `J.Calder@ed.ac.uk`

6.2.10 ProFIT – Prolog with Features Inheritance, and Templates

System type: Implemented formalism.

Theoretical basis: The ProFIT system is an extension of Prolog with sorted feature structures (including multi-dimensional inheritance), finite domains, feature search, cyclic terms, and templates.

Envisaged use: ProFIT can be used to extend your favourite Prolog-based grammar formalism, parser and generator with the expressive power of sorted feature terms. ProFIT works as a pre-processor, which takes a file containing a ProFIT program as input, and gives a file with a Prolog program as output.

Licence: ProFIT is available free of charge.

System requirements: ProFIT has been implemented in Sicstus Prolog (2.1 #9).

Source code: `ftp://ftp.coli.uni-sb.de/pub/profit`

Documentation: `http://coli.uni-sb.de/~erbach/formal/profit/`

Homepage: http://coli.uni-sb.de/ erbach/formal/profit/profit.html

Contact: Gregor Erbach

Address: DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken

Tel.: +49 681 302 5288

Fax.: +49 681 302 5341

Email: `erbach@dfki.uni-sb.de`

ProFIT mailing list: There is a mailing list for ProFIT users. Send mail to `erbach@dfki.uni-sb.de to subscribe or unsubscribe. Send mail to profit-users@coli.uni-sb.de to reach the entire list.`

6.2.11 TAG-GEN

System type: Implemented formalism and grammar.

Theoretical basis: Unification–based Tree–Adjoining Grammar with context–dependent disjunctive linearization rules (CDL-TAG)

Envisaged use: TAG-GEN is a syntactic generator that exploits an incremental and parallel processing scheme. The goal is to build a system that both runs in real-time and produces output that is highly adaptive to expansions and changes in the input. Tree Adjoining Grammars are used as syntactic representation formalism and have demonstrated their adequacy in supporting incremental processing.

Licence: Research, see email contact.

System requirements: Common Lisp, Flavors, CLOS.

Source code: See email contact.

Documentation: See email contact.

Homepage: –

Contact: Wolfgang Finkler, Anne Kilger

Address: DFKI, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany

Tel.: See email contact.

Fax.: +49 681 302 5341

Email: `kilger@dfki.uni-sb.de`
`finkler@dfki.uni-sb.de`

6.2.12 TDL – Type Description Language and Inference System

TDL is part of the PAGE distribution, see page 135 for details.

6.2.13 The XTAG Workbench

System type: Implemented formalism and development platform.

Theoretical basis: FB-LTAG (Feature-Based Lexicalized Tree Adjoining Grammar)

Envisaged use: XTAG is an on-going project to develop a wide-coverage grammar for English using a feature-based and lexicalized Tree Adjoining Grammar formalism. XTAG also serves as an system for the development of TAGs and consists of a predictive left-to-right parser, an X-windows interface, a morphological analyzer, and a part-of-speech tagger.

Licence: Research, see email contact.

System requirements: The system is currently implemented in Common Lisp and C. The graphical interface runs under X windows.

Source code: `ftp://ftp.cis.upenn.edu:/pub/xtag/`

Documentation: `ftp://ftp.cis.upenn.edu/pub/cliff-group/public_html/94/xtag.html`

Homepage: `http://www.cis.upenn.edu/~xtag/`

Contact: XTAG team

Address: 3401 Walnut Street, Suite 400C, University of Pennsylvania, Philadelphia, PA 19104-6228

Tel.: See email contact.

Fax.: +1 215 573 9247

Email: `xtag-request@linc.cis.upenn.edu`

Chapter 7

Exchange Formats for Grammar Formalisms

7.1 Introduction

In this chapter, we propose draft encoding standards (henceforth DES) for three grammatical frameworks, namely Lexical-Functional Grammar (LFG), Head-Driven Phrase Structure Grammar (HPSG), and Tree Adjoining Grammars (TAG).

Our decisions with respect to the choice of the grammatical frameworks and to the content of the encoding standards were mostly determined by practical considerations. The guiding principles in our decisions were the following:

1. Currently existing large grammars must be encodable in the DES.
2. Grammars encoded in the DES must be translatable (automatically) into existing implemented formalisms.
3. The DES must offer the flexibility to accommodate future developments in grammatical theory and in grammar formalisms, and should not limit itself to the present state of the art.

The following paragraphs give some justification for the decisions we have taken, and some arguments for the alternatives we have not been able to pursue. We will provide answers to the following set of questions:

- Why is an encoding standard needed? Why not a standard formalism?
- Why different encoding standards for different grammatical frameworks? Why not a uniform encoding standard?
- Why were LFG, HPSG and TAG chosen?
- What should the expressivity of the encoding standard be? The union, the intersection, or some compromise between the expressivities of various formalisms?
- What is the relation between the encoding standard and different grammatical frameworks? What is its relation to implemented grammar formalisms?
- How can the encoding standard evolve with progress in grammatical theories and in the implementation of grammar formalisms?
- How can existing grammars be translated to the encoding standard?
- Who should use the encoding standard?

Motivation

The purpose of a draft encoding standard is to provide a precise formalised language for representing linguistic knowledge within one grammatical framework, which is independent of a particular implemented grammar formalism.

In order to be usable, the grammars encoded in the DES must be translated into an implemented grammar formalism. Such a translation should take place automatically. A grammar translated from the encoding standard to a particular formalism cannot be expected to run with the best efficiency in a given formalism, but may still require some hand-tuning.[1] However, compared to the alternative of developing or manually converting a grammar, automatic translation and hand-tuning appears to be a reasonable and cost-efficient option.

The question arises naturally as to why we do not propose one standard formalism that meets all these needs and comes with efficient parsing and generation algorithms and a nice collection of grammar development tools.

As we have argued in section 6, different formalisms have their *raison d'être*, since they meet different needs that arise in different NLP projects.

The DES should serve as an encoding format for a repository of grammatical knowledge, which can be re-used within different formalisms for different projects and different kinds of requirements.

Of course, we do not discourage developers of existing or future grammar formalisms from providing direct support for grammars encoded in the DES.

How many encoding standards are needed?

Of course it would be very nice to have one common encoding standard that tears down the barriers which exist between different formalisms, and allows grammars which take up and combine the best ideas from various grammatical theories to be encoded.

However, practical considerations have discouraged us from taking this route.

First of all, it would be extremely hard to find translations from such a high-level encoding to existing implemented grammar formalisms, for example, to make an LFG grammar run in a TAG formalism. That such translations are possible and constitute an interesting research problem is shown by such efforts as the translation of HPSG grammars to TAG (cf. [Kasper *et al.*, 1995]). For the purposes of this report, we have set more modest goals, which are equally important because the translation between different formalisms for the same grammatical framework is also an urgent and unsolved problem.

While a uniform encoding standard would be very satisfactory from the point of view of generalisation and mathematical elegance, it might be at a level of abstraction that is too high for everyday practical grammar writing. For example, the familiar phrase structure rules or initial and auxiliary trees of TAG would have to be described by a set of tree constraints (dominance and precedence) instead. These are clearly exciting research problems which need to be addressed, but we have refrained from doing so for the purposes of this report.[2] .

Why were LFG, HPSG, and TAG chosen?

Given the resources of the project, it was not possible to cover more than three grammatical frameworks with reasonable depth and attention to detail. The choice of LFG, HPSG and TAG was influenced by the fact that these are used at a number of sites, and that the different grammars developed in these theories show enough commonalities to be encoded in one DES. Further,

[1] The requirement for hand-tuning arises because it is necessary for most grammars to add some kind of control information to the declarative specification. This control information can either be implicit in the structuring of the grammars, or be explicitly annotated, such as the designation of one daughter as the head of a phrase for ALEP.

[2] It may be instructive to compare this to the situation in programming languages. The most universal programming languages, such as Turing machines, or the lambda calculus, are appealing because of their mathematical elegance and abstraction, but rarely used for practical programming. Among the practically used languages, there are some that are used for just one specific task, while others have a wide range of applications and support a variety of programming paradigms. The DES would compare to the latter class.

the different formalisms for these formalisms have enough commonalities to allow translation of grammars encoded in the DES into the formalisms.

LFG, HPSG and TAG all have their fixed and accepted core formalism, which is extended in various directions, whereas the other grammatical theories are more in flux, and there are a number of alternative formalisations. Such active research is a of course very useful and productive, but it also means that standardisation attempts would be premature.

Expressivity

The expressivity of the encoding standards is an issue which is determined by two conflicting requirements.

On the one hand, the DES should serve as a repository of grammatical knowledge, and therefore support the full range of expressive constructs used in different grammars (i.e., the union of the expressivities of various implemented formalisms).

On the other hand, grammars written in the encoding standard should be translatable to a wide range of formalisms. In order to make this possible, the DES should be restricted to what is either directly available or can be hacked up in *all* of the formalisms (i.e., the intersection of the expressivities). In practice, this would mean limiting oneself to what can be encoded in the least expressive formalism.[3]

We have not been able to come up with a compromise that reconciles these two requirements. Taking the intersective approach would severely inhibit the acceptance of the DES among grammar writers who are used to a more expressive formalism, and who have presumably chosen their formalism precisely because of its expressivity. It would also limit the usefulness of the DES in the future, when more efficient implementation techniques are known for constructs that many of today's formalisms exclude for reasons of efficient processing.

Therefore, we relax the second requirement that grammars encoded in the standard should be translatable to *all* formalisms. Instead, we provide an expressive draft encoding standard and indicate clearly which constructs can be translated to which formalisms.

We will specify translations of well-defined subsets of the draft encoding standard to various existing formalisms.

The decision for expressivity at the expense of translatability to *all* formalisms is based on the assumption that the grammatical knowledge encoded in grammars changes more slowly and has a longer lifetime than the formalisms in which it is encoded. This assumption is supported by the observation that typical grammars often have an overall development time of up to a decade, whereas formalisms are replaced every four or five years on average. The encoding standard should allow the same basic grammars to be usable over several successive generations of formalisms.

The encoding standard consists of a core which can be translated to all formalisms, and a set of extensions, for each of which it must always be clear to which formalisms it can be translated.

Relation to Grammar Models

The encoding standard does not try to directly encode a grammatical theory as it is described in books or theoretical papers. Rather it tries to take existing implemented grammars as its starting point.[4] In this sense, the DES should primarily support practical grammar engineering, rather than general research into the datatypes used in linguistic theories.

It is crucial that the present encoding standard only specifies the datatypes and constraints employed, but does not say anything about the grammars encoded with these datatypes. In particular, it does not fix a set of types, or feature names, or appropriateness specifications. We

[3] The question would then arise as to why one should use the encoding standard instead of just the least expressive formalism plus some syntactic sugar.

[4] This may raise the problem that implemented grammars are implemented in the way they are because the formalisms used are so restricted. We do not see this as a serious problem because our experience has shown that if some expressivity is really needed, it will sooner or later be provided in a grammar formalism and show up in implemented grammars.

do not believe that it would be possible to fix these questions of linguistic substance at this time. Unlike the EAGLES Lexicon working group, which builds on a rich lexicographic tradition to specify which distinctions need to be encoded in a lexical database, we do not yet know which set of types and features will be needed (either for universal grammar or for particular languages). We believe that a standard which fixes types and features will be useful, but that it is only possible after much more research has been carried out to arrive at a comprehensive theory of universal grammar and language-specific grammars.

Since it will be essential for practical systems to connect a grammar with a lexical database, we have specified a clean interface which can be used to map entries of a lexical database to the datastructures used for the lexicon of a particular grammatical theory. This interface will allow the results of the EAGLES lexicon working group to be combined with the encoding standards developed by the formalism working group.

Relation to Implemented Formalisms

The core of the encoding standard is a subset of each formalism, and the extended encoding standard is a superset of each formalism.

With respect to each individual construct in the extended DES there are three possible relationships to an implemented formalism:

1. The construct is directly supported by the formalism. In this case the translation is trivial.

2. The construct is not encodable in the formalism. In this case, the translation is impossible.

3. The construct is not provided, but can be encoded or approximated in the formalism. These are the interesting parts of the DES, which will be given special attention in this report.

A problem arises when different formalisms have chosen approaches to a problem that are radically different or even incompatible with each other. This is, for example, the case for formalisms which allow variables in type definitions and formalisms which instead permit relational constraints. These problems will be addressed in the descriptions of the individual encoding standards.

Who should use the DES?

Grammars in the DES are meant to be written and understood by humans. Therefore, much attention has been paid to a clear notation.

The translation to different existing formalisms is done automatically, but will require some hand-tuning for efficiency, since grammars in the DES are fully declarative, and do not embody control knowledge.

An important question is how the encoding standards are to be used in conjunction with existing formalisms. In the case of LFG and TAG, we have a fortunate situation because there are implemented formalisms which use the encoding standard directly. In the case of HPSG, however, there is presently no such formalism. This means that grammars written in the encoding standard must be translated into an existing formalism before they can be used and debugged. It is presently not clear how practical such an approach will be. It will be necessary to develop tools that make this translation as effortless as possible. Ideally, this will be the case, if various formalisms can accept the encoding standard as an alternative notation. We expect that the availability of a number of grammars in the encoding standard will persuade the developers of formalisms to support it directly.

How can the encoding standard evolve?

The DES is defined as an open standard, which can be extended as needed. Future versions will be a superset of the current version.

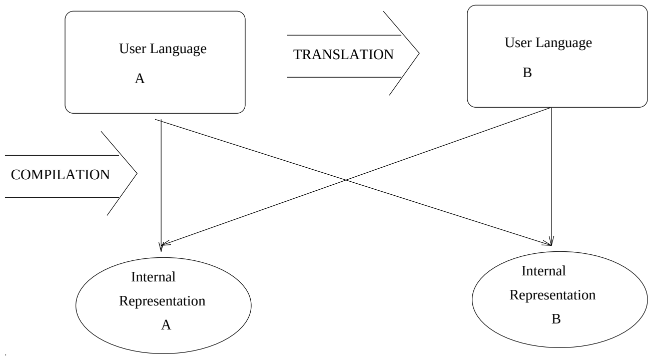


Figure 7.1: Compilation vs Translation

Even the current version contains many constructs that are only supported by some grammar formalisms today, but may become standard in the next generation of formalisms.

How can existing grammars be translated to the DES?

We will not provide tools for fully automatic translation of existing grammars into the encoding standard. The DES should be a format in which grammars are written, and then translated to different formalisms.

7.2 ALEP to TDL migration

Introduction

ALEP2$\mathcal{TDL}$ is a Lisp program able to translate Alep feature descriptions (types, rules, lexical entries, macros) into TDL feature descriptions (types, instances, templates) of the PAGE system [Krieger and Schäfer, 1994; ?]. It was developed as a joint work of CELI (Centro per l'Elaborazione del Linguaggio e del'Informazione) and DFKI (Deutches Forschungszentrum für Künstliche Intelligenz). Its realization took about 20 man-days and largely benefited from the adoption of the compiler Zebu [Laubsch, 1994].

7.2.1 Translation vs. Compilation

In order to make grammars transportable from domain to domain, two basic approaches are conceivable. Either programs are developped which compile different user's languages into the same internal representations, or the user languages themselves are mapped from one into another (fig. 7.1). In ALEP2$\mathcal{TDL}$ the latter approach has been chosen, on the basis of the following facts:

- Having access to the internal representation format of different systems is sometimes difficult, either for lack of documentation or commercial rights. Formalisms, on the contrary, usually come with a good documentation, and, obviously, the problem of protected or compiled code does not arise.
- Compilation of the same grammar into different systems is meaningful only in order to improve the efficiency of a grammar, not to achieve a larger of finer grained coverage. However most of the existing grammars still have to achieve the result of covering an acceptable fragment of a language. Thus the efficiency of the system affects only partially the development process. Moreover, in order to increase the performance of existing grammars much more

suitable technique are at study than simply compilation into an analogous, slightly faster, framework (e.g. compilation of HPSG grammars into TAG).

- The "translation" approach allows a *parallel* development of grammars, in our case Alep Grammars and TDL (or PAGE) grammars. In practice, this means that the output of the translation can be considered by the grammar writer as a real *grammar* which can be debugged, modified or extended.

Once we have shown the desideability of a translation approach over a compilation one, we have still to show why it is desirable to have the same grammar implemented in different formalisms. On this respect, at least the following points can be made:

- The maintenance and the support of unification based frameworks depend on factors which are sometimes independent on the sites where grammars are developed. Having equivalent grammars running on different frameworks is a warrantee against the whims of destiny.
- Different frameworks offer different development tools, and the user may whish to exploit all of them. Just for the sake of exemplification, some user would like to be able to exploit during grammar development both the object oriented architecture of ALEP (in order to handle dictionaries, and test suites) and the graphical facilities of Page, which allows a graphical representation of both rules and lexical entries (which is not possible in ALEP).
- Platforms such as ALEP and PAGE are implemented in different programming languages, Prolog and Lisp, respectively. Even though interfacing a grammar running in a certain programming environment with some module running in a different programming environment is not necessarily a problem, things are made extremely easier if all the modules constituting a NLP system are coded in the same programming language. For instance, it is reasonable to think that a Lisp-based knowledge representation engine such as Rethorique is better interfaceable with Page than ALEP, whereas when we consider a prolog based engine such as Epilogue things are exactly the other way around.

7.2.2 Technology Adopted

The ALEP2$\mathcal{TDL}$ program takes as an input a file of ALEP linguistics descriptions and returns all $\mathcal{TDL}$file of linguistic descriptions. The phases of this process are the following:

1. Normalization: some emacs macros take care of adjusting some 'typographical' details which cannot be dealt with correctly by Zebu. This phase is completely automatic, and amounts to loading an emacs file.
2. Analysis: The zebu compiler analyzes the ALEP file and produces a structure where the ALEP linguistic descriptions are encoded in a more "neutral" formalism (interchange format).
3. Generation: Again Zebu parses the interchange format linguistic description to produce $\mathcal{TDL}$linguistic descriptions.

The division of tasks between analysis and generation is motivated by the fact that the translator has to perform some non-local computations. The clearest example of such a non-local computation is represented by the coding of the type hierarchy. ALEP allows for a split declaration of the type hierarchy and the constraints associated to every type. Thus an ALEP type declaration file can be conceived as mixture of constraints over types and fragments of type hierarchies, the order in which these pieces of information appear being completely irrelevant. The $\mathcal{TDL}$user language, on the contrary, forces, for every type, a conjoint declaration of both its supertypes and the constraints to which such a type is associated. These different strategies can be handled only by assuming a two step strategy. First the ALEP type declaration is scanned, and two different data structures are created, one encoding the constraints which are associated to every type, the other

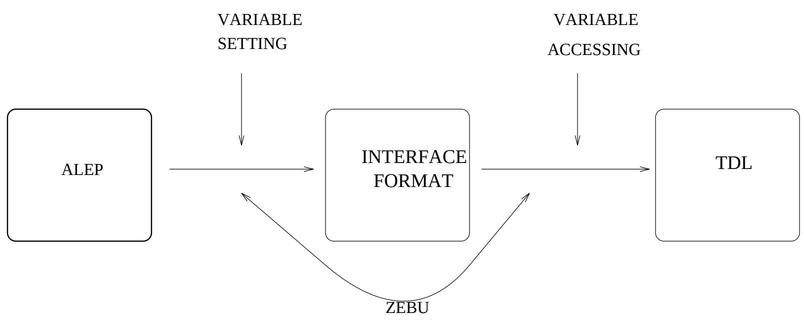


Figure 7.2: The ALEP2$\mathcal{TDL}$ workflow.

one encoding the partial order of the subsumption hierarchy. Then the $\mathcal{TDL}$type descriptions are generated by accessing both these data structures.

Technically, both rules of analysis and generation are encoded in a standard Zebu format, i.e. they are formed of a *test* part, followed by an *action* part. The test part matches the BNF of the input language, whereas the action part is responsible for building the expressions of the target language, mainly through Lisp list manipulating functions. In particular:

- Analysis:

  ```
  Alep BNF -> Interchange BNF + global variable setting
  ```

- Generation:

  ```
  Interchange BNF -> Tdl BNF (through accessing global variables)
  ```

7.2.3 Compatibility Issues

There are some features of both ALEP and $\mathcal{TDL}$which sometimes make the translation from one framework to the other difficult. It is exactly because of these features that, at the current state of art of the ALEP2$\mathcal{TDL}$ module, the translation between ALEP and $\mathcal{TDL}$cannot be considered completely automated. It still requires some post editing work by a $\mathcal{TDL}$user, which can vary from a couple of hours to one working day, in the worst case.

Macros

Since ALEP2$\mathcal{TDL}$ is conceived as a program providing as an output grammars which are fully understandable by a grammar writer, ALEP macros should be, whenever possible, translated into $\mathcal{TDL}$templates. In standard cases this operation is far from being difficult, as ALEP macros and $\mathcal{TDL}$templates resemble in many respects. There are a couple of cases, however, when this operation may be particularly difficult:

- Variables of $\mathcal{TDL}$templates always have scope within the template itself and they cannot be accessed from outside the template. Variables used in ALEP macros, on the contrary have as a scope a whole linguistic description (either a rule or a lexical entry). This means that certain values can be *structure shared* among macros in ALEP , while they cannot be among $\mathcal{TDL}$templates. To overcome these different behaviours required some modification to the interpretation of $\mathcal{TDL}$templates.

- ALEP macros have the possibility of being expanded directly into a feature structure. In other words, they are allowed to represents set of attribute-value pairs which are simply added to a standard features structure description containing other attribute-value pairs. Such a possibility is excluded by the interpretation of $\mathcal{TDL}$templates, and it is not easy to see how they could be implemented without radically changing the $\mathcal{TDL}$compilation program. Thus ALEP macros which exploit this function need to be manually "rebuilt" within the generated $\mathcal{TDL}$descriptions[5].

Type Expansion

$\mathcal{TDL}$has a strategy of type expansion whose primary goal is to make all the constraints associated to a certain type explicit. In normal cases a standard strategy of global type expansion (`expand-all-types`) can be applied in order to obtain a fully constrained grammar. Our tests based on a couple of LS-GRAM grammars proved that such a strategy is not always applicable to converted ALEP grammars, as in most cases this either produces a recursive (non terminating) type expansion, or takes an unreasonable time to produce full fledged types and instances. Thus more sophisticated strategies of type expansion need to be set up by the $\mathcal{TDL}$user in order to obtain a properly constrained linguistic organization.

Notice that such a controlled type expansion would be needed anyway in order to make the resulting $\mathcal{TDL}$grammar reasonably efficient. Indeed, input ALEP grammars, as they come, for instance, from the L-SGRAM project, have been highly engineered. Their efficiency has been highly improved by exploiting a set of "peripheral" settings of the ALEP framework, most importantly head declarations over rules. Such a strategy is unavailable in Page, thus the grammar which is obtained as the output of the translation process is not engineered. As noted in [Uszkoreit, 1991], however, great improvements in terms of efficiency can be achieved, in a $\mathcal{TDL}$like formal language, by controlling type expansion in such a way that only information relevant for filtering is expanded at compile time, while the remaining types are expanded (if necessary) in successive phases (delayed type expansion). Such a strategy of controlled type expansion cannot be deduced from the input ALEP grammar, and it requires an expert $\mathcal{TDL}$user to be properly set.

7.2.4 Architectural Issues

There are a couple of issues which are worth being mentioned in this section, even though they concern more the relationships between the architecture of a standard LS-GRAM grammar and the architecture of a standard $\mathcal{TDL}$grammar (such as the one used in ERGO, VERMOBIL, ...).

The first issue concerns morphology and the morphology-syntax interface. The ALEP2$\mathcal{TDL}$ program is absolutely unable to translate the two-level morphology rules usually forming the morphological component of an ALEP grammar. Consequently, also the translation of word grammar rules (even though perfectly feasible) is useless. The missing translation of the morphological module does not constitute a crucial issue in grammar portability, neither from a theoretical nor from a practical point of view. On one hand, the morphological component of ALEP is written in a completely different formalism, thus the ALEP2$\mathcal{TDL}$ program still constitutes a proof of the portability of unification based linguistic generalization among different formalisms. On the other hand, it could be argued that the morphological module is both conceptually and practically independent from the syntactic semantic module and has a simpler interface structure than the one provided by a two level morphology. For instance the Italian $\mathcal{TDL}$grammar has been interfaced with a MORPHIX based Italian morphology whose coverage, both in terms of lexicon and rules, is much wider than the one of the original ALEP grammar.

The second issues of architectural portability concerns the use of *refinement* grammars, which became almost a standard in the LS-GRAM project. Refinement is a non structure building level of the ALEP processing flow where the structures produced by the analysis component are "decorated" in a monotonic way by unification based rules (and lexical entries) mirroring the one

[5] Actually, a more sophisticated ALEP2$\mathcal{TDL}$ could overcome this problem simply by "exploding" the macros which cannot be translated during the generation phase. This capability should be included in the next release of the system.

of analysis.[6]. In most cases the refinement level is used to build "fancier" semantic representation, to explode the ambiguities of words, and to encode certain lexical generalizations. The reasons why these restrictions have to be delayed to this further level are not conceptual, but practical: the efficiency of the system really increases if certain computations can be performed at a non structure building level. Now, in spite of the fact that refinement grammars can be fully translated into Page grammars by ALEP2$\mathcal{TDL}$, there is no $\mathcal{TDL}$built-in procedure able to implement the behaviour of such a non structure building level. As a consequence, at the state of the art of ALEP2$\mathcal{TDL}$ analysis and refinement grammars are translated as two independent, non interacting modules. Actually, it is not a major problem to modify the behaviour of Page in such a way that it is able to accept such a layered organization of the grammar. Nor it is a major problem to build a merging program able to unify $\mathcal{TDL}$analysis and $\mathcal{TDL}$refinement grammar, in such a way that refinement linguistic descriptions can be seen as types whose evaluation is delayed. The choice among these two strategies will depend only on efficiency issues.

[6]Rules and lexical entries of refinement can be non distinct from the one of analysis.

7.3 HPSG

7.3.1 The EAGLES Encoding Format for HPSG

Introduction

The recent years have seen a growth in both the number of implemented HPSG grammars and the platforms that such grammars have been implemented upon. Although this is a very healthy trend from the perspective of promoting HPSG as a computationally viable grammatical theory, it also poses problems regarding the reusability of HPSG grammars. The specific problem of reusability for HPSG grammars is that grammars developed on one platform cannot be easily run on another platform. The second problem of reusability is that both the HPSG theory and implementation platforms are subject to improvements and changes. This means that grammars written sometime ago become obsolete and have to be recoded.

Apart from the reusability issue, there is also a serious need that linguists be able to test their grammars with as little programming effort as possible and that sample grammars found in texts can be tested and validated with minimum effort.

The EAGLES HPSG encoding format is an attempt to fulfill these needs by coming up with a declarative, extensible and yet efficiently implementable encoding format for HPSG grammars. The following goals were central in the design of the current draft standard:

1. it should be sufficiently expressive to state most if not all of the HPSG type hierarchy, principles, schemas and lexical rules as stated in HPSG-II [Pollard and Sag, 1994b]. Hence, the formalism should permit the development of grammars of significant size and complexity.

2. it should not be tied or dependent in any way to an implementation language - Prolog, LISP or any (typed) feature formalism.

3. it should be restrictive enough to allow fully automatic compilation into several target typed feature formalisms which differ significantly from each other.

Goal (1) is the primary requirement of the encoding standard. Goal (2) is necessary to ensure portability across language platforms and to provide independence from specific implementation languages and can be thought as the reusability requirement. Goal (3) is a practical requirement to ensure the very survival of the standard and to demonstrate the implementability of the standard with existing technology.

The above goals conflict with each other. For instance, the requirement for high expressivity imposed by (1) conflicts with the requirement of translatability into current generation target feature formalisms as imposed by (3). HPSG for instance employs a range of descriptive machinery such as set-valued descriptions and linear precedence constraints which are unavailable in formalisms such as ALE [Carpenter and Penn, 1994], CUF [Dörre and Dorna, 1993] and ALEP [Meylemans, 1994]. For this reason, the aim of the current draft is to define a core standard that can be compiled into most of existing typed feature formalisms. The aim is then to incorporate more sophisticated aspects of HPSG in the near future. Of course, backwards compatibility will have to be maintained. The idea is to allow the standard to evolve in step with developments in HPSG and in implementation technologies - as should be the case with any practical standard.

For a grammar standard to have an impact there is need for enabling software to assist in the migration from the current development environment to the one in which the current grammar standard is employed. There are two issues that need to be addressed:

1. software needs to be developed to enable grammars written in the proposed standard to be automatically compiled into existing feature formalisms - the concrete realisation of goal (3) above.

2. porting of existing grammars into the standard format should be possible with minimal effort.

Overview of the draft standard

HPSG as a grammatical theory is composed of several components that are needed to specify a grammar of any significant size. These different components of HPSG are:

1. type hierarchy and appropriateness conditions
2. schemas
3. principles
4. lexicon
5. lexical rules

These distinctions are primarily motivated by the needs of the grammar writer. However, this distinction may not necessarily be preserved when a grammar is encoded within an feature formalism. Thus for instance, within a sufficiently expressive feature formalism the principles and the schemas can be stated as constraints on the types. This situation is analogous to the case when grammar specifications are encoded in a very expressive logic wherein all HPSG specifications are encoded as a set of axioms.

The approach taken within the EAGLES proposal is to keep grammar specifications as close to the linguists' notion as possible by maintaining the above distinctions in the grammar specification, while leaving open the manner in which such specifications are encoded within feature formalisms. At the same time, attention has been paid in making sure that conventional parsing techniques can still be employed (if desired) and efficiency is not lost. This is essential in order to make grammar specifications usable in a real environment.

The draft standard divides the HPSG specification language into several sub-specification languages. These consist of:

1. type specification
2. feature description
3. relational language
4. schema specification
5. principle specification
6. lexicon specification
7. lexical rule specification
8. morphophonological rule specification

We provide a brief overview of each of the specification sub-languages, providing sample encodings where appropriate.

Type system specification language

The primary concerns in coming up with a suitable type specification language for HPSG grammars were as follows:

1. choice of semantics for multiple inheritance
2. choice of semantics for appropriateness conditions
3. expressivity of the type system *w.r.t.*
    - defining HPSG non-lexical signs

- defining hierarchical lexicons

4. compact minimal syntax

First of all, there are at least 3 different possibilities (and their combinations) in choosing a suitable multiple-inheritance semantics namely:

1. *open world*
   The *open world* interpretation of multiple inheritance is provided in systems such as ProFIT (mutidimensional inheritance) [Erbach, 1995], TDL (open-world mode) [Krieger and Schäfer, 1994] and CUF (subtype spec.) [Dörre and Dorna, 1993]. In essence, an open-world interpretation permits unification of any two types s and t as long as their conjunction $s\&t$ is consistent. It is the least restrictive one. One important advantage of the open-world interpretation is that it permits cross-classification of lexical types which is very important within a hierarchical lexicon.

   The big disadvantage of the purely open-world interpretation is that specifying HPSG non-lexical types becomes cumbersome since the grammar writer has then to specify, for instance, that the type *synsem* is incompatible with *loc* by employing a disjointness statement such as *disjoint(loc,synsem)*. This makes the grammar harder to read. Hence, the open-world interpretation alone is insufficient.

2. *closed world*
   The *closed world* interpretation is provided in systems such as TROLL [Gerdemann and Götz, 1994], TFS [Zajac, 1992b] and CUF (disjunctive spec.). In a closed world interpretation, given the subtypes $[t_1, t_2, t_3]$ of a type s; the type s is interpreted as being equivalent to the disjunction $t_1 \vee t_2 \vee t_3$ *i.e.* $s = t_1 \vee t_2 \vee t_3$.

   One big advantage of a closed-world reasoning interpretation is *strong* type checking in that an underspecified description of type t must be consistent with at least one of its subtypes t_i.

   The disadvantage is that disjunctive expansion is computationally expensive and clever schemes need to be devised to make the process efficient.

3. *greatest lower bound*
   In systems such as LOGIN [Ait-Kaci and Nasr, 1986a] and ALE [Carpenter and Penn, 1994] the inheritance hierarchy is required to be a semi-lattice. The conjunction $s\&t$ of types s and t is then interpreted as being equivalent to the *greatest lower bound* (*i.e. glb(s,t)*) of s and t. Thus in this approach the structure of the type hierarchy itself decides the mutual compatibility of types. Apart from this additional constraint, the *greatest lower bound* interpretation is similar to the *open world* interpretation.

   One advantage of the *greatest lower bound* interpretation in comparison to the *open world* interpretation is that incompatibility of types need not be stated explicitly thereby preserving compactness and readability of the grammar. The disadvantage being that this interpretation becomes quite cumbersome for stating hierarchical lexicons since it requires *unique glb*s.

   The advantage of the *greatest lower bound* and the *open world* interpretation in comparison to the *closed world* interpretation is that they both permit lazy evaluation since implementations do not need to deal with disjunctive expansion. This can also be considered a disadvantage since it means "weaker" type checking.

The greatest lower bound interpretation for multiple inheritance was chosen as a sufficient and minimally expressive mechanism for stating HPSG non-lexical signs. For stating hierarchical lexicons, in addition an open-world interpretation is highly desirable. A combined system that provides both the the glb-interpretation and the open-world interpretation is developed and chosen since it provides the best support for declaring HPSG signs and lexical hierarchies. One important point is that the *closed world* interpretation can be added on top (if needed in the future) without having to change the grammar.

Feature appropriateness constraints [Carpenter, 1992a] [Manandhar, 1993b] are modelled after the ALE formalism as it provides a simple but sufficient mechanism to model HPSG signs. Feature constraints can be attached to types. This is especially useful for specifying hierarchical lexicons. For space reasons, we do not discuss these here.

The parametric type $list(\alpha)$ is built-in to the standard allowing types such as $list(sign)$, $list(synsem)$, $list(quant)$ to be straightforwardly specified. Given below is a partial specification of the HPSG *sign* type.

```
  sign >   [word,non_word]
    approp [synsem:synsem,
            qstore:list(quant),
            qretr:list(quant)].

  cat approp [subcat:list(synsem),
              head:head,
              marking:marking].

   mod_synsem > [synsem,none].
    synsem approp [loc:loc,
                   non_loc:non_loc].

  loc approp [cat:cat,
              cont:sem_obj,
              conx:conx].

  conx approp [backgr:list(psoa),
               c_inds:c_inds].
```

Lexical hierarchies can be specified using feature constraints on types as illustrated in the example below adapted from the lexical hierarchy given in [Pollard and Sag, 1987] repeated in figure (7.3).

```
%% The HPSG hierarchical lexicon
unsat_non_nom >  [strict_intrans, intrans_control, trans].

strict_intrans def
  (synsem:loc:cat:(head:(mod:none &
                         aux:minus &
                         inv:minus) &
                   subj:[synsem] &
                   comps:[] &
                   spr:[] &
                   marking:unmarked)).

trans >  [strict_trans, ditrans, trans_control].

strict_trans def
  (synsem:loc:cat:(head:(mod:none &
                         aux:minus &
                         inv:minus) &
                   subj:[synsem] &
                   comps:[synsem] &
                   spr:[] &
                   marking:unmarked)).
```

However, crucially inheritance hierarchies such as the one shown in figure (7.3) do not form a lattice and hence the glb-semantics does not apply[7]. This is due to the fact that hierarchies such as the one in figure (7.3) employ *cross-classification*. This provides us the motivation for choosing a type hierarchy that combines the open-world and glb interpretations.

Combining open-world and glb interpretations

Our aim in this section is to design an inheritance system in which both the open-world and glb interpretations can be employed simultaneously. Furthermore, we wish to eliminate the need for stating disjointness conditions. The solution we adopt is rather simple:

1. use a declaration *open-world(t)* and
2. interpret every subtype of t using the open-world interpretation
3. interpret the rest using the glb mode
4. allow types in the open-world mode to inherit from types in the glb mode but **not** *vice versa*
5. the above effectively splits the inheritance hierarchy into two subparts:
   (a) a sublattice which is interpreted using the glb semantics
   (b) a poset which is interpreted using the open-world interpretation
6. interpret two types s and t in the open-world mode to be treated as disjoint only if they only have $\perp$ as the common-subtype.

Let *open-type* and *glb-type* be two functions defined by:

- *open-type*$(s) = true$
  $if \exists t : t \geq s$ and *open-world*(t) is declared
- *glb-type*$(s) = \neg$*open-type*(s)

The semantics for the above combined hierarchy can be readily stated by the following definitions.

1. $s \leq t \Longleftrightarrow [\![s]\!]^I \subseteq [\![t]\!]^I$
2. if *glb-type*(s) and *glb-type*(t) then
   $[\![s\&t]\!]^I = [\![glb(s,t)]\!]^I$
   and *glb-type*$glb(s,t)$
   else $[\![s\&t]\!]^I = [\![\perp]\!]^I$
3. $[\![s\&t]\!]^I = \emptyset$
   if $s \geq u$ and $t \geq u$ implies $u = \perp$

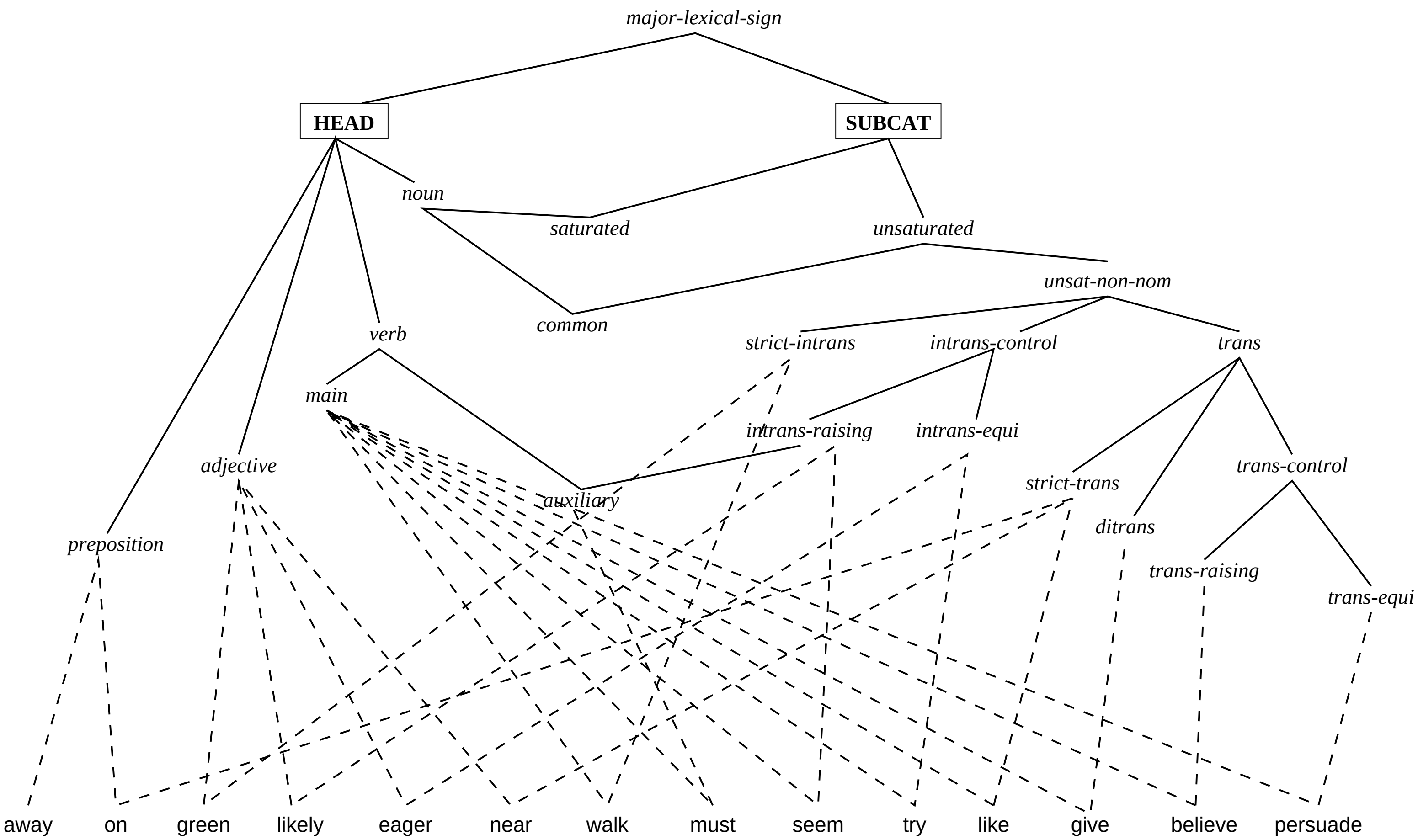
(1) states the interpretation of inheritance as a *subset* relation. (2) corresponds to a standard interpretation of the glb semantics within the glb sublattice. Its only (3) that is different. (3) states that the conjunction of two types s and t is only consistent if they have a common lower bound different from $\perp$. This means that two types s and t are deemed to be consistent only if the grammar writer declares another type which is a subtype of both s and t. Note that (3) applies to both open-types and closed-types. Furthermore definition (2) implies (3).

We think that the above approach for integrating the open-world semantics is the right approach given that *a.*) it avoids the need for disjointness specifications and *b.*) it conforms to the manner in which lexical cross-classifications are stated by the grammar writer.

[7]It is possible to compile a hierarchy such as the one in figure (7.3) into a lattice. This amounts to creating a new type for every possible conjunction of types. A given implementation can choose to do so.

Figure 7.3:

Cross-classification in a lexical hierarchy



(From [Pollard and Sag,87] pp. 206)

With the inclusion of a combined (glb+open-world) inheritance hierarchy we can easily state lexical cross-classifications such as the one from [Pollard and Sag, 1987] shown in figure (7.3). For the lexical hierarchy depicted in figure (7.3) all that needs to be done is to declare *major-lexical-sign* as an open-type by a declaration:

- *open_world*(*major_lexical_type*).

This declaration would then imply that all the subtypes of *major_lexical_type* are interpreted using the open-world semantics.

Schema specification language

The idea here is to state HPSG schemas close to the HPSG book notation but yet ensure that efficient compilation into PS-rules or relational dependencies is possible. The compilation program will then do the appropriate translation to the target feature formalism. For ALEP and ALE the compilation program should translate HPSG schemas into PS-rules. For CUF, the compilation program could translate them into relational constraints. Even TAG compilation such as [Kasper *et al.*, 1995] could be used without having to change the grammar. Hence, this will give us independence from the actual implementation strategy.

The underlying property of HPSG that makes a compilation to PS-rules a win at all is a principle of locality that all schemata and principles tacitly follow: schemata and principles of a grammar may only relate the elements of a local tree. This assumption amounts to requiring that no schema or principle may address a daughter's DTRS feature.

We decided to anchor this important property for efficient parsing fully in the schema format by (allowing the possibility of) separating the value of the DTRS feature from a *sign i.e.* in the current standard the DTRS feature need not be an appropriate feature for the type *sign*.

For instance, head-complement schema (for English) can be stated as follows:

```
schema :: head_comp( syn:loc:head:subj:[_|_],
                     head_dtr:sign &
                     comp_dtrs:list(sign),
                    [head_dtr < comp_dtrs]
                    ).
```

Note that the schema definition contains 3 parts namely:

1. the specification of the 'MOTHER' sign as the first argument of head_comp
2. the specification of the "daughters" sign as the second argument of head_comp
3. the specification of LP constraints as the third argument of head_comp

As we shall illustrate below, this seperation permits an implementation to compile out schema definitions into PS-rules enabling use of existing efficient parsing techniques.

The types that are appropriate for the "daughters" sign can be organised within an inheritance hierarchy as shown in figure (7.4). This means that appropriateness conditions and consistency checking apply to the "daughters" sign. This allows the "daughters" sign to be specified in a hierarchical fashion. For instance, the head_comp schema can be restated as given below:

```
headed_struct >   [head_comp_struct]
           approp [head_dtr:sign].

head_comp_struct >  []
             approp [comp_dtrs: list(sign)].

schema :: head_comp( syn:loc:head:subj:[_|_],
                     head_comp_struct,
                    [head_dtr < comp_dtrs]
                    ).
```

```
%                                    HPSG Schemas
% ============================================================================
%                                     headed_struct
%                                          |
%                      ______________________|___________________________
%                     |              |                 |                  |
%             f_struct  head_mark_struct  head_adjn_struct  head_filler_struct
%                     |
%                 ____|______________________________
%                |                  |                |
%         subj_struct         comp_struct   head_spec_struct
%                |                  |
%        _______|__________     ____|__________
%       |                  |   |               |
% head_subj_struct head_subj_comp_struct  head_comp_struct
%
headed_struct >   [f_struct,
                   head_mark_struct,head_adjn_struct,head_filler_struct]
           approp [head_dtr:sign].

   f_struct  >  [subj_struct,
                 comp_struct,
                 head_spec_struct].

       subj_struct >    [head_subj_struct,head_subj_comp_struct]
                 approp [subj_dtr:sign].

       comp_struct >    [head_subj_comp_struct,head_comp_struct].
                 approp [comp_dtrs:list(sign)].

       head_spec_struct approp [spec_dtr:sign].

   head_mark_struct approp [mark_dtr:sign].
   head_adjn_struct approp [adjn_dtr:sign].
   head_filler_struct approp [filler_dtr:sign].
```

Figure 7.4: The EAGLES specification of HPSG schema types

```
schema :: head_subj(sign,
                    head_subj_struct,
                    [subj_dtr,head_dtr]
                   ).
schema :: head_comp(sign,
                    head_comp_struct,
                    [head_dtr,comp_dtrs]
                   ).
schema :: head_subj_comp(sign,
                         head_subj_comp_struct,
                         [head_dtr,subj_dtr,comp_dtrs]
                         ).
schema :: head_spec(sign,
                    head_spec_struct,
                    [spec_dtr,head_dtr]
                    ).
schema :: head_adjn(sign,
                    head_adjn_struct,
                    [head_dtr,adjn_dtr] %% Only allow adjunct to follow head
                    ).
schema :: head_filler(sign,
                      head_filler_struct,
                      [filler_dtr,head_dtr]
                     ).
```

Figure 7.5: The EAGLES specification of HPSG schemas

Our definition of the head_comp schema as shown above declares that a *sign* can be obtained from a local tree consisting of an object of type *sign* (which is the value appropriate to `head_dtr` feature) and another object of type *list(sign)* (which is the value appropriate to `comp_dtr` feature).

One important consideration during the design of the schema specification language was that compilation of the schema declarations into PS-rules was possible. Consider, for instance the head_comp schema given above. This can be compiled into the following PS-schema:

$sign$	$->$	$sign,$	$list(sign)$
$'\mathrm{MOTHER}'$		head_dtr	comp_dtrs

In this particular case, a left-to-right rule instantiation would instantiate the comp_dtrs list (cf. ALE). In the general case when the position of the head is not fixed a standard head-driven approaches can be employed.

The complete specification of HPSG schemas is provided in fig. 7.5.

Rudimentary support for free word-order is also provided. The notation `[head_dtr < comp_dtrs]` means that the head_dtr "precedes" the comp_dtrs list. On the other hand, the notation `[head_dtr,comp_dtrs]` means that the head_dtr "immediately precedes" the comp_dtrs. Given a local tree consisting of just the head_dtr and the comp_dtrs the two specifications are equivalent.

The encoding format currently does not provide for mechanisms involving complex word order such as using the domain union mechanism [Reape, 1993] or LP constraints [Manandhar, 1995].

The LP mechanism is restricted to local trees only. Partially specified local trees compiles out into multiple PS-rules. LP constraints can be employed to state the relative ordering of pairs of *DTR* features and a *DTR* feature can be declared as `initial` or `final`. Thus the head_subj schema can also be equivalently stated as follows (although very unintuitive!):

```
schema :: head_subj(synsem:loc:head:subj:[],
```

```
                    ( head_subj_struct &
                      subj_dtr:synsem:SubjSyn  &
                      head_dtr:synsem:loc:head:subj:[SubjSyn]
                    ),
                   [initial(subj_dtr), final(head_dtr)]
                  ).
```

All the different types of LP constraints can be freely mixed.

HPSG principles encoding format

The principles encoding format allows HPSG principles to be stated independently of the schema encoding format. This is achieved by employing a principle interface specification that defines the principles that are applicable to a given schema. For instance, the head-feature principle, the marking principle and the subject condition would be stated as follows:

```
principles::
 [ head_feature_principle('MOTHER',head_dtr),

   ~head_mark_struct ->  marking_principle('MOTHER',head_dtr),
   marking_principle('MOTHER',mark_dtr)

   subject_condition(subj_dtr,comp_dtrs)
 ].
```

The specification of the head-feature principle specifies that it applies to every schema which contains a head_dtr. The `'MOTHER'` argument which by definition is present in every schema specification is passed as the first argument of head_feature_principle. Similarly, the subject_condition applies to every schema which contains a subj_dtr and a comp_dtrs. The marking_principle on the other hand, applies to every schema containing a head_dtr but excluding the head_mark_struct. A more detailed specification of the interface to principles is given in fig. (7.6).

These specifications only specify the interface between the schemas and the actual definition of the principles. The definition of the principles themselves is done using a simple relational language (in effect, restricted Prolog with an important extension). The relational language is discussed below.

Relational Language

The relational language is defined to be a pure Horn language (i.e. no cuts (!s) and no negation by failure) without any recursion. However, it provides a host of library routines to compensate for this simplicity. These involve guarded constraints, list manipulation, maplist predicate, list-based set operations etc. This restricted language guarantees termination. The library routines hide low-level operations and allow the grammars to be lot more abstract and portable across different programming language environments.

Guarded constraints provide the ideal mechanism to abstract over the manner in which *conditional* HPSG descriptions are encoded within current implementations. Consider the definition of the subject_condition predicate given below:

```
subject_condition(@inher_slash(SubjSlash),CompDtrs):-
    case(SubjSlash,
             [nelist => set_member(nelist, @accumulate(CompDtrs, inher_slash))]).
```

This specification highlights several important aspects of the relational language.

Firstly, the relational language permits functional annotation (alternatively a macro annotation) of a predicate call. The predicate inher_slash/2 would be defined as follows:

```
inher_slash(Slash, synsem:non_loc:slash:Slash).
```

```
principles ::
[ head_subj_struct -> valence_principle_subj('MOTHER',head_dtr,[subj_dtr]),
  head_comp_struct -> valence_principle_comps('MOTHER',head_dtr,comp_dtrs),
  valence_principle_subj_comps('MOTHER',head_dtr,[subj_dtr|comp_dtrs]),
  valence_principle_spec('MOTHER',head_dtr,[spec_dtr]),

  ~head_mark_struct ->  marking_principle('MOTHER',head_dtr),
  marking_principle('MOTHER',mark_dtr),

  spec_principle(subj_dtr,head_dtr),  % applies to both head_subj_struct,
                                      % head_subj_comp_struct
  head_comp_struct ->  spec_principle2(comp_dtrs,head_dtr),
  spec_principle(mark_dtr,head_dtr),
  % spec_principle does not apply to the rest of the schemas

  head_subj_struct ->  semantics_principle('MOTHER',head_dtr,[subj_dtr]),
  head_comp_struct ->  semantics_principle('MOTHER',head_dtr,comp_dtrs),
  semantics_principle('MOTHER',head_dtr,[subj_dtr|comp_dtrs]),
  semantics_principle('MOTHER',head_dtr,[mark_dtr]),

  subject_condition(subj_dtr,comp_dtrs),     % applies to head_subj_comp schema
  head_comp_struct  -> subject_condition(synsem:@hd(@subj('MOTHER')),comp_dtrs),

  head_subj_struct  -> nonlocal_feature_principle('MOTHER',head_dtr,[subj_dtr]),
  head_comp_struct  -> nonlocal_feature_principle('MOTHER',head_dtr,comp_dtrs),
  nonlocal_feature_principle('MOTHER',head_dtr,[subj_dtr|comp_dtrs]),
  nonlocal_feature_principle('MOTHER',head_dtr,[mark_dtr])
 ].
```

Figure 7.6: The EAGLES specification of HPSG principles' interface

```
% valence_principle_xxx(Mother,Head-Daughter,Functional-Daughters)
%---------------------------------------------------------------
valence_principle_subj(@subj(MSub) & @comps(C) & @spr(Spr),
                       @subj(@append(MSub,FDtrSyns)) & @comps(C) & @spr(Spr),
                       @synsems(FDtrSyns)).

valence_principle_comps(@comps([]) & @subj(S) & @spr(Spr),
                       @comps(@append(MSub,FDtrSyns)) & @subj(S) & @spr(Spr),
                       @synsems(FDtrSyns)).

valence_principle_subj_comps(@comps([]) & @subj([]) & @spr(Spr),
                       @comps(CompDtrSyns) & @subj([SubjSynsem]) & @spr(Spr),
                       @synsems([SubjSynsem|CompDtrSyns])).

valence_principle_spr(@spr(MSub) & @comps(C) & @subj(Subj),
                      @spr(@append(MSub,FDtrSyns)) & @comps(C) & @subj(Subj),
                      @synsems(FDtrSyns)).

% semantics_principle(Mother,Semantic-Head,Other-Dtrs)
%---------------------------------------------------------
% Semantic-Head is adjunct-dtr otherwise head-dtr
semantics_principle(@qstore(MQStore) & @qretr(MRetr) & @cont(MCont),
                    SHead & @cont(SCont), ODtrs) :-
  accumulate([SHead|ODtrs], qstore, DQStore),
  case(SCont, [(psoa, ( MCont = ( @nucleus(Nucl) & @quants(MQuants) ),
                        SCont = ( @nucleus(Nucl) & @quants(SQuants) ),
                        set_difference(MQStore,DQStore,MRetr),
                        append(MRetr,SQuants,MQuants)
                       ))
               ],
                   SCont = MCont,
                   MRetr = [],
                   MQStore = DQStore
        ).
% nonlocal_feature_principle(Mother,Head-Dtr,Other-Dtrs)
%---------------------------------------------------------------
nonlocal_feature_principle(@inher_slash(MISlash) &
                           @inher_que(MIQue) & @inher_rel(MIRel),
                           HeadDtr & @to_bind_slash(HTSlash) &
                                     @to_bind_que(HTQue) &
                                     @to_bind_rel(HTRel),
                           ODtrs):-
  accumulate([HeadDtr|ODtrs], inher_slash, DISlash),
  accumulate([HeadDtr|ODtrs], inher_que, DIQue),
  accumulate([HeadDtr|ODtrs], inher_rel, DIRel),
  set_difference(HTSlash,DISlash,MISlash),
  set_difference(HTQue,DIQue,MIQue),
  set_difference(HTRel,DIRel,MIRel).
```

Figure 7.7: The EAGLES specification of HPSG principles

The functional annotation permits the last argument of the a predicate call to be implicitly bound to the context. Thus a functional annotation `@p(A1, ... , An)` would require a predicate definition for `p/n+1` to exist.

The above definition of the subject_condition is equivalent to the following transformed definition where all the functional annotations have been eliminated:

```
subject_condition(Subj,CompDtrs):-
    inher_slash(SubjSlash, Subj),
    accumulate(CompDtrs, inher_slash, Slashes),
    case(SubjSlash,
             [nelist => set_member(nelist, Slashes)]).
```

Secondly, the library routines built-in to the relational language includes list iteration predicates such as accumulate/3 which in the above call extracts the inherited-slash values of the CompDtrs.

A summary of library predicates is provided below:

1. List manipulation routines - *append/3, select/3, reverse/2.*

2. Maplist routines for iterating over list structures - *maplist/3, accumulate/3.*

3. List based set operations - *set_member/2, set_difference/3, set_union/3, set_subset/2, set_disjoint/2.*

4. Guarded constraints - *case/2, case/3.*

5. FST operations - *replace/3*

The primary aim of including the library predicates is to shield the grammar writer from having to write implementation language dependent low-level routines. At the same time, it enables the relational language itself to be kept simple and small since the more complex machinery is hidden inside the library routines.

This approach constrasts with an alternative solution which would include the full Prolog machinery so that all the library routines can be user defined. But clearly grammars written in this more expressive language would be harder to read and would require a Prolog interpreter/compiler. The library routines are expected to grow when the needs of the grammar writer are not met by the available set of routines.

Our current solution opens the possibility of efficient implementability in other languages and at the same time makes it possible to write compact highly readable grammars.

Thirdly, the relational language provides guarded constraints as a declarative mechanism for specifying HPSG conditional descriptions such as the ones embodied in the subject_condition.

Guarded constraints provide support for *if-then-else* style programming in a declarative setting. Given a constraint of the form C & $if\ G\ then\ B\ else\ E$; C is known as the *context*, G is known as the *guard*; B, E are *actions*. The action B is added to the context whenever the context C *entails* the guard G. On the other hand, the action E is added to the context whenever C disentails G. Following Saraswat's work [Saraswat, 1993] guarded constraints forms the basis of concurrent constraint programming languages such as Oz [Smolka *et al.*, 1995].

Guarded constraints are specified using the case/2 and case/3 predicates which is a syntactic sugaring of the *if-then-else* style syntax. A description of the form `case(C, [ G1 => B1, G1 => B2 ], E)` is equivalent to:

$$\begin{aligned} C\ \& \quad & if\ G1\ then\ B1\ else\ F1 = t\ \& \\ & if\ G2\ then\ B2\ else\ F2 = t\ \& \\ & if\ (F1 = t\ \&\ F2 = t)\ then\ E \end{aligned}$$

The replace operation implements a *finite state transducer* for string manipulation. This is decribed separately in section 7.3.1.

Lexical entry format

Lexical entries are simply identified by a distinguished hook predicate `lex/1`. The purpose of the hook predicate is just to mark feature descriptions that count as lexical entries. The actual lexical entry may be specified via a hierarchical lexicon and/or in an external lexicon database. The definition of the hook predicate can then provide the link to the actual lexical entries. The following example illustrates the use of the `lex/1` hook:

```
% HPSG hierarchical lexicon
main > [trans_verb_sign, ...].

strict_trans > [trans_verb_sign, trans_prep_sign].

trans_verb_sign def
  (synsem:loc:cat:(head:(mod:none &
                         aux:minus &
                         inv:minus) &
                   subj:[synsem] &
                   comps:[synsem] &
                   spr:[] &
                   marking:unmarked )).

% Interface for Transitive Verb lexical entries
% ----------------------------------------------------------------------------

lex(X) :-
    trans_verb_lex(X).

trans_verb_lex(@trans_verb_sign @
                @ortho(O) & @vform(VForm) &
                @verb_np_subj(Subj) &
                @verb_obj1(Obj1) &
                @verb_sem(Nuc)) :-
   trans_verb(O,VForm,Subj,Obj1,Nuc).

% Actual lexical entry
% --------------------

trans_verb(like, bse, top, (I1, top, top, top, top),
                          (I2, acc), liker:I1 & likee:I2).

%% The macros for lexical entries
verb_np_subj((Ind, Case, Num, Per, Gen),
              @subj([@np(Ind) & @case(Case) &
                     @num(Num) & @per(Per) & @gen(Gen)])).

verb_np_obj1((Ind,Case),synsem:loc:cat:comps:[@index(Ind) & @case(Case)|top]).

verb_obj1(Obj,synsem:loc:cat:comps:[Obj | top]).
```

Lexical rules encoding format

Lexical rules are perhaps the least studied amongst the different components of HPSG and at least their form and interpretation most suitable for HPSG is still subject to some debate. On the other hand, lexical rules are playing an increasingly important role within HPSG since linguistic phenomena which were handled by other components of HPSG in the past (*cf. unbounded dependency, adjunct extraction, subject-auxiliary inversion etc.*) are being handled using the mechanism of lexical rules.

The issues that are relevant for an encoding format on lexical rules can be summarised as follows:

1. in order to maximise reusability, it should be possible to write lexical rules largely independently of the feature geometry employed.

2. in order to ease writing of lexical rules lexical rules should only contain information that pertain to changes in an existing lexical entry to create a new one. All other feature-structural information is automatically co-indexed between the input lexical entry and the output lexical entry of a lexical rule.

To achieve objective (1) it is recommended that lexical rules do not contain explicit references to feature names instead these are to be referenced indirectly by employing macro calls.

To achieve objective (2) we largely follow the approach developed in [Meurers and Minnen, 1995] in which lexical rules only mention features that are subject to change by the lexical rule and the values of other features are automatically assumed to be identical (modulo *appropriateness conditions*) between the input and the output of the lexical rule. This approach can also be seen directly in terms of a new logical operation in feature logic dubbed the *update operation*. Under this view lexical rules are update operations that update the input feature description with the output feature description keeping the values of the rest of the features identical between the input and the output. Interestingly, such an operation can be given a purely monotonic characterisation. The update operation is given a thorough treatment elsewhere in this document.

As a concrete illustration of a lexical rule that conforms to the proposals made so far, consider the lexical rule for generating the *3rd singular* verb form from the *base* form.

```
lex_rule::
  finite_3sg(@ortho(O)  & @vform(bse) &
             @aux(minus) & @subj_cat(np)
             ,
             @ortho(NewO) & @vform(fin) &
             @aux(minus)  & @subj_cat(np) & @subj_case(nom)
            ):-
            first_of([replace(O, top - [y]:[i,e,s],NewO),
                        replace(O, top - [e]:[e,s],NewO)
                      ]).
```

Lexical rule application unifies the input of the lexical rule with an existing lexical entry. All feature paths that are not specified in the output of the lexical rule are automatically co-indexed with the corresponding feature path of the input.

The `replace` operation is part of the morphophonological encoding format and is described in the following section.

One interesting aspect of the lexical rule notation employed is that the lexical rule can be employed bi-directionally if so desired. However, we do not stipulate on this possibility in this report[8].

The encoding of morphophonological rules

Morphophonological rules in the proposed encoding format can be specified using two related mechanisms. Firstly, for string manipulation, we propose to employ bi-directional finite-state-transducers (FSTs) which are interpreted as bi-directional guards. These are specified using the replace operator as in the previous example shown below (in a simplified manner):

[8] In theory at least there seems to be no *a priori* reason to require that lexical rules apply only uni-directionally. It is conceivable to think of an alternative lexical rule mechanism wherein the lexicon is construed as the relational closure over the lexical rules from some base lexicon set.

```
test1(O,New):-
   O = [t,r,y],
   replace(O, top - [y]:[i,e,s],NewO).
```

The notation `top` stands for the *id** identity transformation. Now, `top` gets matched with the string `[t,r]` and `[y]` gets replaced with `[i,e,s]`. This results in `NewO` getting instantiated as `[t,r,i,e,s]`.

Since the *replace* operation is bi-directional we can also apply it in reverse as shown below:

```
test2(O,New):-
   NewO = [t,r,i,e,s],
   replace(O, top - [y]:[i,e,s],NewO).
```

In an analogous manner `O` will get instantiated as `[t,r,y]`.

Finally, the `first_of` operator as shown below simply states that the first successful branch is to be chosen:

```
test3(O,New):-
   first_of([replace(O, top - [y]:[i,e,s],NewO),
             replace(O, top - [e]:[e,s],NewO)
            ]).
```

Thus in the case of the instantiation `O = [l,i,k,e]` the second branch (*i.e.* the second replace operation) is chosen.

Secondly, our idea is to simply extend our guarded constraint language to include *regular expressions*. This enables us to declaratively express co-constraints between orthography and syntax. Thus we permit conditionals of the form `case(Ortho,[ c*-v-c* => vform = fin ])` where `c` is a subtype of the built-in `constant` type (presumably standing for *consonant* in this example).

As a more comprehensive illustration of the morphophonological encoding format we illustrate the encoding of the morphology of the Arabic verb **kitab**. The example has been adapted from [Erjavec, 1994] but a related approach can be found in [Bird and Ellison, 1994].

First of all the verb **kitab** exibits the following alternations:

Ortho	**VForm**	**Voice**
katab	I	active
kutib	I	passive
kattab	II	active
kuttib	II	passive
kaatab	III	active
kuutib	III	passive
takaatab	VI	active
tukuutib	VI	passive
nkaatab	VII	active
nkuutib	VII	passive
ktatab	VIII	active
ktutib	VIII	passive
staktab	X	active
stuktib	X	active

In [Erjavec, 1994] it is shown that all the above forms can be derived by assuming the stem to be `(k,t,b)`. Secondly, [Erjavec, 1994] assumes the following vowel-consonant patterns for the **kitab** verb class.

$[c, v, c, v, c]$
$[c, v, c, c, v, c]$
$[c, v, v, c, v, c]$
$[t, v, c, v, v, c, v, c]$
$[n, c, v, v, c, v, c]$
$[c, t, v, c, v, c]$
$[s, t, v, c, c, v, c]$

Erjavec's analysis can be carried over in our setup with some modifications but by essentially retaining the insights as shown in the encoding given below. We have borrowed ideas from [Bird and Klein, 1994] on the use of templates for encoding morphophonological patterns.

```
constant > [v,c]. % vowels and consonants are contant types
  v > [a,i,u].
                  % Expansion of c not shown
verb_entry(k,t,b).

lex(ortho:Ortho &
    stem:[C1 & c,C2 & c,C3 & c] &
    vform:VForm &
    voice:Voice):-
  verb_entry(C1,C2,C3),
  case(VForm, [i   => Ortho = [C1, v, C2, v, C3],
               ii  => Ortho = [C1, v, C2, C2, v, C3],
               iii => Ortho = [C1, V & v, V, C2, v, C3],
               vi  => Ortho = [t, v, C1, V & v, V, C2, v, C3],
               vii => Ortho = [n, C1, V & v, V, C2, v, C3],
               viii=> Ortho = [C1, t, v, V, C2, v, C3]
              ])
       else
            (VForm = x, Ortho = [s, t, v, C1, C2, v, C3]),

  case(Voice,[active => replace(Ortho, ([v]:[a] ; [c]:[c])+ , Ortho)])
     else
          (Voice = passive,
           replace(Ortho, ([v]:[u] ; [c]:[c])+ - [i]:[i] - [c]:[c], Ortho).
```

In the above encoding the instantiation of the `vform` and the `voice` features triggers the instantiation of the orthography. Note that in the second `case` statement above, both the input and the output of the replace operation is the same variable `Ortho`. In the first case: `replace(Ortho, ([v]:[a] ; [c]:[c])+ , Ortho)])` specialises every occurrence of an underspecified vowel into `a`. In the second case: `replace(Ortho, ([v]:[u] ; [c]:[c])+ - [i]:[i] - [c]:[c], Ortho)` replaces every occurrence of an underspecified vowel into `u` in the initial segment.

We have been assuming so far that the lexical entries are generated off-line in which case the above example will require that all the possible values of `vform` and `voice` are iterated to generate all possible lexical entries.

However, we can easily accomodate dynamic lexical lookup by adding another guarded constraint as shown below:

```
   case(Ortho,[ [c, v, c, v, c] => VForm = i,
                [c, v, c, c, v] => VForm = ii,
                [c, v, v, c, v, c] => VForm = iii,
                [t, v, c, v, v, c, v, c] => VForm = vi,
                [n, c, v, v, c, v, c] => VForm = vii,
                [c, t, v, v, c, v, c] => VForm = viii)
       else
```

```
          VForm = x,

case(Ortho,[([a];[c])* => Voice = active])
     else
          Voice = passive
```

These conditionals will "fire" once the `Ortho` variable is instantiated and will cause the appropriate instantiation of the `VForm` variable and `Voice` variable. This in turn will cause the "firing" of the remaining guarded constraints that depend on the variable `VForm` and `Voice`. With this addition the application of the morphophological rules can be done dynamically at runtime.

Discussion

In this report we have described a declarative, language independent, high-level encoding format for HPSG grammars. We believe that the design of the encoding format ensures efficient implementability using existing techniques yet at the same time ensures that the grammars are highly readable and compact. The primary aim of this exercise is to address the re-usability issue of HPSG grammars; to ensure that grammars written in the encoding format have a high chance of being reused in a different software environment from the one that it was created. Thus, declarativity, language independence and efficient implementability were central concerns during the design of the encoding format.

Whether the current encoding format actually turns out to be useful in practice will depend on two factors. Firstly, we need to show that it is possible to make reasonably efficient implementations using the encoding format. Secondly, we need to show that grammars of significant size can be written using the encoding format. Each of these steps will inevitably lead to changes in the encoding format. However, these refinements are a necessary step for a useful encoding standard.

7.3.2 Concrete Syntax for EAGLES HPSG Encoding Format I

TYPE SPECIFICATION LANGUAGE

Syntax:

$$
\begin{array}{lll}
\textit{Type-Definition} \longrightarrow & & \textit{Subtype-Spec} \\
& \mid & \textit{Approp-Spec} \\
& \mid & \textit{Definition-Spec} \\
& \mid & \textit{Combined-Spec}
\end{array}
$$

Subtype-Spec $\longrightarrow$ *Type* > [$\textit{Type}_1, \ldots, \textit{Type}_n$]**.**

Approp-Spec $\longrightarrow$ *Type* **approp [** $\textit{feature}_1{:}\mathrm{Type}_1$,
...,
$\textit{feature}_n{:}\mathrm{Type}_n$**].**

Definition-Spec $\longrightarrow$ *Type* **def** *FD***.**

$$
\begin{array}{lll}
\textit{Combined-Spec} \longrightarrow & & \textit{Subtype-Spec} \\
& & \textit{Approp-Spec} \\
& & \textit{Definition-Spec.} \\
& \mid & \textit{Subtype-Spec} \\
& & \textit{Approp-Spec.} \\
& \mid & \textit{Subtype-Spec} \\
& & \textit{Definition-Spec.} \\
& \mid & \textit{Approp-Spec} \\
& & \textit{Definition-Spec.}
\end{array}
$$

feature $\longrightarrow$ *atom*

Type $\longrightarrow$ *atom*

Open-World-declaration $\longrightarrow$ **open_world(** *Type***).**

Predefined Types:
`top, fail, [], list(`α`), nelist(`α`), constant, boolean, plus, minus`

`list(`α`), nelist(`α`)` are predefined *parametric* type.

Predefined Inheritance Hierarchy:

```
top > [list(α), constant].
constant > [boolean].
boolean > [plus, minus].
list(α) > [ [], nelist(α) ].
```

Terminology

1. **immediate subtype**
   if `s` and `t` appear in a *Type-Definition*:
   `t` > [..., `s`, ...] ...
   then `s` is an immediate subtype of `t`. The purpose of the *immediate subtype* relation is purely for defining the *subtype* relation given below.

2. **subtype(s,t)**
   `t` is a "subtype of" `s` if there is a sequence of immediate subtypes of `s` - `s`,$\mathtt{s}_1$,...,$\mathtt{s}_n$,`t` - that lead to `t`. Furthermore for every type `t` `subtype(t,t)` is true.

3. **approp(s,f)**
   the feature `f` is "appropriate for" `s` if `f` appears in the *Type-Definition* of `s` *i.e.* there exists a *Type-Definition* of the form:
   ```
   s ... approp [ ...,
                  f:...,
                  ... ].
   ```

4. **range(f,s)**
   the "range type" of a feature `f` is `s` if `f:s` appears in the *Type-Definition* *i.e.* there exists a *Type-Definition* of the form:
   ```
   ... approp [ ...,
                f:s,
                ... ].
   ```

5. **s $\leadsto$ t**
   the type `s` "immediately leads to" the type `t` if there is exists a feature `f` and a *Type-Definition* of the form:
   ```
   s ... approp [ ...,
                  f:t,
                  ... ].
   ```

6. **s $\leadsto^*$ t**
   the type `s` "leads to" the type `t` if there is a sequence $\mathtt{s} \leadsto \mathtt{s}_0$, $\mathtt{s}_1 \leadsto \ldots$, $\ldots$, $\mathtt{s}_{n-1} \leadsto \mathtt{s}_n$, $\mathtt{s}_n \leadsto \mathtt{t}$.

7. **common subtype**
   the type `u` is "a common subtype of" the types `s` and `t` if both `subtype(s,u)` and `subtype(t,u)` are true.

8. **most general common subtype**
   the type `u` is "the most general common subtype" of the types `s` and `t` if for every common subtype `v` of `s` and `t` the relation `subtype(u,v)` is true.

9. **open-type(t)**
   the type `t` is an "open-type" if:
   `open-world(s).`
   is declared and `subtype(s,t)` is true.

10. **glb-type(t)**
    the type `t` is a "glb-type" if `open-type(t)` is false.

Predefined Appropriateness Conditions:

`nelist(`α`)  approp  [`α`|list(`α`)].`

Restrictions:

1. (>): for a given type t there may be multiple Subtype-Spec declarations:
   `t > [...].`
   These would be interpreted conjunctively.

2. (approp): for a given type t there is at most a single *Approp-Spec* declaration:
   `t approp [...].`

3. (definition): for any type t there may be multiple *Definition-Spec* declarations:
   `t approp [...].`
   These would be read conjunctively.

4. (top): if the condition `subtype(top,t)` cannot be inferred from existing definitions then `subtype(top,t)` is automatically assumed. In this case, the compiler is expected to issue a warning message indicating that the above subtyping relation has been automatically inferred.

5. (fail): for any type t `subtype(t, fail)` is automatically assumed.

6. (unique appropriateness): for distinct types s and t and a feature f if both approp(s,f) and approp(t,f) is true then:

   - *either* s is a *subtype* of t
   - *or* t is a *subtype* of s
   - *or* there is type u such that t and s are both subtypes of u.

7. (acyclicity): for any types s and t it is not the case that $s \leadsto^* t$ and s is more *general* than t.

8. (semi-lattice): for distinct types s and t such that both `glb-type(s)` and `glb-type(t)` is true:

   - *either* **fail** is the only common subtype of s and t.
   - *or* there is only a single common subtype of s and t (excluding fail).
   - *or* there is a single most general common subtype of s and t.

9. (constant): For the subtypes of the built-in type *constant* only subtyping definitions will be permitted *i.e.* only definitions of the form:
   `constant > [..., t, ...].`
   `t > [...].`

Comment:

1. The "glb part" of the type specification language follows the ALE model as far as expressivity and interpretation of multiple inheritance is concerned. There are several points of departure:

   (a) The specification permits multiple *subtyping* declarations (*Subtype-Spec* in the current terminology) to permit easier combination of grammar signatures from different files and different sources. This should improve modularity by allowing all the inheritance statements relevant for a subpart of a grammar specification to be isolated and put into a single file.

   (b) Subtyping specifications are not always required as in the ALE model thus reducing the size of the signature.

   (c) *Definition-Spec* corresponds to the `cons` definition in ALE but is restricted to feature descriptions as opposed to arbitrary goals in ALE. The *Definition-Spec* declaration has been introduced primarily to support the specification of the hierarchical lexicon. Arbitrary goal attachment has been disallowed to guarantee termination and a simpler computational model.

2. Since *cross-classification* of lexical types is highly desirable for a compact representation of the hierarchical lexicon the encoding format allows combining a *glb* interpretation of the inheritance hierarchy (*cf.* ALE) with an *open-world* interpretation (*cf.* TDL open-world mode) for some user selected set of types. The intention being that lexical types (or a certain selected subset of lexical types) would be interpreted using the open-world interpretation while the other types would be interpreted using the glb interpretation.

FEATURE DESCRIPTION LANGUAGE

Syntax:

$$\begin{array}{rl} FD \longrightarrow & \mathit{feature}{:}FD \\ & \mid \mathit{Type} \\ & \mid \mathit{Var} \\ & \mid FD_1 \ \& \ FD_2 \end{array}$$

Comment:
The feature description language has been kept minimal with the aim of achieving translatability into a wider range of existing typed feature formalisms. However, note that the definition permits *cyclic* feature descriptions.

Future extensions should support *finite domains* (*cf.* ProFIT), limited negation and limited disjunction.

SCHEMA SPECIFICATION LANGUAGE

Syntax:

schema:: *Schema-Name*(*Mother-FD*,*DTRS-FD*, *LocalLPConstraint*).

Remainder Syntax:

Schema-Name $\longrightarrow$ *atom*

Mother-FD $\longrightarrow$ *FD*

DTRS-FD $\longrightarrow$ *FD*

DTR $\longrightarrow$ *feature*

$$\mathit{LocalLPContraint} \longrightarrow \begin{array}{ll} & \mathit{FixedLocalTree} \\ | & \mathit{LPStatements} \end{array}$$

FixedLocalTree $\longrightarrow$ $[Dtr_1,\ldots,Dtr_n]$

LPStatements $\longrightarrow$ $[LP_1,\ldots,LP_k]$

$LP \longrightarrow DTR_1 < DTR_2$

$LP \longrightarrow$ **initial**(DTR)

$LP \longrightarrow$ **final**(DTR)

Restrictions:

1. (Local LP): If the LP condition $DTR_1 < DTR_2$ appears within a schema definition then both DTR_1 and DTR_2 must be appropriate for the *DTRS-FD* of the same schema definition. Thus, the following is ill-formed since the *DTR* feature `adj_dtr` is not appropriate for the *DTRS-FD*:

```
head_comp_struct approp [head_dtr:sign,
                         comp_dtrs:list(sign)].

schema:: head_comp( syn:loc:head:subj:[_|_],
                    head_comp_struct,
                    [ adj_dtr < head_dtr ]
                   ).
```

A correct description of the `head_comp` schema would be as follows:

```
head_comp_struct approp [head_dtr:sign,
                         comp_dtrs:list(sign)].

schema:: head_comp( syn:loc:head:subj:[_|_],
                    head_comp_struct,
                    [head_dtr,comp_dtrs]
                   ).
```

Notes:

1. For list valued attributes such as the `comp_dtrs` feature in the above examples the local tree licenced by the schema corresponds to the concatenation of all the members of that list with the rest of the structure.

Comments:

1. The *schema specification language* permits schemas to be specified independently of HPSG principles. A separate *principle interface specification* is employed to relate principles with the schemas that they apply upon.

2. The *schema specification language* has been designed so that the *Mother-FD* need not contain the *DTRS* features (*i.e.* `subject-dtr, head-dtr` *etc.*) since the *DTRS-FD* is specified separately. This makes chart-based parser/generators a lot more efficient since it considerably reduces the amount of copying of the *DTRS* features. The HPSG English grammar implementation that comes with the ALE-2.0.2 distribution takes advantage of this approach. Note that for chart-based systems the *DTRS* features can be recovered from the *chart*.

3. The simple LP constraints as defined in the current specification permit partial descriptions of local trees that are licenced by the schema definition. An empty *LocalLPConstraint* would, for instance, licence all permutations of the local tree.

PRINCIPLE INTERFACE SPECIFICATION LANGUAGE

Syntax:

principles::
Principle-Interface-Spec$_1$,
...,
Principle-Interface-Spec$_n$.

Remainder Syntax:

Principle-Interface-Spec $\longrightarrow$ *DTRS-Spec* `->` *Principle.*

DTRS-Spec $\longrightarrow$ *DTRS-Type*
| ~*DTRS-Type*
| [*DTRS-Type*$_1$,..., *DTRS-Type*$_n$]

DTRS-Type $\longrightarrow$ *Type*

Principle $\longrightarrow$ *Predicate-Name*(*FArg*$_1$,...,*FArg*$_n$)

FArg $\longrightarrow$ **'MOTHER'** | **'DTRS'** | *DTR* | [*FArg*$_1$, ..., *FArg*$_n$] | [*FArg*$_1$, ..., *FArg*$_n$ | *FArg*]

As an example, the following definition specifies the interface for the `subcat_principle/3`:

```
head_comp_struct approp [head_dtr:sign,
                         comp_dtrs:list(sign)].

schema:: head_comp( syn:loc:head:subj:[_|_],
                    head_comp_struct,
                    [head_dtr,comp_dtrs]
                   ).
principles ::
[  ....
   head_comp_struct -> subcat_principle('MOTHER',head_dtr,comp_dtrs)
   ....
]
```

The actual call for the `subcat_principle/3` would now consist in its arguments the *value* of the 'MOTHER' sign, the *value* of the `head_dtr` and the list value of `comp_dtrs`. `subcat_principle/3` would only be called in the context of `head_comp_struct` being generated by a schema.
Notes:

- 'MOTHER' stands for the *Mother-FD* of a `Schema` definition.
- 'DTRS' stands for the list of *DTR-FD*s of a `Schema` definition.
- If 'DTRS' is the result of applying a schema which employs a *FixedLocalTree* then order of elements in the 'DTRS' list is the same as that in the *FixedLocalTree*, otherwise the order of elements in 'DTRS' is arbitrary.
- The *DTRS-Spec* list [*DTRS-Type*$_1$,..., *DTRS-Type*$_n$] is interpreted disjunctively.
- If the list of *DTRS-Spec* is empty or not defined in a *Principle-Interface-Spec* then the corresponding principle is called for every possible schema for which the *FArg*'s are appropriate. Thus the following interface specification for the `head_feature_principle/2` would allow `head_feature_principle/2` to be called for every schema which has the `head_dtr` feature appropriate for its *DTRS-FD*:

```
principles ::
[  ....
   head_feature_principle('MOTHER',head_dtr)
   ....
]
```

Restrictions:

1. (Mother Type): *DTRS-Type* must be appropriate for at least one *DTRS-FD* of at least one schema definition.

2. (Principle Existence): If a *Principle* P/n is specified in a *Principle-Interface-Spec* then a *Predicate-Definition* for P/n must exist.

RELATIONAL LANGUAGE

Syntax:

Predicate-Head.

Predicate-Head :- *Predicate-Body.*

Remainder Syntax:

Predicate-Head $\longrightarrow$ *Predicate-Name*(FD_1,...,FD_n).

Predicate-Body $\longrightarrow$ *Predicate-Call*$_1$,
...,
Predicate-Call$_n$

Predicate-Call $\longrightarrow$ *Predicate-Name*(*PredArg*$_1$,...,*PredArg*$_n$)
| *Guarded-Constraint*

PredArg $\longrightarrow$ *FD* | *Macro-Call*

Macro-Call $\longrightarrow$ @*Predicate-Call*

Notes:

1. A *Macro-Call* @p(A_1,...,A_n) is identical to V_{n+1} in the *Predicate-Call* p(A_1,...,A_n,V_{n+1}) where V_{n+1} is a fresh variable within the *Predicate-Body* in which @p(A_1,...,A_n) appears.

Restrictions:

1. (non-recursive): There should not be a cyclic dependency between a *Predicate-Call* and a *Predicate-Head.*

2. (Macro): If a *Macro-Call* @p(A_1,...,A_n) is specified in a *Predicate-Body* then the predicate p/$n+1$ must be specified or a built-in.

Terminology:

1. (deterministic): A predicate is called deterministic if there is only a single predicate definition for it and every predicate in the *Predicate-Call* of its body is deterministic.

2. (non-deterministic): A predicate is non-deterministic if it is not deterministic.

Comments:

1. (monotonic) The restriction to the pure Horn subset permits a fully declarative monotonic interpretation. Thus no *cuts* (!s) and no *negation-by-failure.* Thus the language admits various resolution strategies including the Andorra strategy (*cf.* CUF) and the Prolog strategy (*cf.* ALE).

2. (efficiency) The elimination of recursion means that most of the relational predicates can be compiled-out and efficient and simple tabulation techniques can be employed. Note that inclusion of recursion would necessitate an Andorra resolution strategy to enable termination of programs that require goal re-ordering at run-time.

3. (determinism) Non-deterministic predicates can largely be eliminated by use of guarded constraints. In fact, it should be possible to push all user-specified non-determinitic predicates to lexical lookup.

Built-in predicate definitions

The built-in predicate library is subdivided into the following categories:

1. Equality - *=/2*
2. List manipulation - *append/3, select/3*
3. Maplist - *maplist/3, accumulate/3*
4. List-based set operations - *set_member/3, set_difference/3, set_union/3, set_disjoint_union/3, set_subset/3, set_disjoint/2 set_difference/3*
5. Guarded constraints - *case/3, case/2, first_of/1*
6. FST operations - *replace/3*

Re-definition of built-in predicates causes a compile-time error.
Equality predicate

- **Name:** *=/2*
  Mode: =(?,?)
  Description: `=/2` treated as if defined by:

```
'='(X,X).
```

List manipulation

- **Name:** *append/3*
  Mode: append(?,+,?), append(+,?,?)
  Description: `append(List1:`*`list`*(α)`, List2:`*`list`*(α)`, Result:`*`list`*(α)`)` is `true` if:
  `Result` is the result of contatenation of the list `List1` with the list `List2`.

- **Name:** *select/3*
  Mode: select(?,+,?)
  Description: `select(Element:`α`, List:`*`list`*(α)`, Remainder:`*`list`*(α)`)` is `true` if:
  `Element` is a member of `List` and `Remainder` is the remainder of `List` with `Element` removed.

MapList

- **Name:** *maplist/3*
  Mode: maplist(+,+,+)
  Description: `maplist(ArgList:`*`list`*`, Pred:`*`Predicate`*`, RestArgs:`*`list`*`)` is `true` if:
  `ArgList` is of the form `[`$\mathtt{A}_0$`, ..., `$\mathtt{A}_k$`]`
  `RestArgs` is of the form `[`$\mathtt{PArg}_0$`, ..., `$\mathtt{PArg}_n$`]` and
  `Pred(`$\mathtt{A}_i$`,`$\mathtt{PArg}_0$`, ..., `$\mathtt{PArg}_n$`)` is true for each i:$0 \leq i \leq k$

- **Name:** *accumulate/3*
  Mode: accumulate(+,+,?)
  Description: `accumulate(InList:`*`list`*`, Pred:`*`Predicate`*`, OutList:`*`list`*`)` is `true` if:
  `InList` is of the form `[`$\mathtt{A}_0$`, ..., `$\mathtt{A}_k$`]`
  `Pred(`$\mathtt{A}_i$`, `$\mathtt{O}_i$`)` is true for each $0 \leq i \leq k$
  `OutList` is `[`$\mathtt{O}_0$`, ..., `$\mathtt{O}_k$`]`

List based set operations

- **Name:** *set_member/2*
  Mode: *set_member(?,+)*
  Description: `set_member(Elem, Set:list)` is `true` if:
  `Elem` is either token-identical with an element of `Set` (deterministic) or
  `Elem` can be unified with an element of `Set` (non-deterministic)

- **Name:** *set_subset/2*
  Mode: *set_subset(?,+)*
  Description: `set_subset(SubSet:list, Set:list)` is defined as if by:

```
set_subset(Sub, Set):-
      maplist(Sub, set_member, Set).
```

- **Name:** *set_union/3*
  Modes: *set_union(+,+,?)*
  set_union(+,-,+)
  set_union(-,+,+)
  Description: `set_union(Set`$_1$`:list, Set`$_2$`:list, Union:list)` is `true` if:

  1. every element of `Union` is unified with either an element of `Set`$_1$ or `Set`$_2$ and
  2. every element of `Set`$_1$ is unified with an element of `Union`
  3. every element of `Set`$_2$ is unified with an element of `Union`

- **Name:** *set_disjoint/2*
  Mode: *set_disjoint(+,+)*
  Description: `set_disjoint(Set`$_1$`:list, Set`$_2$`:list)` is `true` if:
  every element of `Set`$_1$ is **not** token-identical any element of `Set`$_2$

- **Name:** *set_disjoint_union/3*
  Modes: *set_disjoint_union(+,+,?)*
  set_disjoint_union(+,-,+)
  set_disjoint_union(-,+,+)
  Description: `set_disjoint_union(Set`$_1$`:list, Set`$_2$`:list, Union:list)` is defined as if by:

```
set_disjoint_union(Set1,Set2,Union):-
      set_union(Set1,Set2,Union),
      set_disjoint(Set1,Set2).
```

- **Name:** *set_difference/3*
  Modes: *set_difference(+,+,?)*
  Description: `set_difference(Set`$_1$`:list, Set`$_2$`:list, Union:list)` is defined as if by:

```
set_difference(Set1,Set2,Difference):-
      set_subset(Difference,Set1),
      set_disjoint_union(Difference,Set2,DisjointUnion),
      set_subset(Set1,DisjointUnion).
```

Note: This definition of set_difference/3 follows the mathematical definition:
$D = S_1 - S_2 \Longleftrightarrow D \subseteq S_1 \ \wedge \ S_1 \subseteq D \uplus S_2$

Guarded Constraints

Syntax:

Guarded-Constraint ⟶ **case(** *Var,Cond-Action-Pairs*) **else** *ElseGoal*
| **case(** *Var,Cond-Action-Pairs*)
| **case(** *Var-Cond-Action-Triples*) **else** *ElseGoal*
| **case(** *Var-Cond-Action-Triples*)
| **first_of**([*Predicate-Call*$_1$, ..., *Predicate-Call*$_n$])

Remainder Syntax:

Cond-Action-Pairs ⟶ [*Guard*$_1$ =¿ *Action*$_1$,..., *Guard*$_n$ =¿ *Action*$_n$]

Var-Cond-Action-Pairs ⟶ [(*Var*$_1$ = *Guard*$_1$) =¿ *Action*$_1$,..., (*Var*$_n$ = *Guard*$_n$) =¿ *Action*$_n$]

ElseGoal ⟶ *Action*

Guard ⟶ *RFD* | *RExp*

RFD ⟶ *feature*:*RFD*
| *Type*
| *RFD*$_1$ & *RFD*$_2$

RExp ⟶ *String*
| **top**
| RExp*
| RExp+
| RExp$_1$-RExp$_2$
| RExp$_1$;RExp$_2$
| RExp$_1$&RExp$_2$

Action ⟶ *Predicate-Call*

Restrictions:

1. Every predicate within a call of `first_of/1` must be deterministic.
2. If the *Guard* in a *Cond-Action-Pairs* or *Var-Cond-Action-Pairs* is a *RExp* then the corresponding *Var* must be a *String*.

FST Operations

Syntax:

FST-Operation $\longrightarrow$ **replace(** *Var-String-Spec*, *Transducer*, *Var-String-Spec* **)**

Var-String-Spec $\longrightarrow$ *Var* | *String-Spec*

String-Spec $\longrightarrow$ [$\mathit{Var\text{-}Letter}_1, \ldots, \mathit{Var\text{-}Letter}_n$]

Var-Letter $\longrightarrow$ *Disj-Constants*
| *Var*
| *Var* `&` *Disj-Constants*

Disj-Constants $\longrightarrow$ *Constant* | *Constant*;*Disj-Constants*

Transducer $\longrightarrow$ `top`
| *BasicTransducer*
| *BasicTransducer*`*`
| *BasicTransducer*`+`
| $\mathit{Transducer}_1$ `-` $\mathit{Transducer}_2$

BasicTransducer $\longrightarrow$ `null`:*Disj-Constants-List*
| *Disj-Constants-List*:`null`
| *Disj-Constants-List*:*Disj-Constants-List*
| *BasicTransducer* `;` *BasicTransducer*

Disj-Constants-List $\longrightarrow$ [$\mathit{Disj\text{-}Constants}_1, \ldots, \mathit{Disj\text{-}Constants}_n$]

LEXICON SPECIFICATION LANGUAGE

Recommended Syntax:

Lexical-Entry $\longrightarrow$ *Lexical-Predicate-Name*(*Var-Macro-Call*$_1$, ..., *Var-Macro-Call*$_n$).

Lexical-Predicate-Name $\longrightarrow$ *Predicate-Name*

Var-Macro-Call $\longrightarrow$ Macro-Call | Var & Macro-Call

Lex-Entry-Hook $\longrightarrow$ **lex(***FD***) :-**
Lexical-Predicate-Name(*FD*$_1$, ..., *FD*$_n$).

LEXICAL RULE SPECIFICATION LANGUAGE

Syntax:

lex_rule(*Input-FD, Output-FD***):-**
Predicate-Body.

Notes:

1. *Output-FD* is interpreted as an update term on *Input-FD*. This means that *Output-FD* need only specify the path-values for which the *Input-FD* needs modification. Co-indexation for the rest of the paths is done automatically by the update operation.
   Consider the `finite_3sg` lexical rule given below:

```
lex_rule::
    finite_3sg(@ortho(O) & @vform(bse) &
               @aux(minus) & @subj_cat(np)
               ,
               @ortho(NewO) & @vform(fin) &
               @aux(minus)  & @subj_cat(np) & @subj_case(nom)
              ):-
              first_of([replace(O, top - [y]:[i,e,s], NewO),
                        replace(O, top - [e]:[e,s], NewO)
                      ]).
```

   All the macro calls in the above lexical rules are simply access functions. The replace operation is fully bidirectional.

Restrictions:

- `(String)`: Every *Letter* `L` within a *String* must be defined to be a subtype of the predefined type **constant**.

7.3.3 The Update Operation in Feature Logic

This section describes a new logical operation in feature logic that allows a feature description to be employed for updating another feature description. The update operation is monotonic as opposed to comparable operations such as default unification which is nonmonotonic. Furthermore updates can be computed deterministically and do not involve any bookkeeping reducing space requirements. The update operation currently finds a useful application in the description of HPSG lexical rules.

Introduction

The update operation can be thought of as replacing certain selected feature-paths with new information. Consider for instance the example given in (7.1) which depicts the situation in which the description $\boxed{1}$ is being updated with the description $\boxed{2}$. The updated description is $\boxed{2}$ itself with the additional information $\begin{bmatrix}\text{PER } \mathit{third}\end{bmatrix}$ derived from $\boxed{1}$.

(7.1)

$$\boxed{1}\begin{bmatrix}\text{CASE} & \mathit{acc} \\ \text{PER} & \mathit{third}\end{bmatrix} - \boxed{2}\begin{bmatrix}\text{CASE } \mathit{nom}\end{bmatrix} = \boxed{2}\begin{bmatrix}\text{CASE} & \mathit{nom} \\ \text{PER} & \mathit{third}\end{bmatrix}$$

The idea here is that the feature CASE is being updated with the *nom* value irrespective of its previous content. Since, nothing is said about the PER feature its value is simply carried over.

In order to characterise the logic of the update operation we start out by first considering a more primitive operation which we shall call the *feature difference operation*.

What is feature difference operation?

Stated simply, assume that we are given two feature structures FA and FB then a feature difference operation $FA =_{\{f\}} FB$ states that the feature structure FA is identical to the feature structure FB *except* for the value of the feature f for which it is *unknown* whether the values are identical.

Given below are some examples and non-examples of feature difference.

Let: $FA = \begin{bmatrix}\text{CASE} & \mathit{acc} \\ \text{NUM} & \mathit{sing} \\ \text{PER} & \mathit{third}\end{bmatrix}$ $FB = \begin{bmatrix}\text{CASE} & \mathit{nom} \\ \text{NUM} & \mathit{sing} \\ \text{PER} & \mathit{third}\end{bmatrix}$ $FC = \begin{bmatrix}\text{CASE} & \mathit{acc} \\ \text{NUM} & \mathit{plur} \\ \text{PER} & \mathit{third}\end{bmatrix}$

In the following discussion we assume that atoms are *extensional i.e.* two distinct occurences of the same atom are token-identical.

Then $FA =_{\{case\}} FB$ since FA and FB only differ in their *case*-values. Similarly $FA =_{\{num\}} FC$. However, $FA =_{\{case\}} FC$ does not hold since FA and FC differ in their NUM-values. Also $FB =_{\{case\}} FC$ does not hold since FB and FC in addition to differing in their CASE values also differ in their NUM values.

Also, note that feature difference is a *symmetric* operation *i.e.* $FA =_{\{case\}} FB$ is same as $FB =_{\{case\}} FA$. And, it is also *reflexive* since $FA =_{\{case\}} FA$ and in general $F =_{\{f\}} F$ for any feature description F and any feature f. This is so since the statement F *is identical in every respect to itself except in the value of the feature* f is true for any feature any feature description F and any feature f.

Can we have feature difference involving more than a single feature?

The answer is *yes*. For instance $FB =_{\{case,num\}} FC$ since FB and FC differ in at most two features *case* and *num*.

In the following sections we provide both a denotational and operational semantics of a feature logic enriched with the feature difference operation. Our aim is to extend the operation to sets of feature-paths and then finally to the more general update operation which is the focus of the section.

Feature terms with feature difference

In this section we provide the syntax and semantics of a term language containing feature descriptions and feature difference. We follow and extend the approaches of [Smolka, 1992] for feature logic.

We assume an alphabet consisting of $x, y, z, \ldots \in \mathcal{V}$ the set of *variables*; $f, g, \ldots \in \mathcal{F}$ the set of *feature symbols* and $A, B, C, \ldots \in \mathcal{S}$ the set of type symbols. In addition, the set of atoms $\mathcal{C}$ is assumed to be a subset of $\mathcal{S}$ *i.e.* $\mathcal{C} \subseteq \mathcal{S}$. Furthermore, we require that $\bot, \top \in \mathcal{S}$.

The syntax of our term language is defined by the following BNF definition:

S, T	$\longrightarrow$	
	x	variable
	A	type
	$f : T \quad (f \in \mathcal{F})$	feature term
	$S \& T$	conjunction
	$S_{\{f_1, \ldots, f_n\}}$	feature difference

Semantics

An interpretation structure is a structure of the form: $\mathcal{I} =< \mathcal{U}^I, \mathcal{F}, \mathcal{S}, \mathcal{C}, I >$ where:

1. $\mathcal{U}^I$ is an arbitrary non-empty set
2. I is an interpretation function that maps :
   (a) every feature symbol $f \in \mathcal{F}$ to a partial function $f^I : \mathcal{U}^I \rightharpoonup \mathcal{U}^I$
   (b) every type symbol A in $\mathcal{S}$ to a set : $A^I \subseteq \mathcal{U}^I$ *s.t.*
      - $\bot^I = \emptyset$
      - $\top^I = \mathcal{U}^I$
   (c) every atom a in $\mathcal{C}$ to a singleton $\{a^I\} \subseteq \mathcal{U}^I$ *s.t.* distinct atoms are interpreted distinctly *i.e.*:
      - if $a \not\equiv b$ then $a^I \not\equiv b^I$

An $\mathcal{I}$-*assignment* α is a total function $\alpha : \mathcal{V} \to \mathcal{U}^I$.

The denotation of a term T with respect to an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α is given by the following definitions:

1. $[\![x]\!]^{\mathcal{I},\alpha} = \{\alpha(x)\}$
2. $[\![A]\!]^{\mathcal{I},\alpha} = A^I$
3. $[\![f : T]\!]^{\mathcal{I},\alpha} = \{e \in \mathcal{U}^I \mid f^I(e) = e' \wedge e' \in [\![T]\!]^{\mathcal{I},\alpha}\}$
4. $[\![S \& T]\!]^{\mathcal{I},\alpha} = [\![S]\!]^{\mathcal{I},\alpha} \cap [\![T]\!]^{\mathcal{I},\alpha}$
5. $[\![S_{\{f_1,\ldots,f_n\}}]\!]^{\mathcal{I},\alpha} =$
   $\{e \in \mathcal{U}^I \mid \forall f \in \mathcal{F} : (f \notin \{f_1, \ldots, f_n\} \wedge f^I(e)\downarrow) \Rightarrow f^I(e) \in f^I([\![S]\!]^{\mathcal{I},\alpha})\}$

In the above definitions $f^I([\![S]\!]^{\mathcal{I},\alpha})$ denotes the set:
$f^I([\![S]\!]^{\mathcal{I},\alpha}) = \{f^I(e) \mid e \in [\![S]\!]^{\mathcal{I},\alpha} \ \wedge \ f^I(e)\downarrow\}$

A term T is **consistent** if there exists an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α such that $[\![T]\!]^{\mathcal{I},\alpha} \neq \emptyset$.

Constraint solving with feature difference

The first step in the consistency checking of the augmented feature logic is to breakdown complex terms into conjunctions of simple constraints employing the decomposition rules given in figure 7.8.

Decomposition rules

(DFeat) $\dfrac{x = f : T \ \&\ C_s}{x = f : y \ \&\ y = T \ \&\ C_s}$
where y is new and T is not a variable

(DConj) $\dfrac{x = S \ \&\ T \ \&\ C_s}{x = S \ \&\ x = T \ \&\ C_s}$

(DDiff) $\dfrac{x = S_{\{f_1,\ldots,f_n\}} \ \&\ C_s}{y = S \ \&\ x =_{\{f_1,\ldots,f_n\}} y \ \&\ C_s}$
where y is new

Figure 7.8: Decomposition rules

A constraint is simply a term of the form $x = T$ where x is a variable and T is an term. We say that an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α satisfies $x = T$ written $\mathcal{I}, \alpha \models x = T$ *if*:

- $\mathcal{I}, \alpha \models x = T \iff \alpha(x) \in [\![T]\!]^{\mathcal{I},\alpha}$

The satisfaction relation for conjunctions of constraints is extended in the same way as for predicate logic.

As an example, the decomposition of the feature description corresponding to the feature structure FA results in the following transformation (from 3 applications of (DConj) and 3 applications of (DFeat)):

$$\begin{aligned} & x = (case : acc \ \&\ num : sing \ \&\ per : third) \\ & \longrightarrow \\ & x = case : z_1 \ \&\ z_1 = acc \ \& \\ & x = num : z_2 \ \&\ z_2 = sing \ \& \\ & x = per : z_3 \ \&\ z_3 = third \end{aligned} \tag{7.2}$$

Simplification rules for feature logic

(SEquals) $\dfrac{x = y \ \&\ C_s}{x = y \ \&\ [x/y]C_s}$
if $x \not\equiv y$ and x occurs in C_s

(SFeat) $\dfrac{x = f : y \ \&\ x = f : z \ \&\ C_s}{x = f : y \ \&\ y = z \ \&\ C_s}$

Figure 7.9: Constraint simplification rules for feature logic

Notation: $[x/y]C_s$ denotes result of replacing every occurrence of x with y in C_s.

The decomposition of the feature difference term $FA_{\{case\}}$ will then be (from single application of (DDiff)):

$$\begin{aligned} & x =_{\{case\}} (case : acc \ \&\ num : sing \ \&\ per : third) \\ & \longrightarrow \\ & y = (case : acc \ \&\ num : sing \ \&\ per : third) \ \& \\ & x =_{\{case\}} y \end{aligned} \tag{7.3}$$

which incorporating the earlier derivation yields:

$$
\begin{aligned}
& y = case : z_1 \ \&\ z_1 = acc \ \& \\
& y = num : z_2 \ \&\ z_2 = sing \ \& \\
& y = per : z_3 \ \&\ z_3 = third \ \& \\
& x =_{\{case\}} y
\end{aligned} \tag{7.4}
$$

The next step is to apply the simplification rules given in figure 7.9. The simplification rules take care of any normalisation necessary for the given feature descriptions. For the constraints derived from FA above these are already in normal form and hence further simplification is unnecessary.

To update FA with the value of the CASE feature set to *nom* we can now employ the following description which can be stated in our logic:

$$(case : acc \ \&\ num : sing \ \&\ per : third)_{\{case\}} \ \&\ case : nom$$

Since we already know the decomposition of $x = (case : acc \ \&\ num : sing \ \&\ per : third)_{\{case\}}$ from (7.3) and (7.4) we get:

$$
\begin{aligned}
& x =_{\{case\}} (case : acc \ \&\ num : sing \ \&\ per : third) \ \&\ case : nom \\
& \longrightarrow \\
& y = case : z_1 \ \&\ z_1 = acc \ \& \\
& y = num : z_2 \ \&\ z_2 = sing \ \& \\
& y = per : z_3 \ \&\ z_3 = third \ \& \\
& x =_{\{case\}} y \& \\
& x = case : w_1 \ \&\ w_1 = nom
\end{aligned} \tag{7.5}
$$

Now we can apply our simplification rule for feature difference given in figure 7.10. The first application of (SDiff) to the pair of constraints $y = num : z_2 \ \&\ x =_{\{case\}} y$ yields:
$y = num : z_2 \ \&\ x =_{\{case\}} y \ \&\ x = num : z_2$

resulting in the new set of constraints:

$$
\begin{aligned}
& y = case : z_1 \ \&\ z_1 = acc \ \& \\
& y = num : z_2 \ \&\ z_2 = sing \ \& \\
& y = per : z_3 \ \&\ z_3 = third \ \& \\
& x =_{\{case\}} y \& \\
& \underline{x = num : z_2} \ \& \\
& x = case : w_1 \ \&\ w_1 = nom
\end{aligned} \tag{7.6}
$$

(Note: Derived constraints are indicated with an underline.)

A further application of (SDiff) yields the normal description:

$$
\begin{aligned}
& y = case : z_1 \ \&\ z_1 = acc \ \& \\
& y = num : z_2 \ \&\ z_2 = sing \ \& \\
& y = per : z_3 \ \&\ z_3 = third \ \& \\
& x =_{\{case\}} y \& \\
& x = num : z_2 \ \&\ \underline{x = per : z_3} \ \& \\
& x = case : w_1 \ \&\ w_1 = nom
\end{aligned} \tag{7.7}
$$

Extension to paths

What we want to do in this section is to consider terms of the form $S_{\{P_1,\ldots,P_n\}}$ where the P_i's are paths as opposed to just feature symbols that we have considered so far. *What do we mean by such terms?*

The essential point to consider is the mathematical fact that if two feature structures GA and GB differ in the path $f_1 : f_2$ then this implies that they differ in the path f_1 too. The denotation function given below takes this into account in defining the semantics of the new term $S_{\{P_1,\ldots,P_n\}}$.

Simplification rule for the feature difference constraint

(SDiff) $$\frac{x = f : x_1 \ \&\ x =_{\{f_1,\ldots,f_n\}} y \ \&\ C_s}{x = f : x_1 \ \&\ x =_{\{f_1,\ldots,f_n\}} y \ \&\ y = f : x_1 \ \&\ C_s}$$
if $f \notin \{f_1, \ldots, f_n\}$ and $y = f : x_1 \notin C_s$

Figure 7.10: Simplification rule for the feature difference constraint

$$[\![S_{\{P_1,\ldots,P_n\}}]\!]^{\mathcal{I},\alpha} = \{e \in \mathcal{U}^I \mid \forall P \in \mathcal{F}^* : (P \notin prefix(\{P_1, \ldots, P_n\}) \ \wedge\ P^I(e)\downarrow) \Rightarrow P^I(e) \in P^I([\![S]\!]^{\mathcal{I},\alpha})\}$$

Notation: $\mathcal{F}^*$ denotes the set of all feature-paths generated from $\mathcal{F}$.

Intuitively speaking, the above definition states that $S_{\{P_1,\ldots,P_n\}}$ denotes all feature-structures which are identical to the ones denoted by S except for the values of the paths $P_1, \ldots, P_n$ and their prefixes.

To check for consistency of terms containing the path difference term the first step is to decompose these to the simpler constraint of the form $x =_{\{P_1,\ldots,P_n\}} y$ employing the decomposition rule given in figure 7.11. The next step is to employ the simplification rules for path difference given in figure 7.12. Simplification rule (SPath1) is the replacement for the feature difference simplification rule (SDiff) and adds a feature-value pair $y = f : x_1$ just in case the feature f does not appear as the first prefix in the set of paths $P_1, \ldots, P_n$ since in this case we know (from the semantics) that the f-value of x and y are the same.

Decomposition rule for path difference terms

(DPath) $$\frac{x = S_{\{P_1,\ldots,P_n\}} \ \&\ C_s}{y = S \ \&\ x =_{\{P_1,\ldots,P_n\}} y \ \&\ C_s}$$
where y is new

Figure 7.11: Decomposition rule for path difference terms

Rule (SPath2) is the "recursive" part of the simplification procedure which propagates path difference constraints to f-values of x and y.

Rule (SPEq) is a redundant rule which is given purely to remove constraints which are logically valid.

Simplification rules for the path difference constraint

(SPath1) $$\frac{x = f : x_1 \ \&\ x =_{\{P_1,\ldots,P_n\}} y \ \&\ C_s}{x = f : x_1 \ \&\ y = f : x_1 \ \&\ x =_{\{P_1,\ldots,P_n\}} y \&\ C_s}$$
if $y = f : x_1 \notin C_s$ and f is **not** the *initial* prefix in any of $P_1, \ldots, P_n$

(SPath2) $$\frac{x = f : x_1 \ \&\ x =_{\{f:P_1,\ldots,f:P_n,Q_0,\ldots,Q_m\}} y \ \&\ y = f : y_1 \ \&\ C_s}{x = f : x_1 \ \&\ x =_{\{f,Q_0,\ldots,Q_m\}} y \ \&\ x_1 =_{\{P_1,\ldots,P_n\}} y_1 \ \&\ y = f : y_1 \ \&\ C_s}$$
if f is **not** the *initial* prefix in any of $Q_0, \ldots, Q_m$

(SPEq) $$\frac{x =_{\{P_1,\ldots,P_n\}} x \ \&\ C_s}{C_s}$$

Figure 7.12: Simplification rules for the path difference constraint

Formally, we say that a set of constraints are in *normal form* if no decomposition and simplification rules apply to it. A given term is considered to be inconsistent if their normal form transformation contains a *clash*. We say that a set of constraints contains a clash if:

- there exists variable x and constraints of the form $x = a$ & $x = b \in C_s$ where a and b are distinct atoms

A set of constraints in normal form are *clash-free* if it does not contain a clash. It is clear that if a set of constraints contain a clash then it is *inconsistent*.

It is easy to verify the following soundness and completeness theorems so we do not provide a rigorous proof. The soundness theorem establishes that all our decomposition and simplification rules transform constraints into equivalent constraints. The completeness theorem on the other hand guarantees that clash-free constraints in normal form are indeed consistent.

Theorem 1 (Soundness) *Let $\mathcal{I}, \alpha$ be any interpretation, assignment pair and let C_s be any set of constraints. If a constraint solving rule transforms C_s to C_s' then:*

$$\mathcal{I}, \alpha \models C_s \textit{ iff } \mathcal{I}, \alpha \models C_s'$$

Theorem 2 (Completeness) *A constraint system C_s in normal form is consistent iff C_s is clash-free.*

We are now ready to consider the more general update operation.

The update operation

The idea is to employ a new term of the form $S - PV$ to intuitively mean the feature description obtained after "updating" the description S with the information in the conjunction of path-values PV. For completeness, the full syntax of the extended language is given by the following BNF definitions:

$$\begin{array}{lll} S, T & \longrightarrow & \\ & x & \text{variable} \\ & A & \text{type} \\ & f : T \quad (f \in \mathcal{F}) & \text{feature term} \\ & S - PV & \text{update} \\ & S \& T & \text{conjunction} \end{array}$$

$$\begin{array}{lll} PV_1, PV_2 & \longrightarrow & \\ & f_1 : \cdots : f_n : V & \text{path} - \text{value} \\ & PV_1 \& PV_2 & \text{conjunction} \end{array}$$

$$\begin{array}{lll} V & \longrightarrow & \\ & x & \text{variable} \\ & A & \text{type} \end{array}$$

Note that every path-value term is also an ordinary term and hence the definition of semantics for terms is carried over to path-value terms. We only need to provide the semantics for the update term which is given below:

$$\begin{array}{rl} [\![S - PV]\!]^{\mathcal{I},\alpha} = & [\![PV]\!]^{\mathcal{I},\alpha} \cap \\ & \{e \in \mathcal{U}^I \mid \forall P \in \mathcal{F}^* : (PV \not\models P\downarrow \ \wedge \ P^I(e)\downarrow) \Rightarrow P^I(e) \in P^I([\![S]\!]^{\mathcal{I},\alpha})\} \end{array}$$

Notation: $PV \not\models P\downarrow$ means that PV does not entail that the path P is defined where the entailment relation is as in predicate logic. In other words, we say that PV entails $P\downarrow$ if in every interpretation I there exists a variable assignment α such that $[\![P : \top]\!]^{I,\alpha} \subseteq [\![PV]\!]^{I,\alpha}$.

A closer inspection of the semantics of the update term reveals that it breaks down into the conjunction of two parts. The first part simply says that the denotation of $S - PV$ must contain PV. The second part is very close to the definition of the path difference term. This is intentional so that we can take advantage of the constraint solving machinery we have developed for terms involving path difference. In order to fully take advantage to the constraint solving rules for path difference we need to figure out if for a given path P whether $PV \not\models P\downarrow$ holds. But this is again trivial as shown by the following.

Given the path-value term $P_1 : V_1 \;\&\; \ldots \;\&\; P_n : V_n$ the paths $P_1, \ldots, P_n$ are paths entailed by the path-value term. By implication, this means that all prefixes of $P_1, \ldots, P_n$ are also entailed by the path-value term. Thus we can employ a decomposition rule such as the one given in figure 7.13 to decompose update terms. This discussion is also sufficient to establish the correctness of our decomposition procedure. The decomposition rule breaks down an update term into a ordinary

Decomposition rule for update terms

$$\text{(DUpdt)} \quad \frac{x = S - (P_1 : V_1 \;\&\; \ldots \;\&\; P_n : V_n) \;\&\; C_s}{y = S \;\&\; x = (P_1 : V_1 \;\&\; \ldots \;\&\; P_n : V_n) \;\&\; x =_{\{P_1,\ldots,P_n\}} y \;\&\; C_s}$$

where y is new

Figure 7.13: Decomposition rule for update terms

(path-value) term and a path difference term and hence the constraint solving rules that we have developed earlier in this section can be employed.

As a simple example consider the update term $FA - (case : nom)$. This will be decomposed into the following simpler constraints:

$$\begin{array}{ll} & x = ((case : acc \;\&\; num : sing \;\&\; per : third) - case : nom) \\ & \longrightarrow \\ (7.8) & x = case : nom \;\&\; \\ & y = (case : acc \;\&\; num : sing \;\&\; per : third) \;\&\; \\ & x =_{\{case\}} y \end{array}$$

We know that these constraints can be solved by the existing machinery for dealing with feature difference constraints. Furthermore, our soundness and completeness theorems remain unaltered with the addition of the decomposition rule for update terms. This completes our presentation of the update operation.

Comparison with default unification

Default unification [Bouma, 1990], [Carpenter, 1993b] [Calder, 1990] bears some resemblance to the update operation and hence justifies a comparison.

To see how the update operation differs from default unification it is enough to consider some simple examples. Let $\uplus$ denote the default unification operation, let A and B be two types whose conjunction $A\&B$ is consistent. Then the default unification of DA and DB proceeds as follows:

$$DA = \begin{bmatrix} \text{CASE} & A \\ \text{PER} & third \end{bmatrix} \quad DB = \begin{bmatrix} \text{CASE} & B \end{bmatrix}$$

$$\begin{bmatrix} \text{CASE} & A \\ \text{PER} & third \end{bmatrix} \uplus \begin{bmatrix} \text{CASE} & B \end{bmatrix} = \begin{bmatrix} \text{CASE} & A\&B \\ \text{PER} & third \end{bmatrix}$$

Thus, in the case where information can be combined consistently default unification gives the combined result. This contrasts sharply with the update operation where updating ignores the previous value thus giving:

$$\begin{bmatrix}\text{CASE } A \\ \text{PER} \quad \textit{third}\end{bmatrix} - \begin{bmatrix}\text{CASE } B\end{bmatrix} = \begin{bmatrix}\text{CASE } B \\ \text{PER} \quad \textit{third}\end{bmatrix}$$

The strong property of default unification also leads to its non-monotonic nature. For instance, given the following where x denotes a variable:

$$\begin{bmatrix}\text{CASE } nom \\ \text{PER} \quad \textit{third}\end{bmatrix} \uplus \begin{bmatrix}\text{CASE } x\end{bmatrix} = \begin{bmatrix}\text{CASE } x\&nom \\ \text{PER} \quad \textit{third}\end{bmatrix}$$

default unification co-indexes x with the *nom* value since it is consistent to do so. But now if we add the information that x is actually *acc* then the system needs to retract the conclusion that x is *acc* which it earlier arrived by default reasoning:

$$\begin{bmatrix}\text{CASE } nom \\ \text{PER} \quad \textit{third}\end{bmatrix} \uplus \begin{bmatrix}\text{CASE } x\&acc\end{bmatrix} = \begin{bmatrix}\text{CASE } x\&acc \\ \text{PER} \quad \textit{third}\end{bmatrix}$$

We arrive at the above result if we assume the *priority union* interpretation of default unification where second argument of $\uplus$ has priority over its first argument.

This example also illustrates the fact that implementations of default unification often involve bookeeping of past history to enable retraction of default conclusions.

In contrast, the corresponding update operation proceeds monotonically:

$$\begin{bmatrix}\text{CASE } nom \\ \text{PER} \quad \textit{third}\end{bmatrix} - \begin{bmatrix}\text{CASE } x\end{bmatrix} = \begin{bmatrix}\text{CASE } x \\ \text{PER} \quad \textit{third}\end{bmatrix}$$

but loses the information that the value of the CASE feature was initially *nom*.

Note also that it would be difficult to mimic the behaviour of the update operation employing default unification alone since this essentially requires the "uninstantiation" of an already instantiated value of a feature (in the above example, the value of the feature *case*).

Under the skeptical interpretation of default unification this would be possible if we had:

$$\begin{bmatrix}\text{CASE } nom \\ \text{PER} \quad \textit{third}\end{bmatrix} \uplus \begin{bmatrix}\text{CASE } acc\end{bmatrix} = \begin{bmatrix}\text{CASE } \top \\ \text{PER} \quad \textit{third}\end{bmatrix}$$

But note that this presupposes that we have knowledge about the initial value of the CASE feature so that we can add contradictory information.

In summary, due to the different nature of the update operation in comparison to default unification we currently think that the update operation serves a complementary purpose to default unification at least for the varieties (of default unification) discussed in this section. This issue deserves further investigation and has the potential of leading to hybrid formulations that combine aspects of both the update operation and default unification.

Lexical rules in HPSG and the update operation

Lexical rules in HPSG are employed to derive new lexical entries from existing one. They model various phenomena such as derivation of morphologically related forms of the same verb (such as *likes, liked* from *like*), passive verb forms (*e.g. gave* from *give*) and even filler-gap dependencies.

Consider a simple lexical rule to derive the finite form of a verb from its base form (we shall use a slightly simplified feature geometry notation and ignore orthography):

```
lex_rule(synsem:loc:head:(cat:(vform:bse & aux:minus) &
                          subj:[loc:cat:head:noun]),
         synsem:loc:head:(cat:(vform:fin & aux:minus) &
                          subj:[loc:(cat:head:noun &
                                     cont:index:(per:third &
                                                 num:sing))])).
```

The first argument of *lex_rule* is intended to be the "input" lexical rule and the second argument is the "output" lexical entry. The result of applying a lexical rule to a lexical entry results in the creation of a new lexical entry. The input to the lexical entry is unified with the *sign* of the lexical entry and the output of the lexical entry is added to the database of lexical entries. Failure in unification results in no new lexical entry.

One convention that is adopted is that the values of those paths that are not mentioned in the output of a lexical rule are assumed to be identical to that of the input. This means for instance that the value of the CONT attribute (which contains the semantics) will be token-identical between the input and the output.

Such lexical rules can be straightforwardly stated within the logic developed within this report as follows[9]:

```
lex_rule(x & synsem:loc:head:(cat:(vform:bse & aux:minus) &
                              subj:[loc:cat:head:noun]),
      x
      -
      ( synsem:loc:head:cat:vform:fin &
        synsem:loc:head:cat:aux:minus &
        synsem:loc:head:cat:subj:hd:loc:cat:head:noun &
        synsem:loc:head:cat:subj:tl:elist &
        cont:index:per:third &
        cont:index:per:num:sing)).
```

Note that we have been very careful to write down the lexical rule in exactly the syntax required by our logic which involves repeating subpaths of a given path. But allowing tree-structured paths within our logic only requires a very simple addition to break them down into a linear form. In fact we have deliberately chosen to adhere to the linear path-value syntax to keep our presentation simple.

Our encoding of lexical rules employing the update term shows that no "meta-level" (*cf.meta-rules* in GPSG) and theory external considerations are necessary for dealing with them. Lexical rules can be treated as binary relations in the extended feature logic developed in this report. This gives us a fully declarative semantics and a complete proof procedure.

For a complete treatment of lexical rules, one has to allow lexical rules to apply to the output of lexical rules until no new entries are generated. This procedure is potentially infinite. However such considerations are not the subject of this report. See [Meurers and Minnen, 1995] for a detailed computational treatment of such matters.

Related Work

In related work by Meurers [Meurers, 1995] a partial formal characterisation of HPSG lexical rules is provided. In Meurers approach, the analog of our update term is handled by an "external" *enriching* procedure. In contrast, the updating of a feature within our system is fully internalised within feature logic and is handled by employing update terms which are first class objects in the current setup.

[9] We assume that the list $[H|T]$ is encoded using the feature term $hd : H \;\&\; tl : T$ and the empty list is *elist*.

Conclusions

In this section we have provided a formal characterisation of a new kind of operation in feature logic called the update operation. The update operation allows selected paths of a feature description to be modified while keeping the rest identical. The update operation is fully internalised into feature logic by providing a new feature term called the update term which can be treated just like any feature term. The update operation is monotonic and we have provided a sound and complete decision procedure for determining consistency of terms involving updates. Finally we have shown that update terms can be employed to clarify the formal status of lexical rules in HPSG.

7.3.4 An Example: Compiling HPSG Standard Grammars to CUF

Introduction

This section describes the compilation step which automatically transforms an HPSG grammar written in the proposed encoding standard into a CUF (parsing) program runnable in the current CUF system 2.30. The compilation essentially requires that a parser for HPSG grammars is either written in CUF or put as an addition (a library) on top of CUF.[10]

In order to represent a standard HPSG efficiently in CUF, type system specifications will be mapped to CUF typing specifications and schemata, principles and lexicon specifications as well as lexical rules are mapped onto CUF sorts (relational dependencies).

Type specifications

CUF can easily express closed-world type hierarchies with glb semantics as required by the standard. In order to specify the necessary compilation step, we assume that

- the hierarchy is given as a set H of partial ordering statements $t > t'$ (t' is a subtype of t) representing a BCPO;
- H contains no cycles, i.e. its transitive closure is irreflexive;
- H is free of redundancy, meaning that if $t > t'$ can be concluded by transitivity from other statements in H, then it is not in H;
- H has a unique maximal element `top`, i.e. $\texttt{top} >^{+} t$ for any other t.

The bounded completeness condition ensures that every two elements which have a common subtype have a unique most general common subtype. Hence, for any two types t_1 and t_2, the intersection $\{t \mid t_1 > t \in H\} \cup \{t \mid t_2 > t \in H\}$ is either empty or singleton. Let us call two types disjoint, if they have no common subtype.

To represent H in CUF we model all types as subtypes of the CUF built-in type 'cfs' (complex feature structure). In this way, all types are allowed to have features, subject to further specification by means of feature declarations. The glb semantics of the hierarchy can be encoded in several different ways. A simple way, though perhaps not leading to the most efficient encoding, is the following:[11]

1. for each type t having a nonempty set of immediate subtypes $t_1, \ldots, t_n$ state

$$\texttt{t = t}_1 \texttt{ ; ... ; t}_n \texttt{.}$$

   (replacing `top` by `cfs`);

2. for each type t let $\Delta(t)$ be the set of maximal types that are disjoint from t, but not disjoint from any of t's supertypes; now for each t and $t' \in \Delta(t)$ add the statement

$$\texttt{t} \mid \texttt{t}'\texttt{.}$$

   if t comes before t' in the alphabetical ordering.

Note that $t' \in \Delta(t)$ iff $t \in \Delta(t')$, justifying the last condition. Moreover, whenever t_1 and t_2 are disjoint, then there exist supertypes (not necessarily proper) $t_1' >^* t_1$ and $t_2' >^* t_2$ for which a disjointness statement is generated. Obviously, this is sufficient since $t_1 \mid t_2$ follows from $t_1' \mid t_2'$.

[10] The second option takes the same route that is taken by the LexGram system of Esther König [König, 1995] which implements a lexicalised version of HPSG.

[11] Another, less direct, way would be to compute for each pair of types t_1 and t_2 its glb and adding to the subtyping statements of H glb constraints $t_1 \& t_2 = t_{glb(t_1,t_2)}$ (resp. disjointness constraints, if they have no glb) leaving out redundant statements (i.e. when the same glb results for a supertype of t_1 or t_2).

Feature Appropriateness

No conversion is necessary for feature appropriateness statements. They can be mapped one-to-one to CUF feature declarations.

Feature Descriptions and Relational Language

The relational language is used in the standard to define templates, lexical rules (??), as well as the distinguished predicates 'principles'/2 and 'lex_access'/2. The functional notation for the invocation of relational constraints as macros is viewed as sugared syntax. To translate the relational constraints to CUF we only have to do this syntactic reanalysis in the opposite direction and let the relational terms be sugared notation for CUF's functional-style language.

For stating the details of the translation we use the notation $\widehat{C}$ to denote the CUF equivalent of a construct C.

Feature descriptions: If FD is a feature description, we let $\widehat{FD}$ denote the term resulting from FD by leaving out all occurrences of the @ operator.

Goals: For a goal A we define

$$\widehat{A} := \begin{cases} \texttt{'\$true\$'(r(}\widehat{FD}_1\texttt{,...,}\widehat{FD}_{n-1}\texttt{) \& }\widehat{FD}_n\texttt{)} & \text{if } A = r(FD_1, \ldots, FD_n) \\ \texttt{guard\_i\_1(}\widehat{C}\texttt{,}\vec{X}_1\texttt{) \& ...\& guard\_i\_n(}\widehat{C}\texttt{,}\vec{X}_n\texttt{)} & \\ \quad \texttt{\& else\_i(}\widehat{C}\texttt{,}\vec{X}_0\texttt{)} & \text{if } A = CASE(i) \\ & \quad = case(C, [(G_1, A_1), \ldots, (G_n, A_n)], A_0) \\ \texttt{guard\_i\_1(}\widehat{C_1}\texttt{,}\vec{X}_1\texttt{) \& ...\& guard\_i\_n(}\widehat{C_n}\texttt{,}\vec{X}_n\texttt{)} & \\ \quad \texttt{\& else\_i(}\widehat{C_1}\texttt{,...,}\widehat{C_n}\texttt{,}\vec{X}_0\texttt{)} & \text{if } A = CASE(i) \\ & \quad = case([(C_1, G_1, A_1), \ldots, (C_n, G_n, A_n)], A_0) \end{cases}$$

For the translation of the case statements we assume a unique numbering of all occurrences of case statements in the program and denote the i-th occurrence by $CASE(i)$. Hence, in the definition above, i is the number of the case statement in question. Moreover, $\vec{X}_k$ shall denote the vector of nonlocal variables occurring in goal A_k.

Clauses: Each clause

$$pred(FD_1, \ldots, FD_n) : -A_1, \ldots, A_m.$$

is translated to a CUF sort definition

$$\texttt{pred(}\widehat{FD}_1\texttt{,...,}\widehat{FD}_{n-1}\texttt{) := }\widehat{FD}_n\texttt{ \& }\widehat{A_1}\texttt{ \& ...\& }\widehat{A_m}\texttt{.}$$

Moreover, every occurrence $CASE(i)$ of a case statement induces the definition of the respective `guard_i_j` and `else_i` sorts as follows. if $CASE(i) = case(C, [(G_1, A_1), \ldots, (G_n, A_n)], A_0)$, then add the definitions:

```
% in CUF file
guard_i_1(_,X⃗1) := Â1.
...
guard_i_n(_,X⃗n) := Ân.
else_i(_,X⃗0) := Â0.

% in control file
wait(guard_i_1(Ĝ1,_,...) -> _).
...
wait(guard_i_n(Ĝn,_,...) -> _).
wait(else_i(~Ĝ1 & ... & ~Ĝn,_,...) -> _).
```

And if $CASE(i) = case([(C_1, G_1, A_1), \ldots, (C_n, G_n, A_n)], A_0)$, then add the definitions:

$$
\begin{array}{l}
\texttt{\% in CUF file} \\
\texttt{guard\_}i\texttt{\_1(\_,}\vec{X_1}\texttt{) := }\widehat{A_1}\texttt{.} \\
\ldots \\
\texttt{guard\_}i\texttt{\_}n\texttt{(\_,}\vec{X_n}\texttt{) := }\widehat{A_n}\texttt{.} \\
\texttt{else\_}i\texttt{(}\underbrace{\texttt{\_,...,\_}}_{\text{n times}}\vec{X_0}\texttt{) := }\widehat{A_0}\texttt{.}
\end{array}
$$

$$
\begin{array}{l}
\texttt{\% in control file} \\
\texttt{wait(guard\_}i\texttt{\_1(}\widehat{G_1}\texttt{,\_,...) -> \_).} \\
\ldots \\
\texttt{wait(guard\_}i\texttt{\_}n\texttt{(}\widehat{G_n}\texttt{,\_,...) -> \_).} \\
\texttt{wait(else\_}i\texttt{(\~{}}\widehat{G_1}\texttt{, ..., \~{}}\widehat{G_n}\texttt{,\_,...) -> \_).}
\end{array}
$$

ID-Schemata

The ID-schemata are best construed as ternary relations between a mother structure, a structure containing all daughters, and a list representing the linear order in which the daughters should appear in the tree. Hence, the general form of ID-schema ($A_1, \ldots, A_m$ are optional relational constraints)

$$
\begin{array}{rl}
\langle \textit{M-FD} \rangle \Longrightarrow & D_1, \ldots, D_n --- \\
& \langle \textit{Ds-FD} \rangle \\
& :- \\
& A_1, \ldots, A_m.
\end{array}
$$

can be transformed into the following CUF sort definition:

$$
\begin{array}{ll}
\texttt{id\_schema(}\widehat{\textit{M-FD}}\texttt{,}\widehat{\textit{Ds-FD}}\texttt{) :=} & \texttt{[}\widehat{D_1}\texttt{,...,}\widehat{D_n}\texttt{] \&} \\
& \widehat{A_1}\texttt{ \& ...\& }\widehat{A_m}\texttt{.}
\end{array}
$$

However, in order to use a head-driven parser, we assume a further pre-compilation. Firstly, this will apply the principles which are stated in the binary predicate 'principles', and, secondly, it should single out the head daughter among the daughters. For that purpose the input grammar will need to contain a special declaration of what the concrete name of the head daughter feature is. Using this it is easy to convert the above format into one which is appropriate for head-driven parsing, e.g.

```
ps_rule(Head,LeftDtrs,RightDtrs) := Mother.
```

The Daughters structure itself, which collects all daughters into a feature structure, is irrelevant for parsing and is dropped here. Clearly, we need to assume that the descriptions of the single daughters are retained in the arguments `Head, LeftDtrs` and `RightDtrs`.

For the implementation of this rule pre-compiler we have two options. Either use partial evaluation or macro expansion. The second option has the advantage that the pre-compilation step can be performed without the help of an inference engine for feature logic, but limits the effect that principles can have on ID-schemata to trivial textual insertion. In general, partial evaluation can be used to build the bridge between highly general specifications of the local trees by means of general ID-schemata and principles and the requirement that for the efficiency of the parser we should use PS rules that are sufficiently instantiated to cut down the choice possibilities as much as possible. However, in CUF, partial evaluation is only rudimentarily supported. We close on the pre-compilation issue by noting that there is room for substantial improvement.

A Head-driven Parser

Although ID-schemata, principles and lexical entries can be made into a declarative description of which parse trees are to be associated with which strings, we propose not to leave the interpretation of these to the CUF interpreter (alone). Instead, a parser that implements a head-driven parsing strategy should be interposed. To motivate our decision on the parsing strategy we observe the following:

- Simple (not memoised) top-down parsing is prohibitive, because the phrase structure schemata are so general (and left-recursive).
- Simple (not memoised) bottom-up parsing could work, if we admit a restriction limiting the number of traces that can occur in a given string. This can either be achieved by adopting a requirement that each trace must be lexically licensed, or by banishing them altogether (as is done in chapter 9 of [Pollard and Sag, 1994a]).
- When applying a chart-based technique, we can exploit the fact that the format of the ID-schemata make the locality principle, which effectively lets schemata and principles only be conditions on *local* trees, a fixed property of the framework. This is so, because when applying a rule to form a new completed edge we only need to store the mother structure. The daughters structures may not have an effect higher up in the parse tree. If the daughters themselves were to appear in the mother structure (as in original HPSG), this would lead to an unnecessary duplication of structure to be copied.
- Finally, by the Head Feature Principle and the Valency Principle there is an inherent direction of information flow in HPSG parse trees, which is best exploited by a head-driven bottom-up (head-corner) strategy.

We therefore propose a head-corner parsing strategy with memoisation, as described for instance in [Bouma and van Noord, 1993]. As a first approximation the parser can be written directly in CUF (as shown below) making use of memo control statements. However, in order to achieve maximal efficiency the parser needs to be written in a more low-level language (Prolog or C), accessing the CUF grammar via the foreign language. That way the overhead of the CUF interpreter is avoided and specialised indexing schemes for the access of chart items can be supplied.

The head-corner parser assumes:

1. schemata and principles are compiled into a 4-place relation represented as the CUF sort `ps_rule(HeadDtr,LeftDtrs,RightDtrs) -> Mother.`
2. the lexicon is accessed via CUF sort `lex_access(Word) -> Sign`
3. the head path is declared, e.g. as follows:

   ```
   share_head(synsem:local:cat:H) := synsem:local:cat:H.
   ```

   This clause must be generated by the compiler from a declaration of the head path in the original HPSG grammar. This is not to be confused with the declaration of the head daughter feature.
4. `trace :=` ⟨*TraceSign*⟩ declares the structure of traces (also to be compiled from declarations in the original grammar)

```
%% head-driven parser a la Bouma/vanNoord EACL93
%% we assume that traces may never be heads

hpsg_derive(list) -> sign.
```

```
hpsg_derive(String) :=
        hpsg_derive_lr(String, [], []).


hpsg_derive_lr(list,list,list) -> sign.
%%             +     -   +
hpsg_derive_lr(From, To, RLim) :=
        has_head(From, RLim, Lex, LexFrom, LexTo) &
        head_corner(Lex, LexFrom, LexTo, From, To, From, RLim).


hpsg_derive_rl(list,list,list) -> sign.
%%             -     +   +
hpsg_derive_rl(From, To, LLim) :=
        has_head(LLim, To, Lex, LexFrom, LexTo) &
        head_corner(Lex, LexFrom, LexTo, From, To, LLim, To).


head_corner(sign,list,list,list,list,list,list) -> sign.
head_corner(Sign, From, To, From, To, _, _) := Sign.
head_corner(Head, HFrom, HTo, From, To, LLim, RLim) :=
        head_corner(rule(Head, LeftDtrs, RightDtrs),
                    go_left(LeftDtrs, HFrom, LLim),
                    go_right(RightDtrs, HTo, To1, RLim),
                    From, To, LLim, RLim
                   ).

go_left([], From, _) := From.
go_left([trace|Dtrs], To, Lim) :=  %% skip trace
        go_left(Dtrs, To, Lim).
go_left([hpsg_derive_rl(Mid,To,Lim) |Dtrs], To, Lim) :=
        go_left(Dtrs, Mid, Lim).

go_right([], To, _) := To.
go_right([trace|Dtrs], From, Lim) :=  %% skip trace
        go_right(Dtrs, From, Lim).
go_right([hpsg_derive_lr(From,Mid,Lim) |Dtrs], From, Lim) :=
        go_right(Dtrs, Mid, Lim).


has_head(From, To, Lex, LexFrom, LexTo) :=
        share_head(Lex) &
        true(LexFrom & pos_between(From,To) & [W|LexTo]) &
        true(Lex & lex_entry(W)).
%% guesses a word in the string between From and To s.t. the head info
%% of its lexical entry is compatible with that of the given goal sign
%% (result arg).
```

Library Predicates

For the library routines we have to decide whether to implement them in CUF or in Prolog. Although an implementation directly in CUF would have better termination properties than the Prolog implementation devised above — for example, `append` could be called on a partially constructed list returning constraints on initial parts of the result — we propose an implementation in

Prolog similar to the one in Section ?? for two reasons. Firstly, we thus achieve better agreement between different implementations of the standard, and, secondly, unnecessary overhead for the CUF interpreter can be avoided. The important difference to the implementation given above is that we have to employ CUF abstract data type routines to unify or access the structures given. The routines are then made available at the CUF level via the foreign language interface. Here we specify only the append relation as an example.

```
% in CUF file
foreign( es_append(intern,intern) -> intern ).

% in control file
wait(es_append(list,_) -> _).
wait(es_append(_,_) -> list).

es_append(List1, List2, Out) :-
    cuf_unify(List2,Out),
    cuf_extern2intern([],[],[],Elist),
    cuf_unify(List1,EList).
es_append(List1, List2, Out) :-
    cuf_path_value(List1,['F'],First),
    cuf_path_value(Out,['F'],First),
    cuf_path_value(List1,['R'],Rest1),
    cuf_path_value(Out,['R'],RestOut),
    es_append(Rest1,List2,RestOut).
```

7.4 LFG

See the LFG homepage: http://clwww.essex.ac.uk/LFG.

7.5 TAG: Tree Adjoining Grammar

7.5.1 Introduction

In this context, the important differences are in the syntactic representation, i.e. the representation of elementary trees. Therefore, the following formalizations are restricted to the representation of tree families and elementary trees. Interfacing these descriptions with various types of lexicons should be fairly straightforward. Also the interfacing morphology modules will differ quite substantially in different systems (e.g., file-based lexicons, Kimmo-style module, data-bases, etc.), without any importance to the TAG formalism. The common specification language which is proposed in section 7.5.5 also does not provide for any of the approaches to a compact representation of TAG grammars which are sketched in section 3.2.8.

7.5.2 The XTAG System

We give a pseudo BNF-Form for the format of files containing elementary trees. We use the following operators:

- `expression`* : zero or more repetitions of `expression`
- `expression`$^+$: one or one repetitions of `expression`
- `expression`$^?$: zero or one repetitions of `expression`
- `expression1` | `expression2` : `expression1` or `expression2`
- `` ``expression'' `` : `expression` is a terminal string
- `( expression )` : `expression`, the parentheses are used as a grouping operator

```
File ::= (Tree)*
Tree ::= ``('' Tree-Name Keyword-List ``)'' Tree-List
Tree-Name ::= LISP-string
Keyword-List ::=
``:COMMENT-DISPLAY? '' ( ``NIL'' | ``T'' )
``:FEATURE-DISPLAY? '' ( ``NIL'' | ``T'' )
``:EQUATION-DISPLAY? '' ( ``NIL'' | ``T'' )
``:UNIFICATION-EQUATIONS '' Unification-Equations-String
``:COMMENTS '' Comments-String
``:SHAPE '' ``:NONE''
``:BORDER-WIDTH '' Number
``:CONSTRAINT-STYLE '' ``(:DUTCH :ITALIC :NORMAL)''
``:CONNECTOR '' ``:LINE''
``:DEFAULT-STYLE '' ``(:DUTCH :BOLD :NORMAL)''
``:SUBSCRIPT-STYLE '' ``(:DUTCH :ROMAN :SMALL)''
``:WHITE-SPACE '' ``3''
``:MINIMUM-NODE-SEPARATION '' ``NIL''
``:LEVEL-SEPARATION '' ``NIL''
Comments-String ::= LISP-string

Unification-Equations-String ::= ``"'' ( Unification-Equation )* ``"''
Unification-Equation ::= Feature-Term `` = '' ( Feature-Term | Feature-Value )
Feature-Term ::= Node-Name ( ``_'' Subscript-String )? ( ``.''  ( ``t'' | ``b'' ) )? ``:''  Feature-Path
Feature-Path ::= ``<'' Feature ( `` '' Feature )* ``>''
Node-Name ::= ``NP'' | ``S'' | ``PP'' | ``N'' | ``P'' | ``VP'' | ...
Feature ::= ``mode'' | ``tense'' | ``agr'' | ``passive'' | ...
Feature-Value ::= ``+'' | ``-'' | ``acc'' | ``nom'' | ``3rd'' | ...
```

```
Tree-List ::= ‘‘(’’ Mother-Node ‘‘(’’ ( Subtree )* ‘‘)’’ ‘‘)’’
Mother-Node ::  Node
Subtree ::= Tree-List

Node ::= Node-Label Node-Keyword-List
Node-Label ::= ‘‘(( ’’ Node-Name-String ‘‘ . ’’  Subscript-String ‘‘ ))’’
Node-Name-String ::= ‘‘"’’ Node-Name ‘‘"’’
Subscript-String ::= LISP-String (possibly the empty string ‘‘""’’
Node-Keyword-List ::=
( ‘‘:substp T’’ | ‘‘:headp T’’ | ‘‘:footp T’’ )?
( ‘‘:constraints ""’’ ( ‘‘:constraint-type :DUMMY’’ )? |
‘‘:constraints "NA" :constraint-type :NA’’ )?
( ‘‘:display-feature?’’  display-feature)?
( ‘‘:shape’’ shape’’)?
( ‘‘:default-style’’ default-style)?
( ‘‘:border-width’’ border-width)?
( ‘‘:white-space’’ white-space)?
( ‘‘:connector’’ connector)?
( ‘‘:subscript-style’’ subscript-style)?
```

7.5.3 The VM-GEN System

Again, we give a pseudo BNF-Form for the format of files containing elementary trees:

```
File ::= Terminals Non-Terminals LP-Rules-Shorthands (Tree)*
Terminals ::= ‘‘(’’ ( ‘‘ ’’ Terminal )* ‘‘)’’
Terminal ::= LISP-string
Non-Terminals ::= ‘‘(’’ ( ‘‘ ’’ Non-Terminal )* ‘‘)’’
Non-Terminal ::= LISP-string
LP-Rules-Shorthands ::= ‘‘(’’ ( LP-Rule-Shorthand )* ‘‘)’’
LP-Rule-Shorthand ::= ‘‘(’’ Short-Name ( ‘‘ ’’ LP-expression )* ‘‘)’’

Tree ::= Tree-Name ‘‘(’’ Tree-Structure ‘‘)’’ Feature-Equations
Tree-Name ::= a name (i.e.  Lisp-Symbol)
Tree-Structure ::= Node ‘‘(’’ ( Subtree-Structure )+ ‘‘)’’ | Node

Node ::= Node-Name ( Constraint )? ( LP-rules )?
Node-Name ::= Lisp-Symbol ( ‘‘!’’  )? ( ‘‘*’’ )?
Constraint ::= ‘‘(’’ ( ‘‘:SA’’ ( Tree-Name )* | ‘‘:OA’’ ( Tree-Name )+ ) ‘‘)’’

LP-Rules ::= ‘‘(< ’’ ( LP-Rule )+ ‘‘)’’
LP-Rule ::= ‘‘(’’ Rule-Name Test ( Ordering )+ ‘‘)’’
Rule-Name ::= Lisp-Symbol
Test ::= ‘‘(test ’’ Test-expression ‘‘)’’
Test-Expression ::= ( And-Or-Expression | Feature-Value-Equation )
And-Or-Expression ::= ‘‘(’’ ( ‘‘and ’’ | ‘‘or ’’ ) ( Test-Expression )+ ‘‘)’’
Feature-Value-Equation ::= ‘‘(’’ Number ‘‘(’’ Feature-Path ‘‘)’’ Feature-Value ‘‘)’’
Ordering ::= ‘‘(’’ ( LP-expression )+ ‘‘)’’
LP-expression ::= ( Short-Name | Counting-Expression | Number )
Short-Name ::= a name (i.e.  Lisp-Symbol)
Counting-Expression ::= ‘‘(’’ ( ‘‘1o0 ’’ | ‘‘1 ’’ | ‘‘* ’’ ) Feature-Value ‘‘)’’

Feature-Equations ::= ‘‘(’’ ( Feature-Equation )* ‘‘)’’
Feature-Equation ::= ‘‘(’’ Feature-Term ( Feature-Term | Feature-Value ) ‘‘)’’
Feature-Term ::= ‘‘(’’ Tree-Address ( ‘‘top’’ | ‘‘bottom’’ ) Feature-Path ‘‘)’’
```

```
Tree-Address ::= ``('' ( Number `` '' )* ``)''
Feature-Path ::= ( Feature `` '' )*
Feature ::= ``mode'' | ``tense'' | ``agr'' | ``passive'' | ...
Feature-Value ::= ``+'' | ``-'' | ``acc'' | ``nom'' | ``3rd'' | ...
```

7.5.4 Other TAG Systems

At the University 7, Paris, a French grammar is being developed, based on the XTAG system. Other work on grammar development with TAGs is conducted in Brazil and India. We are currently inquiring about this work. Furthermore, as part of Pleuk (2.5.4, 3.5.4), there is a small TAG system called vN-TAG. The TAG-GEN system is also incorporated into a number of larger systems.

It is expected that all of these systems will be able to translate from the proposed common specification language into their specific representations.

7.5.5 Common Specification Language

Based on the explicit descriptions of XTAG and VM-GEN, we propose a common specification language (CSL) for TAGs which subsumes both descriptions. Since the only important difference is the introduction of linearization rules in VM-GEN, most parts of a translation from the CSL into a specific representation are straightforward. The linearizations, however, can, in principle, be compiled out, thereby making possible a translation from VM-GEN (or a CSL that includes linearization rules) to XTAG. Conversely, an XTAG grammar can be translated into an VM-GEN grammar with trivial (i.e. no) linearization rules.

```
File ::= Terminals Non-Terminals LP-Rules-Shorthands (Tree)*

Terminals ::= ``('' ( `` '' Terminal )* ``)''
Terminal ::= LISP-string

Non-Terminals ::= ``('' ( `` '' Non-Terminal )* ``)''
Non-Terminal ::= LISP-string

Tree ::= Tree-Name Tree-Structure Feature-Equations Display-Information
Tree-Name ::= LISP-String
Tree-Structure ::= ``('' Mother-Node ``('' ( Subtree-structure )* ``)'' ``)''
Mother-Node ::= Node
Subtree-Structure ::= Tree-Structure
Node ::= Node-Label Node-Identification-Type Node-Type ( Constraint )?
         ( LP-rules )? ( Node-Display-Information )?
Node-Label ::= ``('' Non-Terminal `` . ''  Subscript-String ``)'' |
               ``('' Terminal `` . "" )''
Subscript-String ::= LISP-String (possibly the empty string ``""'')
Node-Identification-Type ::= ( ``:tree-address'' | ``:node-label'' )
Node-Type ::= ``('' ( ``:substp'' | ``:headp'' | ``:footp'' ) ``)''
Constraint ::= ``('' ( ``:SA'' ( Tree-Name )* | ``:OA'' ( Tree-Name )+ |
                      ``:NA'' ) ``)''
Node-Display-Information ::=
     ``('' ( ``:display-feature?''  display-feature)?
           ( ``:shape'' shape'')?
           ( ``:default-style'' default-style)?
           ( ``:border-width'' border-width)?
           ( ``:white-space'' white-space)?
           ( ``:connector'' connector)?
           ( ``:subscript-style'' subscript-style)? ``)''
Feature-Equations ::= ``('' ( Feature-Equation )* ``)''
Feature-Equation ::= ``('' Feature-Term ( Feature-Term | Feature-Value ) ``)''
Feature-Term ::= ``('' Node-Identificator ( ``top'' | ``bottom'' ) Feature-Path ``)''
```

```
Node-Identificator ::= ( Node-Label | Tree-Address )
Tree-Address ::= ``('' ( Number `` '' )* ``)''
Feature-Path ::= ( Feature `` '' )*
Feature ::= ``mode'' | ``tense'' | ``agr'' | ``passive'' | ...
Feature-Value ::= ``+'' | ``-'' | ``acc'' | ``nom'' | ``3rd'' | ...

LP-Rules-Shorthands ::= ``('' ( LP-Rule-Shorthand )* ``)''
LP-Rule-Shorthand ::= ``('' Short-Name ( `` '' LP-expression )* ``)''
LP-Rules ::= ``(< '' ( LP-Rule )+ ``)''
LP-Rule ::= ``('' Rule-Name Test ( Ordering )+ ``)''
Rule-Name ::= Lisp-Symbol
Test ::= ``(test '' Test-expression ``)''
Test-Expression ::= ( And-Or-Expression | Feature-Value-Equation )
And-Or-Expression ::= ``('' ( ``and '' | ``or '' ) ( Test-Expression )+ ``)''
Feature-Value-Equation ::= ``('' Number ``('' Feature-Path ``)''
                                 Feature-Value ``)''
Ordering ::= ``('' ( LP-expression )+ ``)''
LP-expression ::= ( Short-Name | Counting-Expression | Number )
Short-Name ::= a name (i.e.  Lisp-Symbol)
Counting-Expression ::= ``('' ( ``1o0 '' | ``1 '' | ``* '' ) Feature-Value ``)''
```

Appendix A

Exchange Formats

A.1 HPSG

A.1.1 A Sample HPSG Grammar in the EAGLES Format

The following grammar fragment is largely (but not entirely) a modification of the *Penn and Carpenter* grammar that is provided with the ALE 2.0.2 distribution [Carpenter and Penn, 1994]. In contrast to the Penn and Carpenter grammar the current grammar conforms to the newer version of HPSG [Pollard and Sag, 1994b] and follows the cross-classification of lexical entries given in [Pollard and Sag, 1987] (pp. 206). The encoding illustrates how various parts of the grammar: *type hierarchy, schemas, principles, lexical entries* and *lexical rules* should be encoded in the encoding format. The encoded grammar is highly readable, compact and retains full declarativity.

To ease readability we have divided the grammar into various subsections.

The inheritance hierarchy

```
sign > [word,non_word]
     approp [phon:phon,
             synsem:synsem,
             qstore:list(quant),
             qretr:list(quant)].

  phon approp [ortho:list(constant)].

case > [nom,acc].

cat approp [subcat:list(synsem),
            head:head,
            marking:marking].

c_inds approp [addressee:ref,
               speaker:ref,
               utt_loc:ref].

conx approp [backgr:list(psoa),
             c_inds:c_inds].

gend > [fem, masc, neut].

head > [func, subst].
```

```
    func > [det, mark]
         approp [spec:synsem].

    subst > [adj, noun, prep, reltvzr, verb]
          approp [prd:bool,
                  mod:mod_synsem].

      noun approp [case:case].
      prep approp [pform:pform].

      verb approp [aux:bool,
                   inv:bool,
                   vform:vform].

  mod_synsem > [synsem,none].
    synsem approp [loc:loc,
                   non_loc:non_loc].

  ind > [it, there, ref]
      approp [gen:gend,
              num:num,
              per:per].

  loc approp [cat:cat,
              cont:sem_obj,
              conx:conx].

  marking > [marked, unmarked].
    marked > [comp, conj].
      comp > [for, that].

  name > [kim,sandy].

  non_loc approp [inherited:non_loc_1,
                  to_bind:non_loc_1].
  non_loc_1 approp [que:list(npro),
                    rel:list(ref),
                    slash:list(loc)].
  num > [plur, sing].

  per > [first, second, third].

  pform > [to, of].

    non_word > [trace,phrase].

  vform > [bse, fin, ger, inf, pas, prp, psp].


%%%%
%% MACROS
%%%%

np(loc:(cat:(head:noun,
```

```
              subcat:[]),
       cont:index:Ind),
   Ind).

nbar(loc:(cat:(head:noun,
              subcat:[(@ detp)]),
       cont:Cont,
     Cont).

case(loc:cat:head:case:Case,Case).


s(loc:(cat:(head:verb,
            subcat:[]),
       cont:Proposition,
  Proposition).

vp(loc:(cat:(head:verb,
              subcat:[synsem]),
       cont:Proposition,
   Proposition).



detp(loc:cat:(head:det,
             subcat:[])).

empty_non_loc(
  synsem:non_loc:(inherited:(que:e_set,
                             rel:e_set,
                             slash:e_set),
                  to_bind:(que:e_set,
                           rel:e_set,
                           slash:e_set))).

empty_inher(
  synsem:non_loc:inherited:(que:e_set,
                            rel:e_set,
                            slash:e_set)).

ortho(phon:ortho:O, O).

num(synsem:loc:cont:index:num:Num, Num).
per(synsem:loc:cont:index:per:Per, Per).
gen(synsem:loc:cont:index:gen:Gend, Gend).
```

The HPSG schema types

```
%                                    HPSG Schemas
% ============================================================================
%                                     headed_struct
%                                          |
%                   ---------------------|----------------------------
%                  |              |                 |                  |
%            f_struct  head_mark_struct  head_adjn_struct  head_filler_struct
%                  |
%              ----|-------------------------------
%             |                   |                |
%         subj_struct         comp_struct   head_spec_struct
%             |                   |
%       -------|----------     ----|-----------
%      |                  |   |                |
% head_subj_struct head_subj_comp_struct  head_comp_struct
%
headed_struct >   [f_struct,
                   head_mark_struct,head_adjn_struct,head_filler_struct]
           approp [head_dtr:sign].

   f_struct  >  [subj_struct,
                 comp_struct,
                 head_spec_struct].

       subj_struct >    [head_subj_struct,head_subj_comp_struct]
                 approp [subj_dtr:sign].

       comp_struct >    [head_subj_comp_struct,head_comp_struct].
                 approp [comp_dtrs:list(sign)].

       head_spec_struct approp [spec_dtr:sign].

   head_mark_struct approp [mark_dtr:sign].
   head_adjn_struct approp [adjn_dtr:sign].
   head_filler_struct approp [filler_dtr:sign].
```

The HPSG schema definitions

```
schema :: head_subj(sign,
                    head_subj_struct,
                    [subj_dtr,head_dtr]
                   ).
schema :: head_comp(sign,
                    head_comp_struct,
                    [head_dtr,comp_dtrs]
                   ).
schema :: head_subj_comp(sign,
                         head_subj_comp_struct,
                         [head_dtr,subj_dtr,comp_dtrs]
                         ).
schema :: head_spec(sign,
                    head_spec_struct,
                    [spec_dtr,head_dtr]
                    ).
schema :: head_adjn(sign,
                    head_adjn_struct,
                    [head_dtr,adjn_dtr] %% Only allow adjunct to follow head
                    ).
schema :: head_filler(sign,
                      head_filler_struct,
                      [filler_dtr,head_dtr]
                     ).
```

The HPSG principles interface definitions

```
principles ::
[ head_subj_struct -> valence_principle_subj('MOTHER',head_dtr,[subj_dtr]),
  head_comp_struct -> valence_principle_comps('MOTHER',head_dtr,comp_dtrs),
  valence_principle_subj_comps('MOTHER',head_dtr,[subj_dtr|comp_dtrs]),
  valence_principle_spec('MOTHER',head_dtr,[spec_dtr]),

  ~head_mark_struct ->  marking_principle('MOTHER',head_dtr),
  marking_principle('MOTHER',mark_dtr),

  spec_principle(subj_dtr,head_dtr),  % applies to both head_subj_struct,
                                      % head_subj_comp_struct
  head_comp_struct ->  spec_principle2(comp_dtrs,head_dtr),
  spec_principle(mark_dtr,head_dtr),
  % spec_principle does not apply to the rest of the schemas

  head_subj_struct ->  semantics_principle('MOTHER',head_dtr,[subj_dtr]),
  head_comp_struct ->  semantics_principle('MOTHER',head_dtr,comp_dtrs),
  semantics_principle('MOTHER',head_dtr,[subj_dtr|comp_dtrs]),
  semantics_principle('MOTHER',head_dtr,[mark_dtr]),

  subject_condition(subj_dtr,comp_dtrs),     % applies to head_subj_comp schema
  head_comp_struct  -> subject_condition(synsem:@hd(@subj('MOTHER')),comp_dtrs),

  head_subj_struct  -> nonlocal_feature_principle('MOTHER',head_dtr,[subj_dtr]),
  head_comp_struct  -> nonlocal_feature_principle('MOTHER',head_dtr,comp_dtrs),
  nonlocal_feature_principle('MOTHER',head_dtr,[subj_dtr|comp_dtrs]),
  nonlocal_feature_principle('MOTHER',head_dtr,[mark_dtr])
 ].
```

The HPSG priciples definitions

```
% valence_principle_xxx(Mother,Head-Daughter,Functional-Daughters)
%--------------------------------------------------------------
valence_principle_subj(@subj(MSub) & @comps(C) & @spr(Spr),
                       @subj(@append(MSub,FDtrSyns)) & @comps(C) & @spr(Spr),
                       @synsems(FDtrSyns)).

valence_principle_comps(@comps([]) & @subj(S) & @spr(Spr),
                       @comps(@append(MSub,FDtrSyns)) & @subj(S) & @spr(Spr),
                       @synsems(FDtrSyns)).


valence_principle_subj_comps(@comps([]) & @subj([]) & @spr(Spr),
                       @comps(CompDtrSyns) & @subj([SubjSynsem]) & @spr(Spr),
                       @synsems([SubjSynsem|CompDtrSyns])).

valence_principle_spr(@spr(MSub) & @comps(C) & @subj(Subj),
                      @spr(@append(MSub,FDtrSyns)) & @comps(C) & @subj(Subj),
                      @synsems(FDtrSyns)).


% head_feature_principle(Mother,Head-Daughter)
%--------------------------------------------
head_feature_principle(@head(H),@head(H)).


% inv_minus_principle(Mother)
%--------------------------------
% parochial: if inv exists, it must be minus
inv_minus_principle(Mother) :-
   Mother = @head(H),
   case(H, [(inv:top, Mother = @inv(minus))]).


% marking_principle(Mother,Mark-Dtr)
%-----------------------------------------------------
% Mark-Dtr is marker-dtr, if any, o.w. head-dtr
marking_principle(@marking(Mark),@marking(Mark)).


% spec_principle(Spec-Dtr,Head-Dtr)
%-----------------------------------------------------
%  Spec-Dtr is either mark-dtr or first comp-dtr
spec_principle(@head(Head),@synsem(HSynsem)) :-
    case(Head, [(subst,true)], Head = @spec(HSynsem)).

spec_principle2([@head(Head)|top],@synsem(HSynsem)) :-
    case(Head, [(subst,true)], Head = @spec(HSynsem)).


% semantics_principle(Mother,Semantic-Head,Other-Dtrs)
%-----------------------------------------------------
% Semantic-Head is adjunct-dtr, if any, o.w. head-dtr
```

```
semantics_principle(@qstore(MQStore) & @qretr(MRetr) & @cont(MCont),
                    SHead & @cont(SCont), ODtrs) :-

  accumulate([SHead|ODtrs], qstore, DQStore),

  case(SCont, [(psoa, ( MCont = @nucleus(Nucl) & @quants(MQuants),
        SCont = @nucleus(Nucl) & @quants(SQuants),
                        set_difference(MQStore,DQStore,MRetr),
                        append(MRetr,SQuants,MQuants)
                       ))
               ],
                   SCont = MCont,
                   MRetr = [],
                   MQStore = DQStore
        ).



% subject_condition(Subj-Dtr,Comp-Dtrs)
%---------------------------------------------------------------
subject_condition(@inher_slash(SubjSlash),CompDtrs):-
    case(SubjSlash,
           [(nelist, set_member(nelist, @accumulate(CompDtrs, inher_slash))],
         true).


% nonlocal_feature_principle(Mother,Head-Dtr,Other-Dtrs)
%---------------------------------------------------------------
nonlocal_feature_principle(@inher_slash(MISlash) &
                           @inher_que(MIQue) & @inher_rel(MIRel),
                           HeadDtr & @to_bind_slash(HTSlash) &
                                     @to_bind_que(HTQue) &
                                     @to_bind_rel(HTRel),
                           ODtrs):-
  accumulate([HeadDtr|ODtrs], inher_slash, DISlash),
  accumulate([HeadDtr|ODtrs], inher_que, DIQue),
  accumulate([HeadDtr|ODtrs], inher_rel, DIRel),
  set_difference(HTSlash,DISlash,MISlash),
  set_difference(HTQue,DIQue,MIQue),
  set_difference(HTRel,DIRel,MIRel).


% single_rel_constraint(Sign)
%-------------------------------
% parochial: Rel set can't have more than one element
% enforced on words and trace as type constraint; enforced on phrases as
%  procedural attachment to rules

single_rel_constraint(@inher_rel(Rel)):-
   case(Rel, [(nelist, Rel = [_])]).

% clausal_rel_prohibition(Sign)
%---------------------------------
% parochial: Sentences must have empty Rel set
```

```
clausal_rel_prohibition(@inher_rel(Rel) & @head(Head) &
                        @subj(Subjs) & @comp(Comps) & @spr(Sprs)):-
   case(Head, [verb => case(Subj,[[]
  => case(Comps,[[]
 => case(Sprs,[ [] => Rel=[] ])])])]).


% conx_consistency_principle(Mother,Dtrs)
%-------------------------------------------
conx_consistency_principle(@background(MBackgr), Dtrs):-
      accumulate(Dtrs, background, MBackgr).

% deictic_cindices_principle(Mother,Dtrs)
%--------------------------------------------
deictic_cindices_principle(@c_inds(MCinds),Dtrs):-
      maplist(Dtrs, c_inds, [MCinds]).

% inv_plus_principle(Mother)
%---------------------------------
% parochial: if inv exists, it must be plus
inv_plus_principle(Mother) :-
   Mother = @head(H),
   case(H, [(inv:top, Mother = @inv(plus))]).
```

The HPSG semantic types

```
%% Semantic Typology goes here
  qfpsoa > [property, un_relation, bin_relation, tri_relation,control_qfpsoa].
    property > [gender,nom_prop]
             approp [inst:ref].
      gender > [human,neuter].
        human > [female,male].

      nom_prop > [book,red,difficult].

    un_relation > [walk,run].
      walk approp [walker:ref].
      run approp [runner:ref].

    bin_relation > [see,hit,naming,composed_of,possess].
      see approp [seer:ref,
                  seen:ref].
      hit approp [hitter:ref,
                  hittee:ref].
      naming approp [bearer:ref,
                     name:name].
      composed_of approp [composite:ref,
                          composition:list(ref)].
      possess approp [possessor:ref,
                      possessed:ref].

    tri_relation > [sell,give].
      sell approp [seller:ref,
                   buyer:ref,
                   sold:ref].
      give approp [giver:ref,
                   given:ref,
                   gift:ref].

    control_qfpsoa > [trying, tending, believing, persuading, bothering]
                   approp [soa_arg:psoa].
      trying approp [tryer:ref].
      persuading approp [persuader:ref,
                         persuaded:ref].
      believing approp [believer:ref].
      bothering approp [bothered:ref].

  sem_det > [forall,exists,the].

  sem_obj > [nom_obj, psoa, quant].
    nom_obj > [npro, pron]
            approp [index:ind,
                    restr:list(psoa)].
      pron > [ana, ppro].
        ana > [recp, refl].
    quant approp [det:sem_det,
                  restind:npro].
    psoa approp [quants:list(quant),
```

```
nucleus:qfpsoa].
```

The HPSG lexical hierarchy

```
%% Start of hierarchical lexicon
%% Adapted from [Pollard and Sag, 87] pp. 206

open_world(lexical_sign). %% Declare that lexical signs are to interpreted
                          %%  in the open-world mode
sign > [lexical_sign].    %% Inherit from sign

lexical_sign > [major_lexical_sign, minor_lexical_sign].

major_lexical_sign > [sign_head, sign_subcat].

sign_head > [noun_sign, verb_sign, adjective_sign, prep_sign].
   noun_sign > [saturated, common_noun_sign]
         def
           (synsem:loc:cat:(head:(noun &
                                  mod:none) &
                            subj:[] &
                            marking:unmarked) &
            (@ empty_inher) &
            qstore:[]).

   verb_sign > [main_sign, auxiliary].

    main_sign > [intrans_verb_sign, trans_verb_sign, ditrans_verb_sign,
                 intrans_raising_verb_sign, intrans_equi_verb_sign,
                 trans_raising_verb_sign, trans_equi_verb_sign]
       def
        (synsem:loc:cat:head:(verb &
                              mod:none &
                              aux:minus &
                              inv:minus)).

   auxiliary def
     (synsem:loc:cat:head:(verb,
                           mod:none,
                           aux:plus)).

sign_subcat > [saturated, unsaturated].

saturated > [proper_name_sign].
       def (synsem:loc:cat:(subj: [] &
                            comps: [] &
                            marking:unmarked) &
              cont:restr:[] &
              conx:backgr:[quants:[]]) &
           (@ empty_non_loc) &
           qstore:e_set).

unsaturated > [common_noun_sign, unsat_non_nom].

common_noun_sign def
 (synsem:loc:(cat:(spr:[(@ detp)] &
```

```
                   subj:[] &
                   comps:[]) &
              cont:(npro &
                    index:(ref & Ind) &
                    restr:[nucleus:inst:Ind &
                           quants:[]]) &
              conx:backgr:[]) &
  (@ empty_inher) &
  qstore:[].

unsat_non_nom > [strict_intrans, intrans_control, trans].
   strict_intrans > [intrans_verb_sign].

strict_intrans def
 (synsem:loc:(cat:(subj:[synsem] &
                   comps:[] &
                   spr:[] &
                   marking:unmarked) &
              cont:quants:[] &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).

intrans_control > [intrans_raising_sign, intrans_equi_sign].
   intrans_raising_sign [intrans_raising_verb_sign].
   intrans_equi_sign [intrans_equi_verb_sign].

intrans_equi_sign def
 (synsem:loc:(cat:(head:(verb &
                         mod:none &
                         aux:minus &
                         inv:minus) &
                   subj:[(@np(Ind1))] &
                   comps:[(@vp(VCont) &
                           @subj(@np(Ind1)))] &
                   spr:[] &
                   marking:unmarked) &
              cont:(nucleus:soa_arg:VCont &
                    quants:[]) &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).

intrans_raising_sign def
 (synsem:loc:(cat:(head:(verb &
                         mod:none &
                         aux:minus &
                         inv:minus) &
                   subj:[X] &
                   comps:[(@vp(VCont) &
                           @subj(X)] &
                   spr:[] &
                   marking:unmarked) &
              cont:(nucleus:soa_arg:VCont &
```

```
                   quants:[]) &
             conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).

trans > [strict_trans, ditrans, trans_control].
    strict_trans > [strict_trans_verb_sign]
    ditrans > [ditrans_verb_sign]

strict_trans def
  (synsem:loc:(cat:(subj:[synsem] &
                   comps:[synsem] &
                   spr:[] &
                   marking:unmarked) &     % nonfinite forms
             cont:quants:[] &
             conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).

ditrans def
  (synsem:loc:(cat:(subj:[synsem] &
                   comps:[synsem,synsem] &
                   spr:[] &
                   marking:unmarked) &
              cont:quants:[] &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).

trans_control > [trans_raising, trans_equi].
    trans_raising > [trans_raising_verb_sign]
    trans_equi > [trans_equi_verb_sign]

trans_raising def
  (synsem:loc:(cat:(subj:[@np(top)] &
                   comps:[(X & @np(top)),
                          (@vp(VCont) &
                           @subj(X))] &
                   spr:[] &
                   marking:unmarked) &     % nonfinite forms
              cont:(nucleus:soa_arg:VCont &
                    quants:[]) &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).

trans_equi def
  (synsem:loc:(cat:(subj:[@np(top)] &
                   comps:[(@np(Ind1)),
                          (@vp(VCont) &
                           @subj(@np(Ind1)))] &
                   spr:[] &
                   marking:unmarked) &     % nonfinite forms
              cont:(nucleus:soa_arg:VCont &
```

```
                    quants:[]) &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[]).
```

The lexical entries

```
% ----------------------------------------------------------------------------
% Verbs
% -----------------------------------------------------------------------------
% Intransitive Verbs
% -----------------------------------------------------------------------------
lex(@intrans_verb_sign @
    @ortho(O) & @vform(VForm) &
    @verb_np_subj(Subj) &
    @verb_sem(Nuc)) :-
   intrans_verb(O,VForm,Subj,Nuc).

verb_np_subj((Ind, Case, Num, Per, Gen),
              @subj([@np(Ind) & @case(Case) &
                     @num(Num) & @per(Per) & @gen(Gen)])).

verb_sem(Sem, @cont_nucleus(Sem)).  %% @verb_sem(Sem) is same
                                    %% as @cont_nucleus(Sem)

%% intrans_verb(ORTHO, VFORM, PRD, (IND, CASE, NUM, PER, GEN), NUCLEUS).

intrans_verb(walk, bse, top, (I, top, top, top, top), walker:I).

% Transitive Verbs
% -----------------------------------------------------------------------------
lex(@trans_verb_sign @
    @ortho(O) & @vform(VForm) &
    @verb_np_subj(Subj) &
    @verb_obj1(Obj1) &
    @verb_sem(Nuc)) :-
   trans_verb(O,VForm,Subj,Obj1,Nuc).


verb_np_obj1((Ind,Case),synsem:loc:cat:comps:[@index(Ind) & @case(Case)|top]).

verb_obj1(Obj,synsem:loc:cat:comps:[Obj | top]).

%% trans_verb(ORTHO, VFORM, PRD, (IND1, CASE, NUM, PER, GEN),
                                 (IND2, CASE), NUCLEUS).

trans_verb(see, bse, top, (I1, top, top, top, top),
                          (I2, acc), seer:I1 & seen:I2).


% Ditransitive Verbs
% -----------------------------------------------------------------------------
lex(@ditrans_verb_sign &
    @ortho(O) & @vform(VForm) &
    @verb_np_subj(Subj) &
    @verb_np_obj1(Obj1) &
    @verb_np_obj2(Obj2) &
    @verb_sem(Nuc)) :-
 ditrans_verb(O,VForm,Subj,Obj1,Obj2,Nuc).
```

```
verb_np_obj2((Ind,Case),synsem:loc:cat:comps:[top,@index(Ind) & @case(Case)|top]).
verb_obj2(Obj2,synsem:loc:cat:comps:[top, Obj2 | top]).

ditrans_verb(give, bse, top, (I1, top, top, top, top),
                             (I2, acc),
                             (I3, acc), giver:I1 & given:I2 & gift:I3).

% bother  - That Bill liked Sandy bothered Kim.
% ------------------------------------------------------------------------------
lex_entry(@trans_verb_sign &
          @ortho(bother) &
          @vform(bse) &
          @verb_np_obj1((I2,acc)) &
          @subj([@s(Cont) & @marking(comp)]) &
          @verb_sem(bothered:Ind2 &
                    soa_arg:Cont)).



% Control Verbs
% ------------------------------------------------------------------------------
% Equi Verbs
% ------------------------------------------------------------------------------
%%
lex(@intrans_equi_verb_sign &
    @ortho(O) &
    @vform(VForm) & @prd(PRD) &
    @verb_subj(Subj) &
    @verb_obj1(@vp(SOA) & @vform(VForm)) & @verb_sem(Sem)):-
  equi_verb(O, VForm, PRD, 1, Subj, (SOA,VForm), Sem).
%% @vp(SOA) above is probably of the wrong type - (skm)


%%   equi_verb(ORTHO, VFORM, PRD, EQUI_ARG_NO,
%%                                SUBJ(INDEX, CASE, PER, NUM, GEN),
%%                                OBJ0, ... , OBJn,
%%                                VPARG, VSem).

equi_verb(try, bse, top, 1, (I, top, top, top, top),
                            (SOA, inf), tryer:I & soa_arg:SOA).


lex(@trans_equi_verb_sign &
    @ortho(O) &
    @vform(VForm) & @prd(PRD) &
    @verb_subj(Subj) &
    @verb_np_obj1(Obj1) &
    @verb_obj2(@vp(SOA) & @vform(VForm)) & @verb_sem(Sem)):-
  equi_verb(O, VForm, PRD, 2, Subj, Obj1, (SOA,VForm), Sem).

equi_verb(persuade, bse, top, 2, (I1, top, top, top, top),
                                 (I2, acc),
                                 (SOA, inf), persuader:I1 &
```

```
                                              persuadee:I2 &
                                              soa_arg:SOA).

% Raising Verbs
% ----------------------------------------------------------------------------
%%

lex(@intrans_raising_verb_sign &
    @ortho(O) &
    @vform(Vform) & @prd(PRD) &
    @verb_subj(Subj) &
    @verb_obj1(@vp(SOA) & @vform(VForm)) &
    @verb_sem(Sem))):-
  raising_verb(O,Vform,PRD,1,NP,VPArg,Sem).

% Subject Raising Verbs
% ------------------------------------------------------------
%
%%raising_verb(ORTHO, VFORM, PRD, CONTROLLED_ARG_NO,
%%                                SUBJ(INDEX, CASE, PER, NUM, GEN),
%%                                OBJ0, ... , OBJn,
%%                                VPARG, VSem).

raising_verb(tend, bse, top, 1, (I, top, top, top, top),
                                   (SOA, inf), soa_arg:SOA).

% Object Raising Verbs
% ------------------------------------------------------------
%

lex(@trans_raising_verb_sign &
    @ortho(O) &
    @vform(Vform) & @prd(PRD) &
    @verb_subj(Subj) &
    @verb_np_obj1(Obj1) &
    @verb_obj2(@vp(SOA) & @vform(VForm)) &
    @verb_sem(Sem))):-
 raising_verb(O,Vform,PRD,2,Subj,Obj1,(SOA,VForm),Sem).

raising_verb(believe,bse,top,2,(I1, nom, top, top, top),(I2,acc),
                               (SOA, inf), believer: I1 & soa_arg:SOA).



%% Personal Pronouns
% ---------------------------------------------------------------------------

lexical_sign > [per_pron_sign].

per_pron_sign def
              (synsem:loc:(cat:(head:(noun &
                                      case:nom &
                                      mod:none) &
                                subj: [] &
```

```
                                comps: [] &
                                marking:unmarked) &
                           cont:(ppro &
                                 index:ref)) &
               (@ empty_non_loc) &
               qstore:[]).

restr_nucleus(Nucleus,Sign):-
         case(Nucleus, [([],Sign = synsem:loc:cont:restr:[])],
                       Sign = synsem:loc:cont:restr:[nucleus:Nucleus &
                                                     quants:[]]).

background_nucleus(Nucleus,Sign):-
         case(Nucleus, [([],Sign = synsem:loc:conx:background:[])],
                       Sign = synsem:loc:conx:background:[nucleus:Nucleus &
                                                          quants:[]]).


%% Some closed class lexical entries are to be entered as specified by HPSG.
%%
% Auxiliaries
% -------------------------------------------------------------------------------

lex(@ortho(can) &
    synsem:loc:(cat:(head:(verb &
                           mod:none &
                           vform:fin &
                           aux:plus) &
                     subj:[(NP & @ np(_) & @ case(nom))],
                     comps:[(@ vp(Prop) &
                             loc:cat:(head:vform:bse &
                                      subj:[NP] &
                                      comps:[]))] &
                     marking:unmarked)  &
              cont:Prop &
              conx:backgr:e_set) &
  (@ empty_non_loc) &
  qstore:e_set).

lex(@ortho(to) &
    synsem:loc:(cat:(head:(verb &
                           mod:none &
                           vform:inf &
                           aux:plus &
                           inv:minus) &
                     subj:[(NP & @ np(_))] &
                     comps:[(@ vp(Prop) &
                             loc:cat:(head:vform:bse &
                                      subj:[NP] &
                                      comps:[]))] &
                     marking:unmarked) &
                cont:Prop &
                conx:backgr:e_set) &
   (@ empty_non_loc) &
```

```
   qstore:e_set).

lex(@ortho(be) &
    synsem:loc:(cat:(head:(verb &
                           mod:none &
                           vform:bse &
                           aux:plus) &
                     subj:[(NP & @ np(_))],
                     comps:[(loc:(cat:(head:(vform:pas &
                                             aux:minus &
                                             prd:plus) &
                                       subj:[NP] &
                                       comps:[]) &
                                   cont:Cont))] &
                     marking:unmarked) &
              cont:Cont &
              conx:backgr:e_set) &
   (@ empty_non_loc) &
   qstore:e_set).

% Complementizers
% ------------------------------------------------------------------------------

lex(@ortho(that) &
    synsem:loc:(cat:(head:(mark &
                           spec:(@ s(_) &
                                 loc:cat:(head:vform:(fin_or_bse) &
                                          marking:unmarked))) &
                     subj:[] &
                     comps:[] &
                     marking:that) &
                conx:backgr:e_set) &
   (@ empty_non_loc) &
   qstore:e_set).


% Possessive Pronouns
% -----------------------------------------------------------------------------

lex(@ortho(my) &
    synsem:loc:(cat:(head:(det &
                           spec:(@ nbar((index:Ind &
                                         restr:Restr)))) &
                     subj:[] &
                     comps:[] &
                     marking:unmarked) &
                cont:(index:(Ind2 &
                             per:first &
                             num:sing) &
                      restr:e_set) &
                conx:(c_inds:speaker:Ind2 &
                      backgr:e_set)) &
   (@ empty_non_loc) &
   qstore:[det:the &
```

```
          restind:(index:Ind &
                   restr:[(nucleus:(possessor:Ind2 &
                                    possessed:Ind) &
                          quants:[]) | Restr])]).


% Possessive Clitic
% -----------------------------------------------------------------------------

lex(@ortho(is) &
    synsem:loc:(cat:(head:(det &
                          spec:(@ nbar((index:Ind &
                                        restr:Restr)))) &
                    subj:[(@ np(Ind2) &
                           loc:cont:(npro & NPCont))] &
                    comps:[] &
                    marking:unmarked) &
             cont:NPCont &
             conx:backgr:e_set) &
  (@ empty_non_loc) &
  qstore:[(det:the &
          restind:(index:Ind &
                   restr:[nucleus:(possessor:Ind2 &
                                   possessed:Ind) &
                          quants:[] | Restr]))]).



% exceptions: lexical entries

lex(@ortho(is) &                 % irregular auxiliary
    synsem:loc:(cat:(head:(verb &
                          mod:none &
                          vform:fin &
                          aux:plus) &
                    subj:[(NP & @ np((per:third &num:sing)) & @ case(nom))] &
                    comps:[(loc:(cat:(head:(vform:pas &
                                            aux:minus & % no "is been"
                                            prd:plus) &
                                      subj:[NP] &
                                      comps:[]) &
                                 cont:Cont))] &
                    marking:unmarked) &
             cont:Cont &
             conx:backgr:e_set) &
  (@ empty_non_loc) &
  qstore:e_set).


%% Expletives
% -----------------------------------------------------------------------------

lexical_sign > [expletive_sign].
```

```
expletive_sign def
  (synsem:loc:(cat:(head:(noun &
                          mod:none) &
                   subj:[] &
                   comps:[]) &
              cont:(ppro &
                    restr:e_set) &
              conx:backgr:e_set) &
  (@ empty_non_loc) &
  qstore:e_set ).

%% Relative Pronouns
% ----------------------------------------------------------------------------

lexical_sign > [rel_pron_sign].

rel_pron_sign def
  (synsem:(loc:(cat:(head:(noun &
                           mod:none) &
                    subj:[] &
                    comps:[]) &
               cont:(npro &
                     index:Ind &
                     restr:[])) &
          non_loc:(inherited:(que:[] &
                              rel:[Ind] &
                              slash:[]) &
                   to_bind:(que:[] &
                            rel:[] &
                            slash:[]))) &
  qstore:[]).

%% Proper Names
% ----------------------------------------------------------------------------

lexical_sign > [proper_name_sign].

proper_name_sign def
  (synsem:loc:(cont:(index:(ref & Ind) &
                     restr:[]) &
               conx:backgr:[nucleus:(naming &
                                     bearer:Ind) &
                            quants:[]]) &
  (@ empty_non_loc) &
  qstore:e_set).


name(Name, synsem:loc:conx:backgr:[nucleus:name:Name]).



%% Quantificational Determiners
% ----------------------------------------------------------------------------
```

```
lexical_sign > [quant_det_sign].

quant_det_sign def
 (synsem:loc:(cat:(head:(det &
                         spec:(@ nbar(NPCont))) &
                   subj:[] &
                   comps:[] &
                   marking:unmarked) &
              cont:(GQ &
                    restind:NPCont) &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[GQ]).

det(Det,synsem:loc:cont:det:Det).

%% Attributive Adjective
% ----------------------------------------------------------------------------

lexical_sign > [attributive_adj_sign].

attributive_adj_sign def
 (synsem:loc:(cat:(head:(adj &
                         prd:minus &
                         mod:(@ nbar((index:Ind &
                                      restr:Restrs)))) &
                   subj:[] &
                   comps:[] &
                   marking:unmarked) &
              cont:(index:Ind &
                    restr:[(nucleus:inst:Ind &
                            quants:[]) | Restrs] &
              conx:backgr:[]) &
  (@ empty_non_loc) &
  qstore:[].


adj_sem(Sem, @restr_nucleus(Sem)). %%  @adj_sem(Sem) is same as
                                   %%  @restr_nucleus(Sem)

% Personal Pronouns
% -----------------------------------------------------------------------------

lex(@per_pron_sign &
    @ortho(O) &
    @per(Per) & @num(Num) & @gen(Gend) &
    @index(I) & @restr_nucleus(R) &
    @background_nucleus(N) & @cinds(CInds)) :-
  p_pron(O,Per,Num,Gend,(I,R,N,CInds)).


%% p_pron(ORTHO, PERSON, NUMBER, GENDER, CASE,
%%                       (INDEX,RESTR|NUCLEUS,BACKGROUND|NUCLEUS,C-INDS)).
```

```
p_pron(she,third,sing,fem,nom,(I, [], female & inst:I, top)).

p_pron(he,third,sing,masc,nom,(I, [], male & inst:I, top)).

p_pron(her,third,sing,fem,acc,(I, [], female & inst:I, top)).

p_pron(him,third,sing,masc,acc,(I, [], male & inst:I, top)).

p_pron(it,third,sing,neut,top,(I, [], neuter & inst:I, top)).

p_pron(i,first,sing,top,nom,(I, [], [], speaker:I)).

p_pron(me,first,sing,top,acc,(I, [], [], speaker:I).

p_pron(we,first,plur,neut,top,(I,composite:I &
                                 composition:[I2,I3], [],  speaker:I2 &
                                                           addressee:I3)).

p_pron(us,first,plur,neut,acc,(I,composite:I &
                                 composition:[I2,I3], [],  speaker:I2 &
                                                           addressee:I3)).

p_pron(you,second,sing,top,top,(I, [], [], addressee:I)).

p_pron(they,third,plur,neut,nom,(I, [], neuter & inst:I, top)).

p_pron(them,third,plur,neut,acc,(I, [], neuter & inst:I, top)).


% Expletive Pronouns
% ----------------------------------------------------------------------------

%% expletive(ORTHO,PERSON,NUMBER,GENDER,INDEX).

lex(@ortho(O) & @per(Per) & @num(Num) & @gen(Gen) & @ind(I)
    @expletive_sign):-
     expletive(O,Per,Num,Gen,I).


expletive(there,third,top,top,there).

expletive(it,third,sing,top,it).

% Relative Pronouns
% ----------------------------------------------------------------------------

lex(@rel_pron_sign &
    @ortho(O) & @case(C) & @background_nucleus(Nucleus)):-
     rel_pron(O,C,Nucleus).

rel_pron(who,nom,human & inst:I).

rel_pron(whom,acc,human & inst:I).
```

```
rel_pron(that,nom,[]).

% Proper Names
% ---------------------------------------------------------------------------

lex(@proper_name_sign & ortho(Name) & @name(Name)):-
       proper_name(Name).

proper_name(kim).

proper_name(sandy).


% Common Nouns
% ---------------------------------------------------------------------------

lex(@common_noun_sign &
                @ortho(O) &
                @per(Per) & @num(Num) & @gen(Gen) &
                @background_nucleus(N) ):-
        common_noun(O,Per,Num,Gen,N).


common_noun(book,third,sing,neut,book).

common_noun(person,third,sing,masc_fem,human).

% Quantificational Determiners
% ---------------------------------------------------------------------------

lex(@quant_det_sign & @ortho(O) & @det(D)):-
     quant_det(O,D).

quant_det(every,forall).

quant_det(a,exists).

quant_det(the,the).

% Adjectives
% ---------------------------------------------------------------------------

lex(@attributive_adj_sign & @ortho(O) & @adj_sem(Sem)):-
           attributive_adj(O,Sem).

attributive_adj(red,red).
```

The lexical rules

```
% Lexical Rules
% ============================================================================
% Finite Verb Formation
% --------------------------------------------------------------------------

% regulars: lexical rule
lex_rule::
    pres_3sg(@ortho(NewO) &
     @vform(bse) &
             @aux(minus) &
     @subj_cat(np)
             ,
             @ortho(NewO) &
     @vform(fin) &
             @subj_case(nom) &
     @subj_per(third) &
     @subj_num(sing)
            ):-
            first_of([replace(suffix(O,[y],[i,e,s],NewO)),
                      replace(suffix(O,[e],[e,s],NewO))
                     ]).

lex_rule::
    pres_3sg(@vform(bse) &
             @aux(minus) &
     @subj_cat(np)
             ,
             @ortho(NewO) & @vform(fin) &
             @subj_case(nom) &
     @subj_per(first_second)
            ).

lex_rule::
    pres_3sg(@vform(bse) &
             @aux(minus) &
     @subj_cat(np)
             ,
             @ortho(NewO) & @vform(fin) &
             @subj_case(nom) &
     @subj_num(plur)
            ).


% Passive Formation
% --------------------------------------------------------------------------

lex_rule::
    pres_3sg(@ortho(O) &
     @vform(bse) &
             @aux(minus) &
     @subj([top]) &
             @comps([Obj1|Rest])
```

```
             ,
             @ortho(NewO) &
     @vform(pas) &
             @subj([Obj1]) &
     @subj_case(top) &
     @comps(Rest)
            ) :-
            first_of([replace(O,give,given,NewO),
      replace(O,see,seen,NewO),
      replace(suffix(O,[y],[i,e,d],NewO)),
                      replace(suffix(O,[e],[e,d],NewO)),
      append(X,[e,d],NewO)
                     ]).

% It-Extraposition
% ---------------------------------------------------------------------------
lex_rule::
    it_extraposition(
     @vform(bse) &
     @subj((S & @ s(_) & loc:cat:marking:comp)) &
             @comps(Rest)
             ,
             @subj((@ np(it))) &
     @comps(NewRest)
            ):-
      append(Rest,[S],NewRest).


lex_rule::
    it_extraposition2(
     @vform(bse) &
     @subj(Subj) &
             @comps(Comps)
             ,
             @subj((@ np(it))) &
     @comps([Subj|NewComps])
            ):-
      append(Prev,[(S & @ s(_) & loc:cat:marking:comp)|Rest],Comps),
      append(Rest,[S],NewRest),
      append(Prev,NewRest,NewComps).



lex(@ortho(seems) &
    synsem:loc:(cat:(head:(verb &
                       mod:none &
                       vform:fin &
                       aux:minus &
                       inv:minus) &
                 subj:[(@ np(it))] &
                 comps:[(@ s(SCont))] &
                 marking:unmarked) &
              cont:SCont &
              conx:backgr:e_set) &
```

```
  (@ empty_non_loc) &
  qstore:e_set).



% Subject Extraction
lex_rule ::
  selr(@comps(Compsin) &
       @non_loc((inherited:slash:[] &
                    to_bind:slash:[])),
       @comps(CompsOut) &
       @non_loc((inherited:slash:[Loc] &
                  to_bind:slash:[]))) :-
    subst((loc:(cat:(head:vform:fin &
                     subj:[] &
                     comps:[]) &
                cont:SCont)),
          (loc:(cat:(head:vform:fin &
                     subj:[(@ np(_) & @ case(nom) & loc:Loc)] &
                     comps:[]) &
                cont:SCont)),
          Compsin,
          Compsout).

subst(X,Y,L1,L2):-
    append(Prev,L1, [X|_]),
    length(Prev,N),
    N1 is N+1,
    nth(N1,L2,Y).

% Complement Extraction Lexical Rule
% ----------------------------------------------------------------------------

lex_rule ::
   celr((@comps(Compsin) &
         @nonloc((inherited:slash:[] &
                  to_bind:slash:[]))),
        (@comps(CompsOut) &
         @nonloc((inherited:slash:[Loc] &
                   to_bind:slash:[])))):-
    select((loc:Loc),Compsin,Compsout).
```

A.1.2 Description and Implementation Notes for Library Routines

The library routines can be grouped into 3 different modules:

- a *list* module for list manipulation
- a *setlist* module for list based set operations
- a *maplist* module for map procedures
- the *replace* operation for string manipulation

We provide some Prolog code mainly as an illustration of these might me implemented. We have something like this in mind for the list module:

```
%% List module
%% Created: 26 Oct 1995 (skm)

append(L1, L2, R):-
    (nonvar(L1); nonvar(R)),
    append1(L1, L2, R).

append(L1, L2, R):-
    error_msg([nl,
              'Insufficiently instantiated variables in :',
              append(L1, L2, R)]
             ),
    !,
    fail.

append1([], L, L).
append1([X|L1], L2, [X|R]):-
        append1(L1,L2,R).


select(X, L, R):-
     nonvar(L),
     select1(X, L, R).

select(X, L, R):-
    error_msg([nl,
              'uninstantiated variables in :',
              select(X, L, R)
              ]),
    !,
    fail.

select1(X, [X|R], R).
select1(X, [Y|L], [Y|R]):-
        select1(X,L,R).
```

For the above kind of code to work, we need to assume that the *append* predicate is called with sufficiently instantiated variables. Some of this can be done at compile time and the compiler may be able to re-order some of the goals in the right order.

A general principle for re-ordering goals is *to apply first those principles that do not contain any guarded constraints or library routines that require variable instantiation.* Thus the *head-feature principle* which just co-instantiates some variable should be called first.

For compilation into systems such as ALEP and ALE, the following needs to be kept in mind. We assume that *schemas* are compiled into PS-rules and a bottom-up parser is employed. We can then assume that HPSG principles are applied as *filters* on the new “mother nodes” generated by the parser. Principles which contain any guarded constraints that require instantiation of RHS items can be executed earlier with the assumption that the RHS items are sufficiently instantiated by the time these principles are invoked. Information from the lexicon to figure out which path-values are always lexically instantiated can be used for finer reordering. Some simple Prolog *mode* analysis techniques could be very useful here.

Alternatively for “proper” implementation of the *list* module within Prolog systems that support coroutining something similar to the following version in Sicstus 2.1.9 can be used:

```
%% List module
%% Created: 26 Oct 1995 (skm)

:- block append(-,?,?), append(?,?,-).
% Block if the first argument and
% the third argument is uninstantiated
append([], L, L).
append([X|L1], L2, [X|R]):-
        append(L1,L2,R).

:- block select(?,-,?).
% Block if the second argument is uninstantiated
select(X, [X|R], R).
select(X, [_|L], R):-
        select(X,L,R).
```

However, only systems such as ProFIT that directly compile typed feature terms into Sicstus Prolog will be able to use the delaying mechanism. By this we mean, it will be difficult to mirror the above delaying version of *append* in ALE since an uninstantiated *list* in ALE is still instantiated as the type *list* and hence not a Prolog variable. On the other hand, systems such as CUF directly support such delaying mechanisms.

Next we describe some of implementation ideas for the *maplist* and *setlist* modules. The code has not be tested at all and hence improvements will certainly be possible.

```
%%  maplist(+<List>, +Predicate, ?<Args>)
%%  calls Predicate with every element of <List> as first argument
%%  and Args as rest of arguments

maplist(L, P, As):-
        nonvar(L),
        nonvar(P),
        !,
        maplist1(L,P,As).

maplist(L, P, As):-
        error_msg([nl,
                   'Maplist called with uninstantiated variable :',
                   maplist(L, P, As)
                  ]),
        !,
        fail.

maplist1([X|L], P, As) :-
        Goal =.. [P,X|As],
```

```
        call(Goal),
        maplist1(L,P,As).

maplist1([], _, _).

%% SetList module
%% Created: 26 Oct 1995 (skm)

%%
%% set_member(?E, +<List>)
%% Returns true if E is either token identical to an element of List
%%    otherwise attempts to unify E with an element of List

set_member(E, Set):-
        nonvar(Set),
        !,
        set_member1(E,Set).

set_member(E, Set):-
        error_msg([nl,
                   'Set_member called with uninstantiated variable :',
                   set_member(E, Set)
                   ]),
        !,
        fail.


set_member1(E,Set):-
        vmember(E,Set,_),
        !.

set_member1(E,Set):-
        select(E,Set,_).   % Resort to list membership

% vmember(X, L, R)
% Check if X is already unified with some member of L
% R is L-[X]
vmember(_, L, _):- var(L), !, fail.
vmember(X, [X1|L], L):- X == X1, !. % for ALE use token_identical(X,X1)
vmember(X, [X1|T], [X1|R]) :-
        vmember(X,T,R).
%%
%% set_subset(?E, +<List>)
%%
set_subset(Sub, Set):-
        maplist(Sub, set_member, [Set]).
%%
%% set_union(+<Set1>,+<Set2>,?<Union>)
%% set_union(+<Set1>,-<Set2>,+<Union>)
%% set_union(-<Set1>,+<Set2>,+<Union>)
%% assumes that intantiated lists are closed

%% set_union(+<Set1>,+<Set2>,?<Union>)
set_union(S1, S2, U):-
```

```
        nonvar(S1),
        nonvar(S2),
        !,
        set_union1(S1, S2, U).

%% set_union(+<Set1>,-<Set2>,+<Union>)
set_union(S1, S2, U):-
        nonvar(S1),
        nonvar(U),
        var(S2),
        !,
        set_union2(S1, S2, U).

%% set_union(-<Set1>,+<Set2>,+<Union>)
set_union(S1, S2, U):-
        var(S1),
        nonvar(U),
        nonvar(S2),
        !,
        set_union2(S2, S1, U).

set_union(S1, S2, U):-
        error_msg([nl,'Uninstantiated set lists',set_union(S1,S2,U)]),
        !,
        fail.

set_union1(S1, S2, U):-
        maplist(S1,set_member,[U]),
        maplist(S2,set_member,[U]).

set_union2(S1, S2, U):-
        S1 = [X|S1R],
        ( vmember(X, U, UR)
             -> set_union2(S1R,S2,U)
            ;
                (   S2 = [X|S2R],
                    set_union2(S1R,S2R,UR)
                )
        ).
%%
%%  set_disjoint_union(S1, S2, DU)
%%
set_disjoint_union(S1, S2, DU):-
        set_union(S1,S2,DU),
        set_disjoint(S1,S2).
%%
%%  set_disjoint(S1, S2)
%%  Returns true of S1 and S2 have no token identical elements
%%
set_disjoint(S1,S2):-
        sort(S1,S1s),
        sort(S2,S2s),
        \+ (S1s == S2s).
```

```
%% For systems that provide inequations
%% the following code is better
%%
% set_disjoint([],_).
% set_disjoint(_, []):- !.
% set_disjoint([X|S1],S2):-
%       set_disjoint2(X,S2),
%       set_disjoint(S1,S2).
%
% set_disjoint2(_,[]).
% set_disjoint2(X,[Y|S2]):-
%       diff(X,Y),   % (X, (=\= Y)) in ALE
%       set_disjoint2(X,S2).


set_disjoint(S1,S2):-
        var(S1),
        var(S2),
        error_msg([nl,
                   'Uninstantiated set lists',
                  set_disjoint(S1,S2)
                  ]),
        !,
        fail.
```

Compilation of Guarded Constraints

We assume that for systems such as CUF which support delaying mechanisms each guarded constraint can be compiled into a CUF relational predicate (*sorts* in CUF terminology).

Consider the case of translating the *subject condition* repeated below into CUF:

```
subject_condition(Sub & non_loc:inherited:slash:Slash,Others):-
   case(Slash,
        [(nelist,set_member(non_loc:inherited:slash:nelist,Others))],
       true).
```

It should roughly correspond to something like the following:

```
subject_condition(Sub & non_loc:inherited:slash:Slash) :=
  Others &
  case1(Slash,Others & set_member(non_loc:inherited:slash:nelist),true).

wait(case1(set,_) -> _).  % Wait until first argument of case1
                          % is more instantiated than set
case1(nelist, Goal, _) :=
      true(Goal).

case1(~nelist, _, Else) :=
      true(Else).
```

However to compile into systems such as ALE we will have to assume that the context terms are sufficiently instantiated when the relational predicates are called. Again compile time checking would be helpful to detect potential cases where this does not happen and automatic re-ordering of goals would be needed.

With these assumptions, guarded constraints can be compiled straightforwardly into Prolog (and ALE) using cuts (!). A simple interpreter for guarded constraints can be written as:

```
%
% Simple interpreter for case statements
%
case(C, Gs, E):-
        nonvar(C),
        !,
        case1(C, Gs, E).

case(C, Gs, E):-
        error_msg([nl,
                  'Insufficiently instantiated context in case statement',
                   case(C,Gs,E)
                  ]).

case1(C, [(G,A)|R], E):-
        \+( \+(C = G)),    % Test
        !,                 % Commit
        call(A),
        case2(C,R).

case1(C, [_|R], E) :-
        case1(C,R,E).

case1(_, [], E):-
        call(E).


case2(C, [(G,A)|R]):-
        \+( \+(C = G)),
        !,
        call(A),
        case2(C,R).

case2(C, [_|R]) :-
        case2(C,R).

case2(_, []).
```

The above interpreter can then be partially evaluated against the actual calls to derive an efficient implementation.

Appendix B

List of Acronyms

ACL	Association for Computational Linguistics
ACM	Association for Computing Machinery
ALE	*Attribute Logic Engine*
ALEP	*Advanced Linguistic Engineering Platform*
ANTHEM	Advanced Natural Language Interface for Multilingual Text Generation in Health Care
ASL	Architectures for Speech and Language
ATN	*Augmented Transition Network*
BCPO	Bounded Complete Partial Order
BIM	Belgian Institute of Management
BNF	Backus-Naur-Form
BUP	Bottom-Up Parsing
CCG	*Combinatory Categorial Grammar*
CEC	Commission of the European Communities
CF	context-free
CFG	*Context-Free Grammar*
CG	*Categorial Grammar*
CHIP	Constraint Handling in Prolog
CL	Computational Linguistics
CLE	*Core Language Engine*
CLIM	Common Lisp Interface Manager
CLP	Constraint Logic Programming
CMU	Carnegie-Mellon University
CNF	conjunctive normal form
COLING	International Conference on Computational Linguistics
COSMA	Cooperative Schedule Management Agent
CRISTAL	Conceptual Retrieval of Information using Semantic Dictionary in Three Languages
CSL	Common Specification Language (for TAG)
CSLI	Center for the Study of Language and Information
CUF	*Comprehensive Unification Formalism*
CUG	*Categorial Unification Grammar*
DCG	*Definite Clause Grammar*
DELIS	Descriptive Lexical Specifications and tools for corpus-based lexicon building
DFG	Deutsche Forschungsgemeinschaft (German Research Foundation)

DFKI	Deutsches Forschungszentrum für Künstliche Intelligenz (German Research Center for Artificial Intelligence)
DG	*Dependency Grammar*
DISCO	A Dialogue System for Autonomous Coorperating Agents
DISCOURSE	
DNF	disjunctive normal form
DRT	Discourse Representation Theory
DS	Deep Structure
DTR	Daughter
DTRS	Daughters
DYANA	Dynamic Interpretation of Natural Language
EACL	European Chapter of the Association for Computational Linguistics
EAGLES	Expert Advisory Group on Language Engineering Standards
EC	European Community
ELSNET	European Network in Langauge and Speech
EN	English
END	
ENGCG	English Constraint Grammar
ENGTWOL	English Two-Level Morphology
ERG	English Resource Grammar
ET	Eurotra Tender
EU	European Union
FB-TAG	*Feature-Based Tree Adjoining Grammar*
FB-LTAG	*Feature-Based Lexicalised Tree Adjoining Grammar*
ProFIT	*Prolog with Features, Inheritance and Templates*
FS	Feature Structure
FTP	File Transfer Protocol
FWG	(EAGLES) Formalism Working Group
GB	*Government and Binding Theory*
GFP	greatest fixpoint semantics
GLB	greatest lower bound
GM	Graphics Manager (of Sicstus Prolog)
GNU	GNU's Not Unix
GPSG	*Generalized Phrase Structure Grammar*
GUI	Graphical User Interface
HCLP	Hierarchical Constraint Logic Programming
HMRS	Heidelberg Minimal Recursion Semantics
HPSG	*Head-Driven Phrase Structure Grammar*
IAI	Institut für Angewandte Informationswissenschaft (Institute for Applied Information Science)
IBM	International Business Machines
ICLP	International Conference in Logic Programming
ID	Immediate Dominance
IJCAI	International Joint Conference in Artificial Intelligence
ILPS	International Logic Programming Symposium
KAIST	Korean Advanced Institute of Science and Technology
KB	Kilobyte
KECU	thousand European Currency Units

KL-ONE	Knowledge Language 1
KONVENS	Konferenz für die Verarbeitung natürlicher Sprache
LALR	Lookahead Left-Right
LFG	*Lexical Functional Grammar*
LHS	left-hand side
LI	logical inferences
LIFE	Logic, Inheritance, Functions and Equations
LILOG	Linguistische und Logische Methoden für das Textverstehen
LISP	List Processing
LOGIN	Logic and Inheritance
LP	Linear Precedence
LR	
LRE	Linguistic Research and Engineering
LS-GRAM	Large-Scale Grammars
LTAG	Lexicalised Tree Adjoining Grammars
LUB	Least Upper Bound
MS	Master of Science
MT	Machine Translation
MULTEXT	Multilingual Text Tools and Corpora
NL	Natural Language
NLL	Natural Language Logic
NLP	Natural Language Processing
OLDT	Ordered selection strategy with Linear resolution for Definite clauses with Tabling
OSF	Open Software Foundation
OT	*Optimality Theory*
PARADICE	Parametrizable Discourse Core Engine
PARC	(XEROX) Palo Alto Research Center
PATR	Parsing and Translation
PF	Phonetic Form
PLEUK	(not an acronym)
PROLOG	Programming in Logic
PSG	Phrase Structure Grammar
RELATOR	Distributed European Linguistic Resource Repository
RGR	Reusability of Grammatical Resources
RHS	right-hand side
RTD	Research and Technology Development
SICS	Swedish Institute for Computer Scince
SLD	Linear Resolution with Selection Function for Definite Clauses
SNALE	Sharp New ALE
SNI	Siemens Nixdorf
SRI	Stanford Research Institute
STUF	Stuttgart Type Unification Formalism
TAG	Tree Adjoining Grammar
TAMIC	Transparent Access to Multiple Information for the Citizen
TDL	Type Description Language
TFL	Typed Feature Logic
TFS	Typed Feature Structures
TUG	Trace and Unification Grammar
UCG	Unification Categorial Grammar
UDRS	Underspecified Discourse Representation Structure
URL	Uniform Resource Locator

UTAG	Unification-Based Tree Adjoining Grammar
VM	Virtual Machine
WG	Working Group
XLE	Xerox Linguistic Environment
XTAG	X-based Tree Adjoing Grammar Development System

Bibliography

[Aarts and Trautwein, 1996] Erik Aarts and Kees Trautwein. Non-associative Lambek Categorial Grammar in Polynomial Time. *Mathematical Logic Quarterly*, 42(1):000–000, 1996.

[Aarts, 1994] Erik Aarts. Proving Theorems of the Lambek calculus of order 2 in Polynomial Time. *Studia Logica*, 53(3):373–387, 1994.

[Aarts, 1995] Erik Aarts. *Investigations in Logic, Language and Computation.* PhD thesis, Utrecht University, 1995.

[Abdel Kader Diagne, 1995] Hans-Ulrich Krieger Abdel Kader Diagne, Walter Kasper. Distributed parsing with hpsg grammars. In *Proceedings of the 4th International Workshop on Parsing Technologies, Prague*, 1995.

[Abeillé *et al.*, 1990] Anne Abeillé, Kathleen Bishop, Sharon Cote, and Yves Schabes. A Lexicalized Tree Adjoining Grammar for English. Technical Report MS-CIS-90-24, Department of Computer and Information Science, University of Pennsylvania, April 1990.

[Abeillé *et al.*, 1994] A. Abeillé, B. Daille, and A. Husson. Ftag an implemented tree adjoining grammar for parsing french sentences. In *3e Colloque International sur les Grammaries d'Arbres Adjoints (TAG+ 3), Technical Report TALANA-RT-94-01, TALANA, Université Paris*, Paris, France, 1994.

[Abeillé *et al.*, to appear] Anne Abeillé, Danièle Godard, Philip H. Miller, and Ivan A. Sag. French Bounded Dependencies. In Sergio Balari and Luca Dini, editors, *Romance in HPSG*. CSLI Publications, Stanford University, to appear.

[Abeillé, 1991] Anne Abeillé. *Une grammaire lexicalisée d'arbres adjoints pour le français.* PhD thesis, Université Paris 7. UFR de Linguistique. Laboratoire d'Automatique Documentaire et Linguistique., Paris. France., Janvier 1991.

[Ait-Kaci and Nasr, 1986a] H. Ait-Kaci and R. Nasr. A logical programming language with built-in inheritance. In *Proceedings of the 13th ACM Symposium an Principles of Programming Languages*, pages 219–228, 1986.

[Aït-Kaci and Nasr, 1986b] Hassan Aït-Kaci and Roger Nasr. LOGIN : A logic programming language with built-in inheritance. *The Journal of Logic Programming*, 3:185–215, 1986. Also appeared as MCC report AI-068-85.

[Aït-Kaci *et al.*, 1993] Hassan Aït-Kaci, Andreas Podelski, and Seth Copen Goldstein. Order-sorted feature theory unification. In Dale Miller, editor, *Proc. of the International Symposium on Logic Programming*, pages 506–524, Vancouver, October 1993. MIT Press.

[Aït-Kaci *et al.*, 1994] Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994.

[Ait-Kaci, 1984] H. Ait-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* PhD thesis, University of Pennsylvania, 1984.

[Ajduckiewicz, 1935] K. Ajduckiewicz. Die syntaktische konnexität. *Studia Philosophica*, 1:1–27, 1935.

[Alshawi and Crouch, 1992] H. Alshawi and R. Crouch. Monotonic semantic interpretation. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pages 32–39, 1992.

[Alshawi *et al.*, 1991] H. Alshawi, D.J. Arnold, R. Backofen, D.M. Carter, J. Lindop, K. Netter, S.G. Pulman, J. Tsujii, and H. Uszkoreit. Eurotra et6/1: Rule formalism and virtual machine design study (final report). Technical report, CEC, 1991.

[Alshawi, 1991] Hiyan Alshawi, editor. *The Core Language Engine.* MIT Press, 1991.

[Alshawi, 1992] H. Alshawi. *The Core Language Engine.* MIT Press, Cambridge,Mass., 1992.

[Amores *et al.*, 1994] J. G. Amores, J. F. Quesada, and D. Tapias. Traduccion automatica basada en el formalismo lfg con entrada y salida por voz. *Comunicaciones de Telefonica I+D*, 5 (2):132–148, 1994.

[Amores, 1992] G. Amores. Coordination, lexical transfer and article insertion in an lfg-based machine translation prototype. *The Georgetown Journal of Languages and Linguistics*, 3 (1):1–23, 1992.

[Aranovich *et al.*, 1994] Raul Aranovich, William Byrne, Susanne Preuss, and Martha Senturia, editors. *The Complementation of French Auxiliaries*, volume 13, Stanford University, 1994. CSLI Publications/SLA.

[Baader *et al.*, 1993] Franz Baader, Hans-Jürgen Bürckert, Bernhard Nebel, Werner Nutt, and Gert Smolka. On the expressivity of feature logics with negation, functional uncertainty, and sort equations. *Journal of Logic, Language and Information*, 2:1–18, 1993.

[Backofen and Smolka, 1995] Rolf Backofen and Gert Smolka. A complete and recursive feature theory. *Theoretical Computer Science*, 146(1–2):243–268, July 1995.

[Backofen and Treinen, 1994] Rolf Backofen and Ralf Treinen. How to win a game with features. In Jean-Pierre Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, Lecture Notes in Computer Science, vol. 845, pages 320–335, München, Germany, 7–9 September 1994. Springer-Verlag.

[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*, 4(1):5–39, 1995.

[Backofen, 1994a] Rolf Backofen. *Expressivity and Decidability of First-order Languages over Feature Trees.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1994.

[Backofen, 1994b] Rolf Backofen. Regular path expressions in feature logic. *Journal of Symbolic Computation*, 17:412–455, 1994.

[Backofen, 1995] Rolf Backofen. A complete axiomatization of a theory with feature and arity constraints. *Journal of Logic Programming*, 24:37–72, 1995.

[Balari, 1991] Sergio Balari. Why german is not a null-subject language. Technical Report 12, University of the Saarland, 1991.

[Bayer and Johnson, 1995] Sam Bayer and Mark Johnson. Features and agreement. In *Proceedings of the 33nd Annual Meeting of the ACL, Massachusetts Institute of Technology*, Cambridge, Mass., 1995. also to appear in 1995 ESSLLI Proceedings on the Workshop on Formal Grammar.

[Bayer, 1993] Sam Bayer. The coordination of unlike categories. Paper presented at the 1993 LSA, subsequent draft under revision at “Language”, 1993.

[Bayer, 1995] Sam Bayer. *Confessions of a Lapsed Neo-Davidsonian.* PhD thesis, Brown University, 1995.

[Beaven, 1990] John Beaven. A unification based treatment of spanish clitics. In Elisabet Engdahl, Mike Reape, Martin Mellor, and Richard Cooper, editors, *Parametric Variation in Germanic and Romance. Edinburgh Working Papers in Cognitive Science, 6.* Center for Cognitive Science, Edinburgh, 1990.

[Bech *et al.*, 1993a] A. Bech, N. Calzolari, U. Heid, E. Hinkelman, L. Lamel, B. Maegard, I. Trancoso, H. Uszkoreit, J. Villadsen, and J. Zeiliger. Language resources in europe. a survey. conducted by project relator for the european language resources association. Survey, 1993. The RELATOR project report is available as a postscript document from `http://de.relator.research.ec.org/l=en/project.mlhtml`.

[Bech *et al.*, 1993b] A. Bech, N. Calzolari, U. Heid, E. Hinkelman, L. Lamel, B. Maegard, I. Trancoso, H. Uszkoreit, J. Villadsen, and J. Zeiliger. Language resources in europe. a survey. conducted by project relator for the european language resources association. Survey, 1993. available as `http://de.relator.research.ec.org/survey.v1.ps`.

[Becker, 1993] Tilman Becker. *HyTAG: A new Type of Tree Adjoining Grammars for Hybrid Syntactic Representation of Free Word Order Languages.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1993.

[Becker, 1995] Tilman Becker. Patterns in metarules. submitted as chapter in a book, CSLI, Stanford, 1995.

[Bell and llegas, to appear] Nuria Bell and Marta V. llegas. Controlled Subjects in Spanish. In Sergio Balari and Luca Dini, editors, *Romance in HPSG.* CSLI Publications, Stanford University, to appear.

[BIM, 1992] BIM. The functional specifications and high-level design of alep 1.0. Technical report, CEC, 1992.

[BIM, 1993] BIM. Alep distributed architecture; integrating user-interface tools and applications in alep. Technical report, CEC, 1993. Version 1.0.

[Bird and Ellison, 1994] Steven Bird and Mark Ellison. One-Level Phonology: Autosegmental Representations and Rules as Finite Automata. *Journal of Computational Linguistics*, 20(1):55–90, 1994.

[Bird and Klein, 1994] Steven Bird and Ewan Klein. Phonological Analysis in Typed Feature Systems. *Journal of Computational Linguistics*, 20(3):455–491, 1994.

[Blackburn and Meyer-Viol, 1994] Patrick Blackburn and Wilfried Meyer-Viol. Linguistics, logic, and finite trees. *Bulletin of the IGPL*, 2:3–29, 1994.

[Blackburn and Spaan, 1993] Patrick Blackburn and Edith Spaan. A modal perspective on the computational complexity of attribute value grammar. *Journal of Logic, Language and Information*, 2:129–169, 1993.

[Blackburn *et al.*, 1993] Patrick Blackburn, Claire Gardent, and Wilfried Meyer-Viol. Talking about trees. In *Proceedings of the 6th Conference of the EACL*, pages 21–29. European Chapter of the Association for Computational Linguistics, 1993.

[Borsley, 1987a] Robert Borsley. A note on HPSG. *Bangor Papers in Linguistics*, 1, 1987.

[Borsley, 1987b] Robert Borsley. Subjects and complements in HPSG. CSLI Report 87-107, Stanford University, Stanford University, 1987.

[Borsley, 1989] Robert Borsley. An HPSG approach to welsh. *Journal of Linguistics*, 25:333–354, 1989.

[Böttcher, 1993] M. Böttcher. Disjunctions and inheritance in the context feature structure system. In *EACL93*, pages 54–60, 1993.

[Bouma and van Noord, 1993] Gosse Bouma and Gertjan van Noord. Head-driven parsing for lexicalist grammars: Experimental results. In *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics*, pages 71–80, Utrecht, The Netherlands, 1993.

[Bouma and van Noord, 1994a] Gosse Bouma and Gertjan van Noord. Constraint-based categorial grammar. In *Proceedings of the ACL*, Las Cruces, 1994. available by WWW: http://www.let.rug.nl/ vannoord/papers/.

[Bouma and van Noord, 1994b] Gosse Bouma and Gertjan van Noord. A lexicalist account of the Dutch verb cluster. In *CLIN 1993, Papers from the fourth CLIN Meeting 1993*, pages 19–34, Groningen, 1994. available by WWW: http://www.let.rug.nl/ vannoord/clin/Clin4/clin4.html.

[Bouma, 1990] Gosse Bouma. Defaults in unification grammar. In *ACL'90*, pages 165–172, Pittsbugh, Pa., 1990. Association for Computational Linguistics.

[Bouma, 1993] Gosse Bouma. *Nonmonotonicity and Categorial Unification Grammar*. PhD thesis, University of Groningen, 1993.

[Boves *et al.*, 1990] L. Boves, J. Landsbergen, R. Scha, and G. van Noord. Language and speech technology. Technical report, NWO Den Haag, 1990. Project plan for the NWO Priority Programme 'Language and Speech Technology'.

[Brachman and Schmolze, 1985] R.J. Brachman and J.G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[Buschauer *et al.*, 1991] Béla Buschauer, Peter Poller, Anne Schauder, and Karin Harbusch. Tree Adjoining Grammars mit Unifikation. Technical report, DFKI, Saarbrücken, 1991.

[Busemann and Harbusch, 1993] Stephan Busemann and Karin Harbusch, editors. *DFKI Workshop on Natural Language Systems: Re-usability and Modularity. Proceedings*, Document D-93-03, 1993.

[Busemann *et al.*, 1994] Stephan Busemann, Stephan Oepen, Elizabeth Hinkelman, Günter Neumann, and Hans Uszkoreit. COSMA–multi-participant NL interaction for appointment scheduling. Technical Report RR-94-34, DFKI, Saarbrücken, 1994.

[Calder, 1990] Jonathan H. R. Calder. *An Interpretation of Paradigmatic Morphology*. PhD thesis, University of Edinburgh, 1990.

[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, 1992a] Bob Carpenter. *The Logic of Typed Feature Structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1992.

[Carpenter, 1992b] Bob Carpenter. Quantifier scoping, 1992. Talk at the European Summer School in Language, Logic and Information, 1992, Lisboa.

[Carpenter, 1993a] B. Carpenter. Compiling typed attribute-value logic grammars. In *Third International Workshop on Parsing Technologies*, pages 39–48, 1993.

[Carpenter, 1993b] Bob Carpenter. Skeptical and credulous default unification with applications to templates and inheritance. In Ted Briscoe, Ann Copestake, and Valeria de Paiva, editors, *Inheritance, Defaults, and the Lexicon*, pages 13–37. Cambridge University Press, 1993.

[Carpenter, 1994] Bob Carpenter. Quantification and scoping: A deductive account. In *Proceedings of the 13th West Coast Conference on Formal Linguistics*, San Diego, 1994. CSLI Press.

[Carpenter, 1995] Bob Carpenter. *Lectures on Type-logical semantics*. MIT Press, 1995.

[Chanod and Tapanainen, 1995a] Jean-Pierre Chanod and Pasi Tapanainen. Creating a tagset, lexicon and guesser for a french tagger. In *Proceedings of the ACL SIGDAT Workshop*, pages 58 – 64, Dublin, 1995.

[Chanod and Tapanainen, 1995b] Jean-Pierre Chanod and Pasi Tapanainen. Tagging french – comparing a statistical and a constraint-based method. In *Proceedings of the 7th Conference of the European Chapter of the ACL*, pages 149 – 156, Dublin, 1995.

[Chomsky and Lasnik, 1991] N. Chomsky and H. Lasnik. Principles and parameters theory. In J. Jacobs et al., editor, *Syntax: an international handbook of contemporary research*. Walter de Gruyter, Berlin, December 1991.

[Chomsky, 1981] N. Chomsky. Lectures on government and binding. In *Studies in Generative Grammar*. Foris, 1981.

[Chomsky, 1991] N. Chomsky. Some notes on economy of derivation and representation. In R. Freiden, editor, *Principles and Parameters in Comparative Grammar*. Cambridge, MIT Press, 1991.

[Chomsky, 1993] N. Chomsky. A minimalist program for linguistic theory. In K. Hale and S. J. Keyser, editors, *The View from Building 20*. Cambridge, MIT Press, 1993.

[Copestake and Briscoe, 1992] A. Copestake and T. Briscoe. Lexical operations in a unification-based framework. In J. Pustejovsky and S. Bergler, editors, *Lexical Semantics and Knowledge Representation*, pages 101–120. Springer-Verlag, 1992.

[Covington, 1994] M. A. Covington. Gb theory as dependency grammar. Technical Report AI-1994-01, Artificial Inteligence Programs, University of Georgia, 1994.

[Cremers, 1993a] Crit Cremers. Coordination as a parsing problem. In K. Sikkel and A. Nijholt, editors, *Natural language parsing; Methods and Formalisms*, pages 139 – 145. University of Twente, 1993.

[Cremers, 1993b] Crit Cremers. *On parsing Coordination Categorially*. PhD thesis, Leiden University, 1993. HIL Dissertations 5.

[Dalrymple and Zaenen, 1995] Mary Dalrymple and Anne Zaenen, editors. *Formal Issues in Lexical-Functional Grammar*. CSLI Lecture Notes 13. Center for the Study of Language and Information, Stanford, CA, 1995.

[Dalrymple *et al.*, 1991] M. Dalrymple, S. M. Shieber, and F. C. N. Pereira. Ellipsis and higher-order unification. *Linguistics and Philosophy*, 14(4):399–452, 1991.

[Davis, 1995] Antony Davis. *Linking Theory and the Hierarchical Lexicon*. PhD thesis, Stanford University, 1995.

[Dekker and Hendriks, 1994] Paul Dekker and Herman Hendriks. Files in focus. In E. Engdahl, editor, *Integrating Information Structure into Constraint-Based and Categorial Approaches*, ESPRIT Basic Reserch Project 6852, Dynamic Interpretation of Natural Language. DYANA-2 Deliverable R1.3.B., pages 27–38. ILLC, Amsterdam, 1994.

[Devillers *et al.*, 1991] C. Devillers, L.D. Burnard, S.M. Hockey, and G. Gibeaux. Eurotra-6/3 text handling design study (final report). Technical report, CEC, 1991.

[Dini and Busa, 1994] Luca Dini and Federica Busa. Generative operations in a constraint-based grammar. In Harald Trost, editor, *KONVENS '94, Verarbeitung Natürlicher Sprache*. Informatik Xpress 6, Germany, 1994.

[Dini and Di Tomaso, 1995] Luca Dini and Vittorio Di Tomaso. Events and individual in italian dynamic locative expressions. In *Proceedings the 4th Conference on the Cognitive Science of Natural Language Processing*, 1995.

[Dini, to appear] Luca Dini. Null Complements and Unselective Binding. In Sergio Balari and Luca Dini, editors, *Romance in HPSG*. CSLI Publications, Stanford University, to appear.

[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, 1990] J. Dörre and A. Eisele. Feature logic with disjunctive unification. In H. Karlgren, editor, *Coling 90: Papers presented to the the Thirteenth International Conference on Computational Linguistics*, pages 100–105. Helsinki University, 1990.

[Dörre and Manandhar, 1995] Jochen Dörre and Suresh Manandhar. On constraint-based Lambek calculi. In J. Dörre, editor, *Computational Aspects of Constraint-based Linguistic Description III,* DYANA-*2 deliverable R1.2.C*. ESPRIT, Basic Research Project 6852, September 1995. To appear in Blackburn, de Rijke (eds.), Specifying Syntactic Structures, papers from the Logic, Structures, and Syntax workshop, Amsterdam, Sept. 1994.

[Dörre and Rounds, 1992] Jochen Dörre and William C. Rounds. On subsumption and semiunification in feature algebras. *Journal of Symbolic Computation*, 13(4):441–461, April 1992.

[Dörre *et al.*, 1995] Jochen Dörre, Esther König, and Dov Gabbay. Fibred semantics for feature-based grammar logic. *Journal of Logic, Language and Information. Special Issue on Language and Proof Theory*, 1995. to appear.

[Dörre, 1993] Jochen Dörre. *Feature-Logik und Semiunifcation*. PhD thesis, Universität Stuttgart, 1993. In German.

[Dowty, 1988] David Dowty. Type raising, function composition and non-constituent conjunction. In Oehrle, Bach, and Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 153–197. Reidel, Dordrecht, 1988.

[Emele and Zajac, 1990a] Martin Emele and Rémi Zajac. A fixed-point semantics for feature type systems. In Stéphane Kaplan and Mitsuhiro Okada, editors, *Conditional and Typed Rewriting Systems, 2nd International Workshop*, LNCS 516, pages 383–388, Montreal, Canada, June 11–14, 1990. Springer-Verlag.

[Emele and Zajac, 1990b] Martin Emele and Rémi Zajac. Typed unification grammars. In *Proc. of the 13^{th} International Conference on Computational Linguistics*, pages 293–298, Helsinki, Finland, 1990.

[Emms and Leiß, 1993] Martin Emms and Hans Leiß. Cut elimination in the second order lambek calculus. DYANA Deliverable R1.1.A, 1993.

[Emms, 1993a] Martin Emms. Extraction covering extensions of lambek calculus are not context free. In *Proceedings of the 9th Amsterdam Colloquium*, 1993.

[Emms, 1993b] Martin Emms. Parsing with polymorphism. In *Proceeding of the Sixth Annual Conference of the European Association for Computational Linguistics*, 1993.

[Emms, 1993c] Martin Emms. Some applications of categorial polymorphism. DYANA Deliverable R1.3.A, 1993.

[Emms, 1994a] Martin Emms. Completeness results for polymorphic lambek calculus. DYANA Deliverable R1.1.B, 1994. Available from ILLC, University of Amsterdam.

[Emms, 1994b] Martin Emms. Movement in labelled and polymorphic calculi. In *Proceedings of the Rome workshop in Categorial Grammar and Linear Logic 1993, special edition of the IGPL*, 1994. to appear.

[Emms, 1995] Martin Emms. An undecidability result for polymorphic lambek calculus. DYANA Deliverable R1.1.C, 1995. Available from ILLC, University of Amsterdam. (to appear in Proceedings of Tenth Amsterdam Colloquium), 1995.

[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 Lin guistics*, pages 201 – 208, Newark, Delaware, 1992.

[Erbach and Uszkoreit, 1990] Gregor Erbach and Hans Uszkoreit. Grammar engineering: Problems and prospects - report of the saarbrücken workshop on grammar engineering. Technical report, Universität des Saarlandes, Saarbrcken, July 1990 1990.

[Erbach *et al.*, 1995] G. Erbach, M. van der Kraan, S. Manandhar, H. Ruessink, W. Skut, and C. Thiersch. Extending unification formalisms. In *2nd Language Engineering Convention*, 1995.

[Erbach, 1994a] G. Erbach. Bottom-up Earley Deduction. In *COLING*, pages 796–802, 1994.

[Erbach, 1994b] G. Erbach. Multi-dimensional inheritance. In H. Trost, editor, *Proceedings of KONVENS '94*, pages 102–111. Springer, 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.

[Erbach, forthcoming] Gregor Erbach. *Bottom-Up Earley Deduction for Preference-Driven Natural Language Processing*. PhD thesis, Universität des Saarlandes, Saarbrücken, forthcoming.

[Erjavec, 1994] Tomaž Erjavec. Finite State Phonology in ALE. In *ILD Deliverable*. Language Technology Group, University of Edinburgh, 1994.

[Evans *et al.*, 1995] Roger Evans, Gerald Gazdar, and David Weir. Encoding lexicalized tree adjoining grammars with a nonmonotonic inheritance hierarchy. In 33^{rd} *Meeting of the Association for Computational Linguistics (ACL'95)*, 1995.

[G. Erbach and Thiersch, 1995] M. van der Kraan W. Skut G. Erbach, S. Manandhar and C. Thiersch. Extending unification formalisms. In *Proceedings of the 2nd Language Engineering Convention*, London, 1995.

[Gabbay, 1991] D. Gabbay. *Labelled Deductive Systems*. Oxford University Press, 1991. (to appear).

[Gazdar *et al.*, 1985] G. Gazdar, E. Klein, G.K. Pullum, and I. Sag. *Generalized Phrase Structure Grammar*. Basil Blackwell, Oxford, 1985.

[Gerdemann and Götz, 1994] Dale Gerdemann and Thilo Götz. Troll: Type resolution system: User's guide. Research report, Universität Tübingen, 1994.

[Groenendijk and International, 1993a] M. Groenendijk and P-E International. *ALEP-0 Graphical User Interface*. CEC, 1993.

[Groenendijk and International, 1993b] M. Groenendijk and P-E International. *XMFED User Guide*. CEC, 1993.

[H. U. Krieger, 1995] U. Schäfer H. U. Krieger. Efficient parameterizable type expansion for typed feature formalisms. In *Proceedings of the 4th MOL*, 1995.

[Halvorsen and Kaplan, 1988] Per-Kristian Halvorsen and Ronald M. Kaplan. Projections and semantic description in Lexical-Functional Grammar. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS-88)*, pages 1116–1122, Tokyo, Japan, November 1988. Reprinted in Dalrymple, Kaplan, Maxwell, and Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 279–292. Stanford: Center for the Study of Language and Information. 1995.

[Haridi and Janson, 1990] S. Haridi and S. Janson. Kernel andorra prolog and its computation model. In *International Conference in Logic Programming (ICLP)*, pages 31–46, 1990.

[Hellwig, 1986] P. Hellwig. Dependency unification grammar. In *COLING 86*, pages 195–198, 1986.

[Hellwig, 1988] P. Hellwig. Chart parsing according to the slot and filler principle. In *COLING 88*, pages 242–244, 1988.

[Hendriks, 1993] Herman Hendriks. *Studied Flexibility. Categories and Types in Syntax and Semantics*. PhD thesis, University of Amsterdam, 1993. ILLC Dissertation Series 1993–5.

[Hendriks, 1994] Herman Hendriks. Information packaging in a categorial perspective. In E. Engdahl, editor, *Integrating Information Structure into Constraint-based and Categorial Approaches*, ESPRIT Basic Research Project 6852, Dynamic Interpretation of Natural Language, DYANA-2 Deliverable R1.3.B., pages 89–116. ILLC, Amsterdam, 1994.

[Hendriks, 1995a] Herman Hendriks. Intonation, derivation, information. Draft, Utrecht University, 1995.

[Hendriks, 1995b] Petra Hendriks. *Comparatives and Categorial Grammar*. PhD thesis, University of Groningen, 1995.

[Hinrichs and Nakazawa, 1989] Erhard Hinrichs and Tsuneko Nakazawa. Flipped out: Aux in german. In *Papers from the 25th Regional Meeting of the Chicago Linguistic Society*, volume 25, pages 193–202, Chicago, Illinois, 1989. CLS, CLS.

[Hinrichs and Nakazawa, 1990] Erhard Hinrichs and Tsuneko Nakazawa. Subcategorization and vp structure in german. In Shaun Hughes and Joe Salmons, editors, *Proceedings of the Third Symposium on Germanic Linguistics*, Amsterdam, 1990. Benjamins.

[Hinrichs and Nakazawa, 1994] Erhard Hinrichs and Tsuneko Nakazawa. Linearizing AUXs in German verbal complexes. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar*, number 46 in Lecture Notes, pages 11–37. CSLI Publications, Stanford University, 1994.

[Hoffmann, 1995a] Beryl Hoffmann. *The computational analysis of the syntax and interpretation of "Free" word order in Turkish*. PhD thesis, Institute for Research in Cognitive Science. University of Pennsylvania, 1995. Technical Report 95-17.

[Hoffmann, 1995b] Beryl Hoffmann. Integrating free word order syntax and information structure. In *Proceedings of the European Association for Computational Linguistics (EACL)*, Dublin, 1995.

[Höhfeld and Smolka, 1988] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IBM Deutschland, October 1988.

[Holzbaur, 1992] C. Holzbaur. DMCAI CLP reference manual. Technical Report TR-92-24, Österreichisches Forschungsinstitut für Artificial Intelligence, 1992.

[Hudson, 1984] Richard Hudson. *Word Grammar.* Basil Blackwell Ltd., 1984.

[IAI *et al.*, 1991] IAI, CAP Gemini, and SNI. Et6/2 software environment study vol2 outline design study (final report). Technical report, CEC, 1991.

[Jacobson,] Pauline Jacobson. Towards a variable-free semantics. (in preparation).

[Jacobson, 1994] Pauline Jacobson. i-within-i effects in a variable-free semantics and a categorial syntax. In *Proceedings of the 9th Amsterdam Colloquium.* University of Amsterdam, ILLC., 1994.

[Jacobson, to appear] Pauline Jacobson. The syntax/semantics interface in categorial grammar. In S. Lappin, editor, *Handbook of Semantics.* Blackwell, (to appear).

[Jaffar and Lassez, 1987] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings 14th POPL*, 1987.

[Johnson and Dörre, 1995] M. Johnson and J. Dörre. Memoization of coroutined constraints. In *Proceedings of the 32rd Annual Meeting of the Association for Computational Linguistics*, 1995.

[Johnson, 1988] Mark Johnson. *Attribute-Value Logic and the Theory of Grammar*, volume 16 of *CSLI Lecture Notes.* CSLI, 1988.

[Johnson, 1991] Mark Johnson. Features and formulae. *Computational Linguistics*, 17(2):131–151, 1991.

[Johnson, 1994] Mark Johnson. Computing with features as formulae. *Computational Linguistics*, 20(1):1–25, 1994.

[Joshi *et al.*, 1975] Aravind K. Joshi, L. S. Levy, and M. Takahashi. Tree Adjunct Grammars. *J. Comput. Syst. Sci.*, 10(1), 1975.

[Joshi, 1987a] Aravind K. Joshi. An introduction to Tree Adjoining Grammars. In A. Manaster-Ramer, editor, *Mathematics of Language.* John Benjamins, Amsterdam, 1987.

[Joshi, 1987b] Aravind K. Joshi. An introduction to Tree Adjoining Grammars. In A. Manaster-Ramer, editor, *Mathematics of Language*, pages 87–115. jb, Amsterdam, 1987.

[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 and Maxwell III, 1988] R. M. Kaplan and J. T. Maxwell III. An algorithm for functional uncertainty. In *Proceedings of the 12th International Conference on Computational Linguistics*, pages 297–302, Budapest, Hungary, 1988.

[Kaplan and Maxwell, 1988] Ronald M. Kaplan and John T. Maxwell. Constituent coordination in Lexical-Functional Grammar. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING '88)*, pages 303–305, Budapest, 1988. Reprinted in Dalrymple, Kaplan, Maxwell, and Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 199–210. Stanford: Center for the Study of Language and Information. 1995.

[Kaplan and Maxwell, 1993] Ronald M. Kaplan and John T. Maxwell. LFG grammar writer's workbench. Technical report, Xerox PARC, 1993.

[Kaplan and Zaenen, 1988] Ronald M. Kaplan and Annie Zaenen. Long-distance dependencies, constituent structure, and functional uncertainty. In M. Baltin and A. Kroch, editors, *Alternative Conceptions of Phrase Structure*. University of Chicago Press, Chicago, 1988.

[Kaplan and Zaenen, 1989] Ronald M. Kaplan and Annie Zaenen. Functional precedence and constituent structure. In Chu-Ren Huang and Keh-Jiann Chen, editors, *Proceedings of the Republic of China Computational Linguistics Conference (ROCLING II)*, pages 19–40. Academia Sinica, Taipei, 1989.

[Kaplan, 1987] Ronald M. Kaplan. Three seductions of computational psycholinguistics. In Peter Whitelock, Mary McGee Wood, Harold L. Somers, Rod Johnson, and Paul Bennett, editors, *Linguistic Theory and Computer Applications*, pages 149–181. Academic Press, London, 1987. Also: CCL/UMIST Report No. 86.2: Alvey/ICL Workshop on Linguistic Theory and Computer Applications: Transcripts of Presentations and Discussions. Center for Computational Linguistics, University of Manchester, Institute of Science and Technology, Manchester. Reprinted in Dalrymple, Kaplan,Maxwell, and Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 337–367. Stanford: Center for the Study of Language and Information. 1995.

[Karlsson *et al.*, 1995] Fred Karlsson, Atro Voutilainen, Juha Heikkil"a, and Arto Anttila. *Constraint Grammar. A Language-Independent System for Parsing Unrestricted Text.* Mouton de Gruyter, Berlin, New York, 1995.

[Karttunen, 1994] L. Karttunen. Constructing lexical transducers. In *Proceedings of COLING-94*, pages 406–411, 1994.

[Kasper and Rounds, 1986] R. Kasper and W. Rounds. A logical semantics for feature structures. In *24th Annual Meeting of the Association for Computational Linguistics*, pages 257–265, 1986.

[Kasper and Rounds, 1990] Ropert T. Kasper and William C. Rounds. The logic of unification grammar. *Lingusitic and Philosophy*, 13:35–58, 1990.

[Kasper *et al.*, 1995] Robert Kasper, Bernd Kiefer, Klaus Netter, and K. Vijay-Shanker. Compilation of hpsg to tag. In *33rd Annual Meeting of the Association for Computational Linguistics, MIT, Cambridge, MA*, 1995.

[Kathol, 1994] Andreas Kathol. Passives without lexical rules. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German Grammar in HPSG*. CSLI Publications, Stanford, CA, 1994.

[Kay, 1980] M. Kay. Algorithm schemata and data structures in syntactic processing. Technical report, Xerox PARC, 1980.

[Kilger and Finkler, 1995] A. Kilger and W. Finkler. Incremental generation for real–time applications. DFKI Report RR-95-11, German Research Center for Artificial Intelligence – DFKI GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1995.

[Kiss, 1991a] Tibor Kiss. The grammars of lilog. In Otthein Herzog and Claus-Rolf Rollinger, editors, *Text Understanding in LILOG*, number 546 in Lecture Notes in Artificial Intelligence, pages 183–199. Springer-Verlag, Berlin, 1991.

[Kiss, 1991b] Tibor Kiss. On the treatment of optionality in HPSG. IWBS Report 194, IWBS, Stuttgart, 1991.

[König, 1995] Esther König. Lexgram - a practical categorial grammar formalism. In *Proceedings of the Workshop on Computational Logic for Natural Language Processing. A Joint COMPULOGNET/ELSNET/EAGLES Workshop*, Edinburgh, Scotland, April 1995.

[Koskenniemi *et al.*, 1992] Kimmo Koskenniemi, Atro Voutilainen, and Pasi Tapanainen. Compiling and using finite-state syntactic rules. In *Proceedings of COLING*, pages 156 – 162, 1992.

[Koskenniemi, 1990] Kimmo Koskenniemi. Finite-state parsing and disambiguation. In *Proceedings of COLING*, pages 229 – 232, 1990.

[Kraak, 1995] Esther Kraak. French clitics: A categorial perspective. Master's thesis, Utrecht University, 1995.

[Krieger and Nerbonne, 1993] Hans-Ulrich Krieger and John Nerbonne. Feature-based inheritance networks for computational lexicons. In Ted Briscoe, Ann Copestake, and Valeria de Paiva, editors, *Default Inheritance Within Unification-Based Approaches to the Lexicon*, pages 90–136. Cambridge University Press, Cambridge, 1993. Also published as Report RR-91-31, DFKI, Saarbrücken.

[Krieger and Schäfer, 1994] H.-U. Krieger and U. Schäfer. $\mathcal{TDL}$—a type description language for constraint-based grammars. In *Proceedings of the 15th International Conference on Computational Linguistics*, 1994.

[Krieger, 1995] H. U. Krieger. Typed feature structures, definite equivalences, greatest model semantics and nonmonotonicity. In *Proceedings of the 4th MOL*, 1995.

[Kroch and Joshi, 1985] A. Kroch and A. K. Joshi. Linguistic relevance of Tree Adjoining Grammars. Technical Report MS-CIS-85-18, Department of Computer and Information Science, University of Pennsylvania, 1985.

[Laubsch, 1994] Joachim Laubsch. *ZEBU A Tool for specifying Reversible LALR(1) Parsers*, 1994.

[Lloré and Morrill, 1995] Lloré and Morrill. Difference lists and difference bags for logic programming of categorial deduction. In *Proceedings of SEPLN XI, Deusto*, 1995. Also available as Report de Recerca LSI-95-30-R , Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.

[Manandhar, 1993a] S. Manandhar. *Relational Extensions to Feature Logic: Applications to Constraint Based Grammars.* PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Manandhar, 1993b] Suresh Manandhar. CUF in Context. In Jochen Dörre, editor, *Computational Aspects of Constraint-Based Linguistic Description I.* DYANA-2, Deliverable R1.2.A, August 1993.

[Manandhar, 1993c] Suresh K. 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 *Proc. of the 32th ACL*, 1994.

[Manandhar, 1995] S. Manandhar. Deterministic consistency checking of LP constraints. In *Proceedings of the 7th European Conference of the Association for Computational Linguistics*, 1995.

[Matiasek, 1994] J. Matiasek. *Principle-based Processing of Natural Language using CLP Techniques.* PhD thesis, Technische Universität, Vienna, October 1994.

[Matsumoto *et al.*, 1983] Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom-up parser embedded in prolog. In *New Generation Computing*, pages 1:145–158. 1983.

[Maxwell III and Kaplan, 1991] J.T. Maxwell III and R.M. Kaplan. A method for disjunctive constraint satisfaction. In M. Tomita, editor, *Current issues in parsing technology*, pages 173 – 190. Kluwer, Boston, 1991.

[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.

[Meurers and Minnen, 1995] W. Detmar Meurers and Guido Minnen. A Computational Treatment of Lexical Rules as Covariation in Lexical Entries. In *Proceedings of the Fifth International Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Portugal, 1995.

[Meurers, 1995] Walt Detmar Meurers. Towards a Semantics for Lexical Rules as used in HPSG. In *Proceedings of the Conference on Formal Grammar*, Barcelona, Spain, august 1995.

[Meylemans and Simpkins, 1993] P. Meylemans and N. Simpkins. Towards a portable platform for language research and engineering. In *Thirteenth International Conference on Artificial Intelligence, Expert Systems and Natural Language*, volume 3, pages 161–170, 1993.

[Meylemans, 1994] Paul Meylemans. ALEP - Arriving at the next platform. *ELSNews*, 3(2):4–5, 1994.

[Miller and Sag, 1993] Philip H. Miller and Ivan A. Sag. French clitic climbing without clitics or climbing, 1993. Unpublished ms., University of Lille and Stanford University.

[Milward, 1994a] David Milward. Dynamic dependency grammar. *Linguistics and Philosophy*, 17:561–605, 1994.

[Milward, 1994b] David Milward. Non-constituent coordination: Theory and practice. In *Proceedings of COLING 94*, pages 935–941, Kyoto, Japan, 1994.

[Milward, 1995] David Milward. Incremental interpretation of categorial grammar. In *Proceedings of the 7th Conference of the European Chapter of the ACL*, pages 119–126, Dublin, 1995.

[Monachesi, 1993a] Paola Monachesi. Object clitics and clitic climbing in italian hpsg grammar. In *Proc. of EACL 93*. Utrecht, 1993.

[Monachesi, 1993b] Paola Monachesi. On *si* constructions in italian hpsg grammar. In *Proceedings of ESCOL 10*. Cornell University, 1993.

[Monachesi, 1993c] Paola Monachesi. Restructuring verbs in italian hpsg grammar. In *Proceedings of CLS 29*, 1993.

[Monachesi, to appear] Paola Monachesi. Italian Clitics and Restructuring Verbs: a Lexical Analysis. In Sergio Balari and Luca Dini, editors, *Romance in HPSG*. CSLI Publications, Stanford University, to appear.

[Montague, 1974] Richard Montague. *Formal Philosophy. Selected Papers of Richard Montague*. Yale University Press. New Haven, 1974.

[Moortgat and Kurtonina, to appear] Michael Moortgat and Natasha Kurtonina. Structural control. In P. Blackburn and M. de Rijke, editors, *Structures and Syntax*. Kluwer, (to appear).

[Moortgat and Oehrle, 1993a] Michael Moortgat and Richard Oehrle. Logical parameters and linguistic variation. Lecture notes ESSLLI, Lisboa, 1993.

[Moortgat and Oehrle, 1993b] Michael Moortgat and Richard T. Oehrle. Adjacency, dependency and order. In *Proceedings of the 9th Amsterdam Colloquium*, 1993.

[Moortgat and Oehrle, 1994] Michael Moortgat and Richard T. Oehrle. Adjacency, dependency and order. In *Proceedings 9th A'dam Colloquium*, 1994.

[Moortgat, 1988] Michael Moortgat. *Categorial Investigations – Logical and Linguistic Aspects of the Lambek Calculus.* GRASS. Foris Publications, 1988.

[Moortgat, to appeara] Michael Moortgat. Categorial grammar logics. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*. Elsevier, (to appear).

[Moortgat, to appearb] Michael Moortgat. In situ binding: a modal approach. In *Proceedings of the 10th Amsterdam Colloquium*, (to appear).

[Moortgat, to appearc] Michael Moortgat. Multimodal linguistic inference. *JoLLI*, (to appear). preliminary version in IGPL Bulletin 3 (2-3), 1995.

[Morrill and Aris, 1993] Glyn Morrill and Teresa Solias Aris. Tuples, discontinuity and gapping in categorial grammar. In *Proceedings of the Meeting of the European Chapter of the Association for Computational Linguistics*, Utrecht, 1993.

[Morrill, 1994a] Glyn Morrill. *Type Logical Grammar: Categorial Logic of Signs, research monograph.* Kluwer Academic Publishers, Dordrecht, 1994.

[Morrill, 1994b] Glyn V. Morrill. *Type Logical Grammar: Categorial Logic of Signs.* Kluwer, Dordrecht, 1994.

[Morrill, 1995a] Glyn Morrill. Clausal proofs and discontinuity. *Bulletin of the IGPL*, 3(2, 3):403–427, 1995.

[Morrill, 1995b] Glyn Morrill. Discontinuity in categorial grammar. *Linguistics and Philosophy*, 18:175–219, 1995.

[Morrill, 1995c] Glyn Morrill. Higher order linear logic programming of categorial deduction. In *Proceedings of the Meeting of the European Chapter of the Association for Computational Linguistics*, Dublin, 1995. Refinement and condensation of Report de Recerca LSI-94-42-R , Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya.

[Moshier and Pollard, 1993] M. Andrew Moshier and Carl J. Pollard. The domain of set-valued feature structures. CLAUS Report 35, Universität des Saarlandes - Computerlinguistik, 1993.

[Nerbonne *et al.*, 1993] John Nerbonne, Klaus Netter, Kader Diagne, Ludwig Dickmann, and Judith Klein. A Diagnostic Tool for German Syntax. *Machine Translation*, 1993.

[Nerbonne *et al.*, 1994] John Nerbonne, Klaus Netter, and Carl Pollard, editors. *German in Head-Driven Phrase Structure Grammar.* Number 46 in Lecture Note Series. CSLI, Stanford, 1994.

[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*, number 46 in Lecture Notes, pages 109–150. CSLI Publications, Stanford University, 1994.

[Netter, 1992] Klaus Netter. On Non-Head Non-Movement. An HPSG Treatment of Finite Verb Position in German. In G. Görz, editor, *Proceedings of KONVENS 92*, Berlin and Heidelberg and New York, 1992. Springer.

[Netter, 1993] Klaus Netter. Architecture and Coverage of the Disco Grammar. In Busemann and Harbusch [1993], pages 1–10.

[Netter, 1996] Klaus Netter. Functions and categories. an HPSG treatment of german (working title). Doctoral Dissertation, Computational Linguistics Department, Universität des Saarlandes (forthcoming), 1996.

[Neumann, 1993] Günter Neumann. Grammatikformalismen in der Generierung und ihre Verarbeitung. FBO-Verlag, Baden-Baden, 2 1993. KI Themenheft: Generierung.

[Neumann, 1994a] G. 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, March 1994.

[Neumann, 1994b] G. Neumann. *A Uniform Computational Model for Natural Language Parsing and Generation.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1994.

[Oehrle and Moortgat,] Richard T. Oehrle and Michael Moortgat. Grammatical logics. NSF Project Description.

[Oehrle, 1995a] Richard T. Oehrle. Labeled deduction, binding, quantification. Paper delivered at the Proof Theory Symposium. ESSLLI. Barcelona., 1995.

[Oehrle, 1995b] Richard T. Oehrle. Lecture notes on prosody and grammatical architecture. ESSLLI. Barcelona., 1995.

[Oehrle, 1995c] Richard T. Oehrle. Some 3-dimensional systems of labelled deduction. *Bulletin of the IGPL. 3.2,3*, pages 429–448, 1995.

[Oehrle, 1995d] Richard T. Oehrle. Term-labeled categorial type systems. *Linguistics & Philosophy*, 17:633–678, 1995.

[Oepen *et al.*, 1996] Stephan Oepen, Klaus Netter, and Judith Klein. TSNLP — Test Suites for Natural Language Processing. In John Nerbonne, editor, *Linguistic Databases*, CSLI Lecture Notes. Center for the Study of Language and Information, 1996. forthcoming.

[Ousterhout, 1994] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[Pereira and Warren, 1983] F.C.N. Pereira and D.H.D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, 1983.

[Pereira, 1981] F.C.N Pereira. Extraposition grammars. *Computational Linguistics*, 7, 1981.

[Pollard and Moshier, 1990] Carl J. Pollard and M. Drew Moshier. Unifying partial descriptions of sets. In P. Hanson, editor, *Information, Language, and Cognition. Vol. 1 of Vancouver Studies in Cognitive Science*, pages 285–322. University of British Columbia Press, 1990.

[Pollard and Sag, 1987] Carl Pollard and Ivan Andrew Sag. *Information-Based Syntax and Semantics: Volume 1 Fundamentals*, volume 13 of *Lecture Notes*. Center for the Study of Language and Information, Stanford, CA, 1987.

[Pollard and Sag, 1994a] C. Pollard and I. Sag. *Head Driven Phrase Structure Grammar.* University of Chicago Press and CSLI Publications, Stanford: Center for the Study of Language and Information, 1994.

[Pollard and Sag, 1994b] Carl Pollard and Ivan Andrew Sag. *Head-driven Phrase Structure Grammar.* Chicago: University of Chicago Press and Stanford: CSLI Publications, 1994.

[Pollard, 1989] Carl Pollard. Sorts in unification-based grammar and what they mean. Unpublished manuscript, 1989.

[Pollard, 1994] Carl J. Pollard. Toward a Unified Account of Passive in German. In John Nerbonne, Klaus Netter, and Carl Pollard, editors, *German in Head-Driven Phrase Structure Grammar*, number 46 in Lecture Notes, pages 273–296. CSLI Publications, Stanford University, 1994.

[Prevost and Steedman, to appear] Scott Prevost and Mark Steedman. Specifying intonation from context for speech synthesis. *Speech Communication*, (to appear). Accepted April 26, 1994, Revised May 16, 1994.

[Quesada and Amores, 1995] J. F. Quesada and G. Amores. A computational model for the efficient retrieval of very large structure-based knowledge bases. In *Proceedings of KRUSE Symposium*, University of California at Santa Cruz, 1995.

[Quesada and Amores, 1996] J. F. Quesada and J. G. Amores. C for natural language processing. forthcoming, 1996.

[Rea, 1994] Domain union and word order variation in german. In J. Nerbonne, K. Netter, and C. Pollard, editors, *German in HPSG*. CSLI, 1994.

[Reape, 1991] Mike Reape. An introduction to the semantics of unification-based grammar formalisms. Deliverable R3.2.A, DYANA, Centre for Cognitive Science, University of Edinburgh, January 1991.

[Reape, 1993] Mike Reape. Getting Things in Order. In Wietske Sijtsma and Arthur van Horck, editors, *Discontinuous Constituency*. Berlin: Mouton de Gruyte, 1993.

[Rentier, 1994] Gerrit M. Rentier. A lexicalist approach to dutch cross serial dependencies. In *Papers from the 30th Regional Meeting of the Chicago Linguistic Society*, volume 30, Chicago, Illinois, 1994. CLS, CLS.

[Riehemann, 1993] S. Riehemann. Word formation in lexical type hierarchies. Master's thesis, University of Tubingen, 1993.

[Riehemann, 1994] Susanne Riehemann. Morphology and the hierarchical lexicon. Stanford University, 1994.

[Rogers and Vijay-Shanker, 1994] James Rogers and K. Vijay-Shanker. Obtaining trees from their descriptions: An application to Tree-Adjoining Grammars. *Computational Intelligence*, 10:401–421, 1994.

[S. Starosta, 1986] H.Nomura S. Starosta. Lexicase parsing: A lexicon-driven approach to syntactic analysis. In *COLING 86*, pages 195–198, 1986.

[Samuelsson, 1994a] C. Samuelsson. *Fast Natural-Language Parsing Using Explanation-Based Learning*. PhD thesis, The Royal Institute of Technology and Stockholm University, February 1994.

[Samuelsson, 1994b] C. Samuelsson. Notes on LR parser design. In *Proceedings of the 15th International Conference on Computational Linguistics*, pages 386 – 390, 1994.

[Samuelsson, 1995] C. Samuelsson. An efficient algorithm for surface generation. In *IJCAI 95*, 1995.

[Sanfilippo, 1990] A. Sanfilippo. *Grammatical Relations, Thematic Roles and Verb Semantics*. PhD thesis, University of Edimburgh, 1990.

[Sanfilippo, 1993] Antonio Sanfilippo. Grammatical relations in unification categorial grammar. *Lingua e Stile, XXVIII-2*, pages 171–199, 1993.

[Sanfilippo:, to appear] Antonio Sanfilippo:. Thematically Bound Adjuncts. In Sergio Balari and Luca Dini, editors, *Romance in HPSG*. CSLI Publications, Stanford University, to appear.

[Saraswat, 1993] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Schabes, 1990] Yves Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars*. PhD thesis, University of Pennsylvania, Philadelphia, PA, August 1990. Available as technical report (MS-CIS-90-48, LINC LAB179) from the Department of Computer Science.

[Shieber *et al.*, 1983] S.M. Shieber, H. Uszkoreit, F.C.N. Pereira, J.J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In B.J. Grosz and M.E. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge*, pages 39–79. SRI International, Menlo Park, CA, 1983.

[Shieber *et al.*, 1990] S.M. Shieber, G. van Noord, F.C.N. Pereira, and R.C. Moore. Semantic-head-driven generation. *Computational Linguistics*, 16(1):30–42, 1990.

[Shieber *et al.*, 1991] S.M. Shieber, G. van Noord, R.C. Moore, and F.C.N. Pereira. Semantic-head-driven generation. *Computational Linguistics*, 16:30–42, 1991.

[Shieber, 1985] S.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, 1988] S.M. Shieber. A uniform architecture for parsing and generation. In *COLING-88*, pages 614–619, 1988.

[Simpkins *et al.*, 1993] N.K. Simpkins, G. Cruickshank, and P-E International. *ALEP-0 Virtual Machine Extensions.* CEC, 1993.

[Smolka and Treinen, 1994] Gert Smolka and Ralf Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3):229–258, April 1994.

[Smolka *et al.*, 1995] G. Smolka, M. Henz, and J. 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, 1992] G. Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51 – 87, 1992.

[Srinivas *et al.*, 1994] B. Srinivas, Dania Egedi, Christy Doran, and Tilman Becker. Lexicalization and grammar development. In *Proceedings of KONVENS 94*, Vienna, Austria, 1994.

[Steedman, 1984] Mark Steedman. A categorial theory of intersecting dependencies in dutch infinitival complements. In De Geest and Putseys, editors, *Proceedings of the International Conference on Complementation*, pages 215–226, Dordrecht, 1984. Foris.

[Strzalkowski, 1991] T. Strzalkowski. A general computational method for grammar inversion. In T. Strzalkowski, editor, *ACL SIG workshop "Reversible Grammar in Natural Language Processing"*, pages 91 – 99, 1991.

[Tesnière, 1959] L. Tesnière. *Éléments de Syntaxe Structurale.* Klincksieck, Paris, 1959.

[Treinen, 1993] Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science, vol. 711, pages 734–743, Gdańsk, Poland, 30 August–3 September 1993. Springer-Verlag.

[Trost and Matiasek, 1994] H. Trost and J. Matiasek. Morphology with a null-interface. In *Proceedings of the 15th International Conference on Computational Linguistics*, 1994.

[Trost, 1993] H. Trost, editor. *Feature Formalisms and Linguistic Ambiguity.* Ellis Horwood, Chichester, 1993.

[Uszkoreit *et al.*, 1994] Hans Uszkoreit, Rolf Backofen, Stephan Busemann, Abdel Kader Diagne, Elizabeth A. Hinkelman, Walter Kasper, Bernd Kiefer, Hans-Ulrich Krieger, Klaus Netter, Günter Neumann, Stephan Oepen, and Stephen P. Spackman. DISCO–An HPSG-based NLP System and its Application for Appointment Scheduling. In *Proceedings of the 15th International Conference on Computational Linguistics, COLING-94*, Kyoto, Japan, 1994.

[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 Benthem, 1986] Johan van Benthem. Categorial grammar. In *Essays in Logical Semantics*, chapter 7, pages 123–150. Reidel, Dordrecht, 1986.

[van Noord and Bouma, 1994] Gertjan van Noord and Gosse Bouma. The scope of adjuncts and the processing of lexical rules. In *Proceedings of Coling 94*, pages 250–256, Kyoto, 1994.

[van Noord, 1990] G. van Noord. An overview of head-driven bottom-up generation. In R. Dale, C. Mellish, and M. Zock, editors, *Current Research in Natural Language Generation*, pages 141–165. Academic Press, London, 1990.

[van Noord, 1991] Gertjan van Noord. Head corner parsing for discontinuous constituency. In *29th Annual Meeting of the Association for Computational Linguistics*, 1991.

[van Noord, 1994] G. van Noord. Head corner parsing for tag. *Computational Intelligence*, 10, 1994.

[Versmissen, 1993a] Koen Versmissen. Categorial grammar, modalities and algebraic semantics. In *Proceedings EACL93*, Utrecht, 1993.

[Versmissen, 1993b] Koen Versmissen. Word order domains in categorial grammar. In *Linguistic Applications of Multimodal and Polymorphic Categorial Grammar*, Dyana-2 Deliverable R1.3.C. Amsterdam, 1993.

[Vijay-Shanker and Joshi, 1988] K. Vijay-Shanker and A.K. Joshi. Feature structure based Tree Adjoining Grammars. In *Proceedings of the* 12^{th} *International Conference on Computational Linguistics (COLING'88)*, Budapest, August 1988.

[Vijay-Shanker and Schabes, 1992] K. Vijay-Shanker and Yves Schabes. Structure sharing in Lexicalized Tree-Adjoining Grammars. In 20^{th} *Meeting of the Association for Computational Linguistics (ACL'92)*, 1992.

[Vijay-Shanker, 1992] K. Vijay-Shanker. Using descriptions of trees in a tree adjoining grammar. *Computational Linguistics*, pages 519 – 530, 1992.

[Voutilainen and Tapanainen, 1993] Atro Voutilainen and Pasi Tapanainen. Ambiguity resolution in a reductionistic parser. In *Proceedings of EACL*, pages 394 – 403, 1993.

[Voutilainen, 1993] Atro Voutilainen. Nptool, a detector of english noun phrases. In *Proceedings of the Workshop on Very Large Corpora*, pages 42 – 51, Ohio State University, Ohio, 1993. Association for Computational Linguistics.

[Voutilainen, forthcomming] Atro Voutilainen. The design of a (finite-state) parsing grammar. In E. Roche and I. Schabes, editors, *Finite State Devices for Natural Language Processing*. MIT Press, forthcomming.

[Warren, 1992] David S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93 – 111, 1992.

[Webelhuth, 1995] Gert Webelhuth, editor. *Government and Binding Theory, and the Minimalist Program*. Blackwell, Oxford, 1995.

[Weyers, 1996] C. Weyers. Präferenzgesteuerte Unifikation von Merkmalsstrukturen. Master's thesis, Univ. of Saarbrücken, 1996.

[Wilson and Borning, 1993] M. Wilson and A. Borning. Hierarchical constraint logic programming. Technical Report 93-01-02a, Department of Computer Science and Engineering, University of Washington, May 1993.

[XTAG, 1995] project team XTAG. A lexicalized Tree Adjoining Grammar for English. Technical Report 95-03, IRCS, University of Pennsylvania, Philadelphia, USA, 1995.

[Zajac, 1992a] Rémi Zajac. Inheritance and constraint-based grammar formalisms. *Computational Linguistics*, 18(2):159–182, 1992.

[Zajac, 1992b] Rémi Zajac. Inheritance and Constraint-Based Grammar Formalisms. *Computational Linguistics*, 18(2):159–182, 1992.