Natural Language Generation with Head-Driven Phrase Structure Grammar

A thesis submitted to the University of Manchester Institute of Science and Technology for the degree of Doctor of Philosophy 1998

Graham Wilcock

Centre for Computational Linguistics Department of Language Engineering
Abstract

The thesis discusses natural language generation with Head-Driven Phrase Structure Grammar (HPSG). A series of six implementations are described. The first three implementations are all based on the HPSG theory of the 1994 textbook by Pollard and Sag. The next three relatively experimental implementations are based on significant revisions of HPSG theory after 1994. Most of the systems use a head-driven generation algorithm, as it is argued that head-driven generation is the natural approach with a head-driven grammar such as HPSG.

The first implementation, which uses the ALE typed feature system of Carpenter and Penn, is a fairly full implementation of the 1994 HPSG textbook. The textbook version of HPSG is taken as an already known starting point without detailed explanation, and the description is concerned primarily with how the theory is implemented. This first implementation follows the 1994 HPSG theory closely, but it can only perform parsing, not generation.

By contrast, the second implementation can do generation as well as parsing. Three head-driven generation algorithms are compared: van Noord’s head-driven bottom-up generator, Shieber et al’s semantic head-driven generator and Haruno et al’s chart-based semantic head-driven generator. Matsumoto’s SAX parser and SGX chart-based semantic head-driven generator are introduced.

An English Engine is described which uses SAX and SGX for parsing and generation with a DCG grammar which has many HPSG features and mechanisms. However the grammar does not include inheritance-based typed feature structures and therefore only approximates to HPSG theory.

The third implementation combines the strengths of the first two into a single framework by using Erbach’s ProFIT typed feature system. This completely revised version of English Engine performs efficient parsing and generation using SAX and SGX, but combines them with an HPSG grammar based on an inheritance-based typed feature structure representation using ProFIT. This system includes an implementation of delayed lexical choice in generation with HPSG, which exploits the monotonicity of subsumption in the sort hierarchy. This approach to delayed lexical choice, based on deliberate underspecification of features, extends and generalizes previous approaches.

A number of revisions of HPSG theory have been proposed since 1994. Some of the revisions, and experimental implementations of their main ideas, are described in the second part of the thesis. The fourth implementation is based on the proposals of Sag (1997) to eliminate all empty categories from HPSG theory. This has specific importance for the implementations in this thesis, as SAX and SGX do not permit empty categories. Sag’s revised analysis of English relative clauses is implemented in ProFIT using a hierarchy of construction types. Constraints on phrase types are implemented by ProFIT templates. This implementation handles all forms of English relative clauses without using empty categories, but it can only do parsing, not generation.

The thesis includes a proposal for another theoretical revision in HPSG: the lexicalization of context. The proposed change strengthens the role of semantic heads in the overall organization of HPSG grammar. It is needed to solve fundamental problems with handling contextual information in head-driven generation. Theoretical, computational and linguistic justifications for the proposal are presented. These include a sketch of an analysis of register variation, proposing a lexicalist account of case assignment and register restrictions within Sag’s revised analysis of English relative clauses. The fifth implementation is a basic implementation of these theoretical revisions.

Another major theoretical revision is the use of Minimal Recursion Semantics (MRS) in HPSG. In order to perform generation using MRS, the sixth implementation switches to non-head-driven generation. Phillips’ bag generation algorithm for categorial grammar and indexed logical form is adapted for use with HPSG and MRS. The algorithm is then modified to perform incremental generation with HPSG, using simple rules for utterance continuation and repairs. Difficulties in using HPSG for incremental generation are discussed, and a chart-based solution is proposed.
Declaration

No portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acknowledgements

This work was done as a visiting researcher of Sharp Corporation at UMIST. I am particularly indebted to Hitoshi Suzuki of Sharp for making it possible. Thanks also to Junzo Ogawa, Osamu Nishida, Takehiko Yoshimi and others at Sharp for their help.

The start of the work was supervised by Prof Jun-ichi Tsujii. Its completion was supervised by Bill Black. I am most grateful to them both for all their kind help and wise advice. Thanks also to Paul Bennett, John Phillips, Arturo Trujillo and many other past and present members of the Centre for Computational Linguistics at UMIST for valuable discussions.

I am grateful to Prof Yuji Matsumoto of NAIST for allowing me to spend time as a visiting researcher at his laboratory and for his collaboration on SAX and SGX. Thanks also to Yasuharu Den, Takehiro Utsuro and Takashi Miyata for their help at NAIST.

Many thanks to Stefan Busemann, Gregor Erbach and others at DFKI Saarbrücken, and to Antonio Sanfilippo, Pete Whitelock and others at SLE Oxford for discussions and comments.

Finally, I thank Kristiina Jokinen for valuable comments and invaluable encouragement, and also for pilgrimages in Shikoku and volcanoes in Kyushu, for quiet ripples on calm Finnish lakes and fierce snowstorms in Kansai, for a crazy reindeer, and for those magical times when, for example, the gentle Nara deer tiptoe in silhouette across the face of the moon.
Contents

1 Introduction .......................................................... 7
   1.1 Generation with HPSG ........................................ 7
       1.1.1 A Trivial Matter? ...................................... 7
       1.1.2 Generation from What? ................................ 8
   1.2 Aims of the Thesis ........................................... 9
   1.3 Structure of the Thesis ...................................... 10
   1.4 Original Contributions of the Thesis ...................... 11
   1.5 Published Material ......................................... 12

I Towards Generation with HPSG (1994 Theory) .................. 13

2 Parsing with HPSG and ALE: Implementing the Textbook ...... 14
   2.1 HPSG in Pollard and Sag 1994 .............................. 14
   2.2 The ALE Typed Feature System ............................. 14
   2.3 Implementing the Textbook ................................ 15
       2.3.1 A System of Signs ................................... 16
       2.3.2 Agreement ............................................ 19
       2.3.3 Complement Structures ............................... 24
       2.3.4 Unbounded Dependencies .............................. 26
       2.3.5 Relative Clauses .................................. 30
       2.3.6 Binding Theory .................................... 32

3 Generation with DCG and SGX: An English Engine .......... 36
   3.1 Head-Driven Generation Algorithms ....................... 36
       3.1.1 Head-Driven Bottom-Up Generation .................. 37
       3.1.2 Semantic Head-Driven Generation ................... 38
       3.1.3 Chart-based Semantic Head-Driven Generation .... 39
   3.2 SAX and SGX .............................................. 39
       3.2.1 The SAX Chart Parser ................................ 40
       3.2.2 The SGX Head-Driven Generator .................... 42
       3.2.3 Combining SAX and SGX ............................ 43
3.3 An English Engine (DCG Version) ............................................. 44  
3.3.1 Comparison with Core Language Engine ................................. 44  
3.3.2 Semantic Head-Driven DCG Grammar ..................................... 46  
3.3.3 Event-based Logical Form .................................................. 46  
3.3.4 Lexicon and Lexical Rules ................................................. 48  
3.3.5 Robust Parsing and Generation ............................................ 49  
3.3.6 A Dialogue Interface ..................................................... 50  
3.3.7 DCG and HPSG ........................................................... 52  

4 Generation with HPSG and ProFIT: Delayed Lexical Choice ......... 54  
4.1 The ProFIT Typed Feature System ......................................... 54  
4.2 An HPSG English Engine ................................................... 55  
4.2.1 ProFIT with SAX and SGX ................................................. 56  
4.2.2 Two-stage Grammar Compilation ....................................... 57  
4.2.3 Lexicon Compilation ..................................................... 59  
4.2.4 Advantages of ProFIT/SAX over ALE .................................. 60  
4.3 An HPSG English Grammar .................................................. 63  
4.3.1 Sort Hierarchy .......................................................... 63  
4.3.2 Templates ............................................................... 66  
4.3.3 ID Schemata ............................................................. 68  
4.3.4 HPSG Principles ........................................................ 69  
4.4 Delayed Lexical Choice ..................................................... 72  
4.4.1 Monotonicity and Subsumption ......................................... 72  
4.4.2 Syntactic-Semantic Lexicon ............................................. 73  
4.4.3 Generation with Delayed Lexical Choice .............................. 74  
4.4.4 Reversible Delayed Lexical Choice .................................... 75  
4.4.5 Underspecification and Pronouns ...................................... 76  
4.5 Summary ................................................................. 78  

II Problems and Solutions in HPSG (Post-1994 Revisions) ............. 80  
5 Eliminating Empty Categories: Relative Clause Constructions ....... 81  
5.1 Problems in the 1994 HPSG Theory ........................................ 81  
5.2 Revisions to HPSG Theory since 1994 .................................... 82  
5.3 Implementing Sag 1997 ...................................................... 84  
5.3.1 Eliminating Traces by Lexical Rule .................................... 84  
5.3.2 Lexicalization of Nonlocal Features .................................. 86  
5.3.3 Phrase and Clause Type Hierarchy .................................... 87  
5.3.4 Phrase and Clause Type Constraints .................................. 89  
5.3.5 Relative Clause Constructions ....................................... 93  
5.4 Summary ................................................................. 98
6 Head-Driven Generation: Lexicalization of Context

6.1 Set-Valued Features ........................................ 100
  6.1.1 Phrasal Amalgamation .................................. 100
  6.1.2 Lexical Amalgamation .................................. 100
  6.1.3 Lexicalization of Quantifier Scoping .................. 101
  6.1.4 Lexicalization of Context ............................ 102
6.2 Semantic Heads ........................................... 103
  6.2.1 Quantificational-Semantic Heads ...................... 103
  6.2.2 Contextual-Semantic Heads ........................... 104
  6.2.3 Semantic Heads and Generation ....................... 106
6.3 Register Variation ........................................ 108
  6.3.1 Relative Pronouns ..................................... 108
  6.3.2 Clausal Constraints ................................... 109
  6.3.3 Lexical Constraints ................................... 110
  6.3.4 Interaction of Constraints ........................... 111
6.4 Implementing Lexicalization of Context .................. 113
  6.4.1 The BUG Bottom-Up Generator ....................... 113
  6.4.2 BUG with HPSG and ProFIT ......................... 114
  6.4.3 Quantifiers and Context ............................. 116
  6.4.4 Lexical Amalgamation in ProFIT ..................... 117
6.5 Summary .................................................. 120

7 Towards Incremental Generation: Minimal Recursion Semantics 121

7.1 Machine Translation ....................................... 122
  7.1.1 Generation from Logical Forms ....................... 122
  7.1.2 Generation from Lexical Signs ....................... 123
  7.1.3 Generation from Minimal Recursion Semantics ....... 123
7.2 Bag Generation ........................................... 124
  7.2.1 Phillips' Algorithm .................................. 124
  7.2.2 Implementing MRS in ProFIT ......................... 127
  7.2.3 The Semantics Principle ............................. 130
7.3 Incremental Generation ................................... 132
  7.3.1 Interactive Dialogues ................................ 132
  7.3.2 Incremental Generation with HPSG ................... 133
  7.3.3 Syntactic Constituency ............................... 135
7.4 Elliptical Generation .................................... 136
  7.4.1 Information Structure ............................... 136
  7.4.2 Generation from New Information .................... 137
8 Conclusion

8.1 Other Approaches ......................................................... 139
  8.1.1 Compiling HPSG to TAG ............................................. 139
  8.1.2 Abstract Machines for HPSG ......................................... 140
  8.1.3 Grammar Inversion ..................................................... 141
  8.1.4 Pre-compiling Phrasal Signs ......................................... 141
8.2 Evaluation of the Research ................................................ 142
  8.2.1 Generation with HPSG ................................................. 142
  8.2.2 HPSG Implementation ................................................. 142
  8.2.3 Use of Existing Algorithms ........................................... 143
  8.2.4 Reversibility ........................................................... 144
  8.2.5 Theoretical Changes .................................................. 144
8.3 Future Work ............................................................. 145

Bibliography ................................................................................. 146

Appendix A: Demonstration for Chapter 4 .................................. 152

Appendix B: Demonstration for Chapter 5 .................................. 168

Appendix C: Demonstration for Chapter 6 .................................. 177
Chapter 1

Introduction

1.1 Generation with HPSG

In natural language analysis, and other areas of computational linguistics, Head-Driven Phrase Structure Grammar (HPSG) (Pollard and Sag, 1987; 1994) has become one of the most widely used grammatical frameworks. In natural language generation, on the other hand, Systemic Functional Grammar (SFG) (Halliday, 1985) has been one of the most influential linguistic frameworks, as discussed by Matthiessen and Bateman (1991). It is interesting that using HPSG for generation has been almost as unpopular as using SFG for parsing.

1.1.1 A Trivial Matter?

At first sight it is surprising that HPSG is not widely used for generation. There are at least three general points which suggest that generation with HPSG should, apparently, be quite straightforward and could even be considered a trivial matter.

First, HPSG is a syntactic theory, so “generation with HPSG” refers primarily to syntactic or surface generation. Natural language generation is usually divided into strategic text planning and tactical surface realization (often with an intermediate microplanning component). Most research on natural language generation has tended to concentrate on strategic text planning. By contrast with the difficult and challenging problems involved in text planning, the details of tactical realization should apparently be relatively simple, whatever grammatical framework is used.

Second, researchers who work on algorithms for parsing and surface generation (for example the SHD semantic head-driven generation algorithm) tend to be primarily interested in efficiency. However, it is considered important that their proposed algorithms are “general”. This means that, although the algorithms are usually illustrated with simple grammars (usually simple DCGs), it is assumed that they can be used equally well with more complex grammars such as HPSG.

Third, researchers who work on grammar formalisms are usually satisfied if an implementation can parse with their formalism (if they are interested in implementation at all) and don’t worry about whether their formalism is useful for generation. However, in HPSG it is considered very important that the formalism is expressed in a purely declarative and non-procedural way. This is assumed to make the grammar useable in either direction, for generation or for parsing.
Therefore, given a declarative grammar and a general algorithm, it should apparently be simple
to do tactical surface generation. It should apparently be particularly straightforward to combine
a head-driven generation algorithm (such as the SHD algorithm) with a head-driven grammar
formalism (such as HPSG) to perform surface realization of simple phrases like *Kim walks*.

Surprisingly, it turns out that to generate even such a simple phrase as *Kim walks*, which is used
as an elementary example in Chapter 1 of the standard HPSG textbook, requires the solution of
fundamental theoretical difficulties. In fact, we do not even reach this point until Chapter 6 of this
thesis. We therefore reject the view that generation with HPSG is a trivial matter.

1.1.2 Generation from What?

Why have computational implementations of HPSG concentrated on parsing rather than genera-
tion? One basic problem is that, when generation with HPSG has been suggested, the first question
has usually been “Generation from what?”

It is relatively clear that parsing starts from a surface string. Before considering what genera-
tion should start from, perhaps we should examine parsing with HPSG. Some fundamental issues,
including the nature of HPSG grammar, the implementation of inheritance-based typed feature
structures, and the implementation of chart-based processing algorithms, will be equally funda-
mental in generation. We therefore start in Chapter 2 with a description of an implementation
of HPSG with the ALE typed feature system which does parsing only. We also introduce chart
processing in Chapter 3 with a description of the SAX chart parser before describing the related
SGX generator.

Basically, parsing ends when syntactic analysis of the string ends. Any semantic representations
built during syntactic analysis may then be extracted, interpreted and used for application purposes
(database queries, machine translation, interactive dialogues etc.). Various different semantic rep-
resentations have been developed for use with such applications, partly due to differences between
particular semantic theories and partly due to differences in the requirements of the applications
themselves.

The semantic theory which has long been associated with HPSG is Situation Semantics (Barwise
and Perry, 1983). Since the dominant paradigm of computational linguistics has made use of feature
structures for linguistic description and unification for linguistic processing, the standard semantic
representation in HPSG has been a feature structure version of Situation Semantics, in which the
principal operation of semantic composition has been implemented by unification of the semantic
features of the components. The first answer to what generation should start from is, therefore,
to generate from Situation Semantics representations.

One of the distinctive characteristics of HPSG is its emphasis on a fundamentally head-driven
organization of grammar. Primarily this is embodied in syntax, by identity of basic features
between head daughters and mothers, and by subcategorization for complements by syntactic
heads, but it is also embodied in semantics, by a clear definition of semantic heads and by identity
of semantic features between semantic head daughters and mothers. It can in fact be argued that
HPSG is naturally organized for generation using semantic head-driven algorithms. Therefore an
investigation of semantic head-driven generation, starting from representations based on Situation
Seman tics, constitutes the main body of the thesis. This investigation starts in Chapter 3, continues in Chapter 4, and reaches its conclusion in Chapter 6.

In machine translation however, “head-switching” between languages (when the syntactic or semantic head of a source language structure does not naturally transfer to the head of a translationally equivalent structure in the target language) means that a strongly head-driven approach to semantics is undesirable. The problem of logical form equivalence is also crucial for generation in machine translation. A flat, list-based semantic representation is therefore more suitable. Minimal Recursion Semantics (Copestake et al., 1997) has been developed specifically to provide such a flat representation for HPSG. For generation from such flat lists, we need a non-head-driven approach. We briefly investigate non-head-driven generation with HPSG, starting from Minimal Recursion Semantics representations, in Chapter 7.

In interactive dialogues, generation needs to start from an incomplete bag of semantic terms, and then needs to continue incrementally as more semantic terms are given. MRS relations are apparently suitable for use as semantic terms which can be added incrementally. We therefore investigate, also in Chapter 7, whether an HPSG grammar is suitable for use in incremental generation, starting from an incomplete bag of MRS relations and continuing incrementally as more MRS relations are given.

1.2 Aims of the Thesis

The main aim of the thesis can be described as an “exploration” into natural language generation with HPSG. We consider this to be an exploration, because how to do generation with HPSG is relatively unknown territory. We restrict the aim to surface generation or tactical realization, and do not discuss strategic text planning.

The second aim of the thesis is to investigate how to make satisfactory computational implementations of HPSG. We go about the exploration of natural language generation by means of a series of implementations, which are described in the successive chapters. We therefore aim to find out what tools and techniques are effective for such implementations. However, the thesis is not particularly concerned with the efficiency of the implementations, and does not present comparative measurements of speed and so on (though the implementations with ProFIT and SAX/SGX are rather fast).

The third aim of the thesis is to answer the question: Can existing generation algorithms be used with HPSG grammars? Rather than being concerned with the relative efficiency of the algorithms, we aim to address more basic problems concerning whether HPSG grammars are suitable for use with particular generation algorithms.

The fourth aim of the thesis is to investigate the reversibility of HPSG grammars. The implementation issues are relevant to parsing as well as generation, and as far as possible it is desirable to make implementations which can perform both parsing and generation. This naturally leads to the question of reversibility.

An important question is whether it is necessary to make any changes to HPSG grammatical theory in order to support generation or to ensure reversibility. In Chapter 6 we propose a change to
HPSG theory, in order to solve the problems with generation of phrases like *Kim walks*, mentioned above. This chapter aims to present theoretical, computational and linguistic justifications for the proposed change.

As far as different approaches to generation are concerned, the main aim of the thesis is to investigate semantic head-driven generation with HPSG quite thoroughly. We also investigate non-head-driven generation from flat lists such as Minimal Recursion Semantics representations, and this leads to an investigation of whether HPSG can be used for incremental generation. However, these aims are secondary to the investigation of head-driven generation.

### 1.3 Structure of the Thesis

The thesis has two main parts. After this introductory chapter, **Part I** describes a series of three practical implementations in Chapters 2, 3 and 4. These are all based on the 1994 textbook version of HPSG theory presented by Pollard and Sag (1994). **Part II** describes three comparatively experimental implementations in Chapters 5, 6 and 7. These are all based on significant revisions of HPSG theory in the period 1994-98.

**Chapter 2** describes a fairly full implementation of the 1994 textbook. Since the textbook (Pollard and Sag, 1994) is widely known, we do not go over its details. We take this textbook version of HPSG as the assumed background and starting point for the thesis. The purpose of the chapter is to describe this background by summarising the implementation which I completed in 1994 using ALE 2.0. The aim is not to describe the syntactic features and so on, which are fully described in the textbook, but to describe the implementation. Unfortunately, ALE 2.0 provided only a parser and not a generator, so this implementation can only be used for parsing.

**Chapter 3** describes a second implementation, which (by contrast) can do generation successfully, but in which a DCG grammar approximates to HPSG theory to only a limited extent. In Chapter 3 we discuss head-driven generation algorithms, and why head-driven generation is a natural approach to generation with HPSG. We introduce the SAX parser and the SGX head-driven generator, which we will use throughout Chapters 3, 4 and 5. We then describe a natural language engine called English Engine, which uses SAX and SGX for parsing and generation with a DCG grammar which includes many of the features and mechanisms of HPSG. Finally we mention an adaptation of English Engine as a natural language dialogue interface, which raises issues about using HPSG in a dialogue framework which we will discuss in Chapter 7.

**Chapter 4** describes a third implementation, which aims to combine the strengths of the first two into a single framework. In the system of Chapter 2 the grammar conforms closely to HPSG theory but the system can only be used for parsing, and the system in Chapter 3 can do generation successfully but its DCG grammar approximates to HPSG theory to only a limited extent. In Chapter 4, a completely revised version of English Engine performs efficient parsing and generation using SAX and SGX, but combines them with an HPSG grammar based on a typed feature structure representation, using the ProFIT typed feature system. Chapter 4 also describes an implementation of delayed lexical choice in generation with HPSG, which exploits subsumption in the sort hierarchy.
Chapter 5 describes some relevant changes in HPSG theory after the 1994 textbook version. Several revisions which occurred in the period 1994-98 are important for the work described in Part II of the thesis. This chapter describes an implementation of the proposals of Sag (1997), which eliminate all empty categories from HPSG theory and give a new analysis of English relative clauses using a hierarchy of construction types. This has a specific importance for the implementations in this thesis, as SAX and SGX do not permit empty categories and hence the grammar in Chapter 4 cannot handle relative clauses. The fourth implementation, described in Chapter 5, handles all forms of English relative clause using the new HPSG theory without empty categories.

Chapter 6 proposes a significant change to HPSG theory: the lexicalization of context. The change is proposed in order to strengthen the role of semantic heads in the overall organization of HPSG grammar. It is also needed to solve fundamental problems with the inclusion of contextual information in head-driven generation. This chapter presents theoretical, computational and linguistic justifications for the proposed change. These include a brief sketch of an analysis of register variation within Sag’s new analysis of English relative clauses. The chapter also describes a basic implementation of the new theory.

Chapter 7 briefly investigates non-head-driven generation with HPSG, starting from Minimal Recursion Semantics representations. We show how an existing bag generation algorithm, developed for use with categorial grammar and indexed logical form, can also be used with HPSG and Minimal Recursion Semantics. In this chapter we also describe an approach to incremental generation with HPSG. In contrast to categorial grammar, HPSG has some fundamental difficulties with highly incremental generation. However, we suggest a chart-based solution to the problem.

Chapter 8 is the conclusion, which briefly mentions some other approaches and related work. The contributions of the thesis are then evaluated, and future work is proposed.

At the end of the thesis, demonstration tests of the implementations described in Chapters 4, 5 and 6 are shown in Appendices A, B and C respectively.

1.4 Original Contributions of the Thesis

The thesis makes contributions both in practical implementations and in HPSG theory. For example, my 1994 implementation of the HPSG textbook, described in Chapter 2, is still one of the fullest ALE implementations of HPSG theory.

The combination of the SAX parser, the SAX morphological analyzer, and the SGX generator to make a practical natural language engine, as described in Chapter 3, is an original contribution. Previously SAX had been used for parsing Japanese, with the JUMAN Japanese morphological analyzer. This was the first combination of SAX with SGX, as well as the first application of SAX and SGX to English.

The combination of SAX and SGX with ProFIT to make a practical HPSG-based parsing and generation engine, as described in Chapter 4, is an original contribution. ProFIT has also been used by Lappin and Shih (1996) for parsing English with HPSG in the ELLIP project, but not for generation. This method of combining SAX and SGX with ProFIT has subsequently been adopted by Miyata and Matsumoto (1998) for parsing and generation of Japanese legal texts.
The implementation of HPSG lexical rules so that they are used on-line during morphological analysis is an original contribution. This approach allows much faster compiling of the lexicon than in ALE, as described in Chapter 4. For practical grammar development, the slowness of grammar compilation with ALE has been a major bottleneck, which this approach eliminates.

The implementation of delayed lexical choice in generation with HPSG, described in Chapter 4, is an original contribution which extends and generalises previous work. The established approach to delayed lexical choice is to replace the morphological lexicon with a lexicon of stems for the syntactic generation stage. Our method is more flexible, specifying a syntactic-semantic lexicon in place of a lexicon of stems. One advantage of this approach is that it can be applied to classes of words which do not have any kind of stems, such as pronouns. Another advantage is that it can be applied in order to delay choices for any kind of feature, not only for morphological forms.

The use of reversible delayed lexical choice in parsing with HPSG, also described in Chapter 4, is an original contribution. The technique developed for delayed lexical choice in generation can be applied to parsing, allowing deliberate underspecification of selected features for robust parsing.

The implementation of the proposals of Sag (1997), described in Chapter 5, is an original contribution. The theory, eliminating empty categories from HPSG and analysing English relative clauses using a hierarchy of construction types, is of course Sag’s theory, but the implementation in ProFIT is a contribution of the thesis.

The lexicalization of context, proposed in Chapter 6, is a significant original contribution to HPSG theory. The change is proposed in order to strengthen the role of semantic heads in the overall organization of HPSG grammar. It is also needed to solve fundamental problems with the inclusion of contextual information in head-driven generation. The thesis presents theoretical, computational and linguistic justifications for the proposed change.

The brief sketch of an analysis of register variation within Sag’s new analysis of English relative clauses, presented in Chapter 6, is also an original contribution to HPSG theory.

The demonstration that several existing algorithms can be used with HPSG grammars is an original contribution. The use of SAX/SGX with HPSG is shown in Chapter 4, van Noord’s BUG algorithm and Phillips’ generation algorithm are used with HPSG in Chapters 6 and 7.

The approach to incremental generation with HPSG described in Chapter 7 is an original contribution. A further contribution is the suggested chart-based solution to the problem which HPSG, in contrast to categorial grammar, has with highly incremental generation.

1.5 Published Material

Some of this material has appeared in papers at international conferences. Chapter 4 was presented at HPSG-96 (Wilcock and Matsumoto, 1996a) and COLING-96 (Wilcock and Matsumoto, 1996b). Chapter 6 was presented at HPSG-97 (Wilcock, 1997) and COLING-ACL-98 (Wilcock and Matsumoto, 1998). Chapter 7 was presented at INLG-98 (Wilcock, 1998).

The proposal for lexicalization of context as a revision of HPSG theory (part of Chapter 6) has been published in Lexical and Constructional Aspects of Linguistic Explanation, G. Webelhuth, J.-P. Koenig and A. Kathol (eds), Stanford: CSLI Publications (Wilcock, 1999).
Part I

Towards Generation with HPSG
(1994 Theory)
Chapter 2

Parsing with HPSG and ALE: Implementing the Textbook

2.1 HPSG in Pollard and Sag 1994

HPSG gradually became widely used in both theoretical and computational linguistics in the years following 1987, when “Volume I” of the HPSG textbook (Pollard and Sag, 1987) was published. By the early 1990’s, draft versions of the intended “Volume 2” were in widespread use in photocopied form, and HPSG had become perhaps the dominant grammar theory in computational (but not theoretical) linguistics. When the intended “Volume 2” was finally published in 1994, due to the extensive revisions since 1987 it was simply entitled “Head-Driven Phrase Structure Grammar” (Pollard and Sag, 1994), and the version of HPSG it describes has been regarded as the standard version of HPSG.

Since the textbook (Pollard and Sag, 1994) is widely known, we will not go over its details. We will take this textbook version of HPSG as the assumed background and starting point for the thesis. In Chapter 5 we will describe changes in HPSG theory from the 1994 textbook version, as several revisions which occurred in the period 1994-98 are important for the work described in the rest of the thesis. The purpose of this Chapter is to describe the background by summarising the fairly full implementation of the 1994 textbook version of HPSG, which I completed in 1994 using ALE 2.0, the best available software engine for HPSG implementation. The aim is not to describe the syntactic features and so on, which are fully described in the textbook, but to describe the implementation.

2.2 The ALE Typed Feature System

Based on the underlying logic of typed feature structures described by Carpenter (1992), the Attribute Logic Engine of Carpenter and Penn (1994; 1998) provides a complete framework for implementing HPSG grammars. More specifically, the ALE system was designed to provide all the facilities required to implement the HPSG theory presented by Pollard and Sag (1994). The ALE system therefore provides not only a mechanism for unification of typed feature structures
following a type hierarchy declaration, but also a format for lexical entries, a format for empty
categories, a format for lexical rules, a format for grammar rules, a definite clause programming
language, a constraint specification language, and a built-in chart parser. The ALE system is itself
implemented in Prolog, and the HPSG implementation described here was done using ALE 2.0
(Carpenter and Penn, 1994) with SICStus Prolog 2.9. This implementation still works perfectly
today, using ALE 3.0 released in 1998 (Carpenter and Penn, 1998), with SICStus Prolog 3.5 (SICS,
1995).

As ALE 2.0 did not include a generator, my 1994 implementation was intended only for parsing.
A head-driven generator has been added in the new version, ALE 3.0. How to use HPSG for gen-
eration is, of course, the subject matter of the rest of the thesis. In particular, any implementation
of head-driven generation with ALE 3.0 will need to take account of the issues raised in Chapter 6.

In order to include both an English HPSG grammar and a Japanese JPSG grammar (Gunji,
1987) in the implementation, the root `bot of the type hierarchy includes a subroot `hpsg_bot for an
HPSG-specific type hierarchy, and a subroot `jpsg_bot for a JPSG-specific type hierarchy, as shown
in Figure 2.1. The root also includes the “universal” types `bool, `set and `list.

% Universal Signature
% ================

bot sub [univ_bot, hpsg_bot, jpsg_bot].

univ_bot sub [bool, list, set].

  bool sub [plus, minus].
    plus sub [].
    minus sub [].

  set sub [e_set, ne_set, h_set, j_set].
    e_set sub [].
    ne_set sub [h_ne_set, j_ne_set]
      intro [elt:bot, elts:set].

  list sub [e_list, ne_list, h_list, j_list].
    e_list sub [].
    ne_list sub [h_ne_list, j_ne_list]
      intro [hd:bot, tl:list].

Figure 2.1: The Root of the Type Hierarchy

2.3 Implementing the Textbook

The rest of this chapter presents some illustrative fragments from my 1994 ALE implementation
of Pollard and Sag (1994). The examples are presented according to the first six chapters of the
book. (The three final chapters of the book are mainly concerned with refinements and revisions
of the material in the first six chapters).

The source code of the implementation is intended to be self-documenting, and the grammatical
details are intended to be those described by Pollard and Sag (1994), so further explanation is kept
to a minimum.

2.3.1 A System of Signs

The root of the HPSG type hierarchy is shown in Figure 2.2. Instead of a sign having just two subtypes word and phrase, there is a slight complication to include trace and clitic as well. Each of the six ID schemata is declared as a distinct subtype of phrase.

```%
% HPSG Signature
%
```

```hpsg_bot sub [sign, con_struc, synsem, local, nonloc, nonlocal, category, part_of_speech, modifier, marking, case, vform, pform, content, relation, gender, number, person, context, contextual_indices, register].

sign sub [word, non_word]
    intro [synsem: synsem, qstore: quant_set, retrieved: quant_list].
word sub [].
non_word sub [phrase, trace, clitic].
phrase sub [schema1, schema2, schema3, schema4, schema5, schema6]
    intro [dtrs: con_struc].
    schema1 sub [].
    schema2 sub [].
    schema3 sub [].
    schema4 sub [].
    schema5 sub [].
    schema6 sub [].
trace sub [].
clitic sub [].

con_struc sub [coord_struc, head_struc].
coord_struc sub [].
head_struc sub [head_comp_struc, head_adj_struc, head_mark_struc, head filler_struc]
    intro [head_dtr: sign, comp_dtrs: list].
head_comp_struc sub [].
head_adj_struc sub []
    intro [comp_dtrs: e_list, adj_dtr: sign].
head_mark_struc sub []
    intro [comp_dtrs: e_list, marker_dtr: sign].
head_filler_struc sub []
    intro [comp_dtrs: e_list, filler_dtr: sign].

synsem sub []
    intro [loc: local, nonloc: nonloc].
local sub []
    intro [cat: category, cont: content, conx: context].
nonloc sub []
    intro [inher: nonlocal, to_bind: nonlocal].
nonlocal sub []
    intro [que: npro_set, rel: ref_set, slash: loc_set].
```

Figure 2.2: Part of the HPSG Type Hierarchy
Immediate dominance schemata are introduced by Pollard and Sag (1994) in Chapter 1. In ALE the abstract schemata must be approximated by phrase structure rules. ALE provides a fixed format for grammar rules. An example rule in the ALE format, which implements the HPSG Schema 1, is shown in Figure 2.3. Goals are added to the rule, to execute ALE definite clause procedures during parsing, which check all the relevant HPSG principles.

```
schema1_S_NP_VP rule
(Phrase, schema1,
  dtrs:(head_comp_struc, head_dtr:HeadDtr,
       comp_dtrs:[CompDtr]))
  ==>
  cat>
  (CompDtr, phrase,
   synsem:loc:cat:head:noun),
  cat>
  (HeadDtr, phrase,
   synsem:loc:cat:head:(verb, vform:fin)),
  goal>
  subcat_principle(HeadDtr, [CompDtr], Phrase),
  goal>
  saturated(Phrase),
  goal>
  trace_principle(HeadDtr, [CompDtr]),
  goal>
  english_trace_constraint(HeadDtr, [CompDtr]),
  goal>
  quantifier_retrieval([HeadDtr, CompDtr], Phrase),
  goal>
  hpsg_principles([HeadDtr, CompDtr], Phrase),
  goal>
  english_inv_condition(inv:minus, Phrase),
  goal>
  singleton_rel_constraint(Phrase),
  goal>
  relative_uniqueness_principle([HeadDtr, CompDtr], Phrase),
  goal>
  clausal_rel_prohibition(Phrase).
```

Figure 2.3: ALE Grammar Rule for HPSG Schema 1

Pollard and Sag (1994) also introduce the Subcategorization Principle and the Head Feature Principle in Chapter 1. Some examples of the ALE definite clauses which implement these principles are shown in Figure 2.4.
hpsg_principles([HeadDtr|CompDtrs], Phrase) if
head_feature_principle(HeadDtr, Phrase),
marking_principle(HeadDtr, Phrase),
semantics_principle([HeadDtr|CompDtrs], Phrase),
spec_principle(CompDtrs, HeadDtr),
ncnlocal_feature_principle([HeadDtr|CompDtrs], Phrase),
binding_principles(HeadDtr, CompDtrs),
conx_consistency_principle([HeadDtr|CompDtrs], Phrase),
conx_indices([HeadDtr|CompDtrs], Phrase).

% Head Feature Principle
% head_feature_principle(HeadDtr, Phrase).

head_feature_principle(synsem:loc:cat:head:Head,
synsem:loc:cat:head:Head) if
true.

% English Inversion Condition (schema3 plus, others minus).
% english_inv_condition(inv:Inv, Phrase).

english_inv_condition(inv:Inv,
synsem:loc:cat:head:(verb, V)) if
true.

% Marking Principle
% marking_principle(Head_or_MarkerDtr, Phrase).

marking_principle((MarkerDtr, synsem:loc:cat:(head:marker,
marking:M)),
(synsem:loc:cat:marking:(M, marked),
dtrs:marker_dtr:MarkerDtr)) if
true.

marking_principle((HeadDtr, synsem:loc:cat:marking:M),
(synsem:loc:cat:marking:(M, unmarked),
dtrs:head_dtr:HeadDtr)) if
true.

Figure 2.4: Definite Clauses for some HPSG Principles
2.3.2 Agreement

We show details of agreement with the example *she walks*. Using the ALE macro facility, a macro `e_ppro` for English personal pronouns is defined in Figure 2.5. The macro has 3 parameters for the pronoun’s case, its agreement features, and its contextual background conditions. The lexical entry for *she* is specified by using the macro with suitable parameters, as shown in Figure 2.5. Of course, the agreement features for *she* include feminine gender.

```
 e_ppro(Case, Agr, Reln) macro
doord,
synsem:loc:(cat:(head:(noun, case:Case,
  mod:no_mod),
  subcat:[]),
  cont:(ppro, index:(Agr,Tag),
    restr:(elt:(nucleus:(Restr, inst:Tag,
      reln:Reln),
        quants:e_list),
      elts:e_set)),
  conx:backgr:(elt:Restr,
    elts:e_set)),
  @ nonlocal_free,
  @ quantifier_free.
```

```
 she ---> @ e_ppro(nom, (gend:fem, num:sing, pers:third), female).
```

Figure 2.5: A Macro for Personal Pronouns

The lexical entry for *walks* is not specified directly, but is derived from the base form by lexical rule. The base form *walk* is specified by means of a macro `e_iv` for simple intransitive verbs, shown in Figure 2.6. The agreement features for the subject in the subcat list are left unspecified.

```
e_iv(Reln, Arg1:Sort) macro
  @ e_verb_bse,
synsem:loc:((cat:subcat:[[@ e_np(nom, _, Tag1, Sort)]],
  cont:nucleus:(reln:Reln,
    Arg1:Tag1))).
```

```
 walk ---> @ e_iv(walk, walker:animate).
```

Figure 2.6: A Macro for Simple Intransitive Verbs

During compilation of the lexicon, ALE expands the `e_ppro` macro to produce the full lexical entry for *she* in the compiled lexicon, and expands the `e_iv` macro to produce a full lexical entry for *walk*. Lexical entries in the compiled lexicon can be displayed using the ALE `lex` command. The compiled entry for *she* is displayed in Figure 2.7.
WORD: she
ENTRY:

Figure 2.7: Lexical Entry for she
AMES provides a fixed format for lexical rules. An example is shown in Figure 2.8. The rule derives finite 3rd person singular verb forms from base forms, specifying third singular index agreement for the subject in the subcat list of the derived entry, but leaving gender unspecified.

\begin{verbatim}
BSE_TO_FIN_THIRD_SING_LEX_RULE
  (word,
   synsem:(loc:(cat:(head:(vform:bse,
     aux:Aux,
     inv:Inv,
     prd:bool),
     subcat:[]
     cont:[]
     restr:Restr,
     conx:Cont),
   subcat:[]
     cont:Cont,
     conx:(backgr:Backgr,
     c_inds:C_Inds)),
   nonloc:Nonloc),
   qstore:Qstore,
   retrieved:Retrieved)
  **>
  (word,
   synsem:(loc:(cat:(head:(vform:fin_pres,
     aux:Aux,
     inv:Inv,
     mod:no_mod,
     prd:minus),
     subcat:[]
     cont:[]
     restr:Restr,
     conx:Cont, nucleus:locn:EventLocn),
   subcat:[]
     cont:Cont, nucleus:locn:EventLocn),
   nonloc:Nonloc),
   qstore:Qstore,
   retrieved:Retrieved)
  
  MORPHS
  be becomes is,
  buy becomes buys,
  have becomes has,
  (X,o) becomes (X,oes),
  (X,y) becomes (X,ies),
  X becomes (X,s).
\end{verbatim}

Figure 2.8: Lexical Rule for 3rd Singular Verb

AMEs applies lexical rules during compilation of the lexicon. In Chapter 4 we will argue that this is wrong, both in practice and in theory. From the lexical entry for walk, AMEs derives a separate lexical entry for walks by applying the lexical rule. The resulting entry for walks in the compiled lexicon is shown in Figure 2.9, using the AMEs lex command.
Figure 2.9: Lexical Entry for walks
Note that in the lexical entry for *walks* the agreement features, on the INDEX of the subject in the SUBCAT feature, include unspecified gender (GEND gender). When the input sentence *she walks* is parsed using the ALE rec command, as shown in Figure 2.10, the result includes feminine gender (GEND fem) in the agreement features of WALKER.

```
schema
QSTORE e_set
RETRIEVED e_list
SYNSEM symsem
  LOC local
    CAT category
      HEAD [16] verb
      AUX minus
      INV minus
      VFORM fin_pres
      SUBCAT [16] e_list
    CONT qsoa
      NUCLEUS [13] walk_soa
        LOCN [8] location_ref
        RELN walk
        WALKER [0] ref
        GEND fem
        NUM sing
        PERS third
      QUANTS [14] e_list
    CONX context
      BACKGR ne_psca_set
        ELT [15] tense_soa
          RELN temporally_overlaps
          TIME1 [8]
          TIME2 [9] location_ref
        ELTS ne_psca_set
          ELT [1] inst_psca
          INST [0]
          RELN female
        ELTS e_set
      C_IND [6] contextual_indices
        ADDRESSEE addressee_ref
        SPEAKER speaker_ref
      U_LOC [9]
    NONLOC nonloc
    INHER nonlocal
      QUE e_set
      REL e_set
      SLASH e_set
    TG_BIND nonlocal
      QUE npro_set
      REL ref_set
      SLASH loc_set
```

Figure 2.10: Parsing *she walks*
2.3.3 Complement Structures

Lexical entries for verbs with a range of different complement structures, including subject and object equi and raising verbs described in (Pollard and Sag, 1994) Chapter 3, are implemented by ALE macros as shown in Figure 2.11.

\[
\text{hope} \rightarrow \text{e_tv_subj_equi_vp}(\text{hope, hoper:human, hoped:psoa, inf}).
\]

\[
\text{try} \rightarrow \text{e_tv_subj_equi_vp}(\text{try, tryer:animate, tried:psoa, inf}).
\]

\[
\text{try} \rightarrow \text{e_tv_subj_equi_vp}(\text{try, tryer:animate, tried:psoa, ger}).
\]

\[
\text{e_tv_subj_equi_vp}(\text{Reln, Arg1:Sort1, Arg2:psoa, VP_Vform}) \text{ macro}
\]
\[
\text{e_verb_bse,}
\]
\[
\text{synsem:loc:}(\text{cat:subcat:}[(\text{e_np(nom, Tag1, Sort1)}),
\text{ (e_vp(VP_Vform, e_np(_-,Tag1, Sort1))},
\text{ loc:cont:Soa})],
\text{ cont:nucleus:}(\text{reln:Reln,}
\text{ Arg1:Tag1,}
\text{ Arg2:Soa, psoca})).
\]

\[
\text{persuade} \rightarrow \text{e_ditv_obj_equi_inf}(\text{persuade, persuader:human,}
\text{ persuadee:human, persuasion:psoa}).
\]

\[
\text{e_ditv_obj_equi_inf}(\text{Reln, Arg1:Sort1, Arg2:Sort2, Arg3:psoa}) \text{ macro}
\]
\[
\text{e_verb_bse,}
\]
\[
\text{synsem:loc:}(\text{cat:subcat:}[(\text{e_np(nom, Tag1, Sort1)}),
\text{ (e_np(acc, Tag2, Sort2))},
\text{ (e_vp(inf, e_np(_-,Tag2, Sort2))},
\text{ loc:cont:Soa})],
\text{ cont:nucleus:}(\text{reln:Reln,}
\text{ Arg1:Tag1,}
\text{ Arg2:Tag2,}
\text{ Arg3:Soa, psoca})).
\]

\[
\text{believe} \rightarrow \text{e_tv_obj_raising_inf}(\text{believe, believer:human,}
\text{ believed:entity}).
\]

\[
\text{e_tv_obj_raising_inf}(\text{Reln, Arg1:Sort1, Arg2:Sort2}) \text{ macro}
\]
\[
\text{e_verb_bse,}
\]
\[
\text{synsem:loc:}(\text{cat:subcat:}[(\text{e_np(nom, Tag1, Sort1)}),
\text{ (e_np(acc, Tag2, Sort2))},
\text{ (e_vp(inf, e_np(_-,Tag2, Sort2))},
\text{ loc:cont:Soa})],
\text{ cont:nucleus:}(\text{reln:Reln,}
\text{ Arg1:Tag1,}
\text{ Arg2:Soa, psoca})).
\]

Figure 2.11: Lexical Entries of Equi and Raising Verbs

Successful parsing of \textit{I hope to persuade you to try to believe him to be happy} with 4 “control” verbs, is shown in Figure 2.12. The semantic representation in CONT (CONTENT) shows that HOPER is correctly coindexed with PERSUADER, both being SPEAKER in CONX (CONTEXT), and PERSUADEE is correctly coindexed with TRYER and BELIEVER, all being ADDRESSEE in CONX.
Figure 2.12: Parsing Equi and Raising Verbs
2.3.4 Unbounded Dependencies

The version of HPSG theory in Pollard and Sag (1994) includes empty categories, that is, invisible constituents or phonetically null elements. The revisions to HPSG proposed by Sag (1997), which we describe in Chapter 5, eliminate all empty categories from the theory.

As a tool designed for the 1994 theory, ALE supports empty categories, providing a special format to declare them as lexical entries. The basic empty category in the 1994 theory is the trace element, which has the special lexical entry shown in Figure 2.13.

```plaintext
empty
trace,
  synsem:(loc:Local,
  nonloc:(inh:e_set,
    rel:e_set,
    slash:(elt:Local,
      elts:e_set)),
  to_bind:(que:e_set,
    rel:e_set,
    slash:e_set)),
@ quantifier_free.
```

Figure 2.13: Lexical Entry for Trace

When initializing the chart for parsing, ALE has to insert every distinct empty category at every position in the chart. Without restrictions, this would clearly cause combinatorial explosion. The HPSG Trace Principle and the English Trace Constraint of Pollard and Sag (1994) Chapter 4 prevent traces being combined in syntactic constituents unless they are strictly subcategorized for by a substantive head. These constraints are implemented by definite clauses as shown in Figure 2.14. The constraints are included in the grammar rules by goals, as was shown in Figure 2.3.

```plaintext
% trace_principle(HeadDtr, CompDtrs).
trace_principle(_, []) if
  !, true.
trace_principle(HeadDtr, (hd:trace, tl:Rest)) if
  !, substantive(HeadDtr),
  trace_principle(HeadDtr, Rest).
trace_principle(HeadDtr, tl:Rest) if
  trace_principle(HeadDtr, Rest).
% english_trace_constraint(HeadDtr, CompDtrs).
english_trace_constraint(_, []) if
  !, true.
english_trace_constraint(HeadDtr, (hd:(Trace,trace), tl:Rest)) if
  !,
  strict_subcat(HeadDtr, Trace), % English trace constraint
  english_trace_constraint(HeadDtr, Rest).
english_trace_constraint(HeadDtr, (hd:-, tl:Rest)) if
  english_trace_constraint(HeadDtr, Rest).
```

Figure 2.14: Definite Clauses for Trace Principle

The traces are passed up to higher nodes by the inheritance of SLASH features, which is con-
trolled by the Nonlocal Feature Principle shown in Figure 2.15. The mechanism for handling
nonlocal features in Pollard and Sag (1994) is quite clumsy. There are three features, QUE, REL
and SLASH, but two distinct sets of these features need to be handled, the INHERITED set and
the TO-BIND set. The features in the INHERITED set are formed by set union of the features
of the INHERITED sets of all the daughters, minus any elements from the TO-BIND sets of the
Head Daughter. This whole mechanism is simplified in the revised theory described in Chapter 5.

```prolog
% nonlocal_feature_principle([HeadDtr|NonheadDtrs], Phrase).
nonlocal_feature_principle([HeadDtr|NonheadDtrs],
    synsem:nonloc:inher:PhraseInher) if
  union_inher([HeadDtr|NonheadDtrs], DtrsInher),
  subtract_to_bind(HeadDtr, DtrsInher, PhraseInher).
union_inher([], (que:e_set, rel:e_set, slash:e_set)) if
  !, true.
union_inher((hd:synsem:nonloc:inher:(que:Q1, rel:R1, slash:S1),
               tl:Rest),
            (que:Que, rel:Rel, slash:Slash)) if
  union_inher(Rest, (que:Qs, rel:Rs, slash:Ss)),
  set_union(Q1, Qs, Que),
  set_union(R1, Rs, Rel),
  set_union(S1, Ss, Slash).
subtract_to_bind(
    synsem:nonloc:to_bind:(que:TBQ, rel:TBR, slash:TBS),
    (que:Qs, rel:Rs, slash:Ss),
    (que:Que, rel:Rel, slash:Slash)) if
  set_subtract(Qs, TBQ, Que),
  set_subtract(Rs, TBR, Rel),
  set_subtract(Ss, TBS, Slash).
```

Figure 2.15: Definite Clauses for Nonlocal Feature Principle

The simplest unbounded dependency construction described by Pollard and Sag (1994) in Chap-
ter 4 is the topicalization filler-gap construction. This is handled by Schema 6, the Head-Filler
schema, shown in Figure 2.16. Note that the Head Daughter has a non-empty TO-BIND SLASH
feature.

The Head-Filler schema licenses topicalizations such as Bagels, I like. Here we show one possible
result of parsing the man he saw, analysed as a topicalization in Figure 2.17. In the next section
we will also use the man he saw as an example of a noun phrase with a relative clause modifier.
\% Schema6 licenses X --> Filler Head

schema6 rule
(Phrase, schema6,
dtrs: (head_filler_struc, head_dtr:HeadDtr,
    filler_dtr:FillerDtr))

==>  
  cat>
  (FillerDtr, phrase,
synsem:loc:Filler),
cat>
  (HeadDtr, phrase,
synsem: (loc:cat: (head:vform:fin,
        subcat:[]),
        nonloc: (inher:slash:elt:Filler,
to_bind: slash: (elt:Filler, elts:e_set))),
goal>
  subcat_principle(HeadDtr, [], Phrase),
goal>
  head_feature_principle(HeadDtr, Phrase),
goal>
  english_inv_condition(inv:minus, Phrase),
goal>
  marking_principle(HeadDtr, Phrase),
goal>
  quantifier_retrieval([HeadDtr, FillerDtr], Phrase),
goal>
  semantics_principle([HeadDtr, FillerDtr], Phrase),
goal>
  spec_principle([FillerDtr], HeadDtr),
goal>
  nonlocal_feature_principle([HeadDtr, FillerDtr], Phrase),
goal>
  trace_principle(HeadDtr, []),
goal>
  english_trace_constraint(HeadDtr, []),
goal>
  singleton_rel_constraint(Phrase),
goal>
  relative_uniqueness_principle([HeadDtr, FillerDtr], Phrase),
goal>
  clausal_rel_prohibition(Phrase),
goal>
  conx_consistency_principle([HeadDtr, FillerDtr], Phrase),
goal>
  conx_indices([FillerDtr, HeadDtr], Phrase).

Figure 2.16: Grammar Rule for Head-Filler Schema 6

28
?- rec [the, man, he, saw].

...
2.3.5 Relative Clauses

The analysis of English relative clauses in Pollard and Sag (1994) Chapter 5 involves more empty categories, the so-called null relativizers. This analysis has been completely replaced by Sag (1997), which we describe in Chapter 5, in which null relativizers as well as traces are completely eliminated.

The theory of Pollard and Sag (1994) Chapter 5 posits two different null relativizers for English. The “First Null Relativizer” is used in Wh-relative clauses, as in the man whom he saw. The “Second Null Relativizer” is used in non-Wh-relative clauses, as in the man he saw. To save space, only the implementation in ALE of the second one is shown, in Figure 2.18.

```
empty
word,
   synsem:(loc:(cat:(head:(relativizer,
      mod:(loc:(cat:head:noun,
         cont:(npro, index:Tag,
            restr:(elt:nucleus:(inst:Tag,
               reln:Sort),
               elts:Rest))),
         ncnloc:to_bind:rel:(elt:Tag, elts:e_set))),
         subcat:[(@ scomp(_, fin),
            loc:cont:RelCont,
            ncnloc:inher:slash:(elt:(Slash, cont:npro),
               elts:e_set))],
      cont:(npro, index:Tag,
         restr:(elt:nucleus:(inst:Tag,
            reln:Sort),
            quants:e_list),
         elts:(elt:RelCont,
            elts:Rest)))),
   ncnloc:(inher:(que:e_set,
      rel:(elt:Tag, elts:e_set),
      slash:e_set),
   to_bind:(que:e_set,
      rel:e_set,
      slash:(elt:(Slash, cat:head:noun,
         cont:index:Tag),
      elts:e_set))))),
@ background_free,
@ quantifier_free.
```

Figure 2.18: Lexical Entry for 2nd Null Relativizer

The result of parsing the man he saw, using this second null relativizer together with the theory of Pollard and Sag (1994) Chapter 5, is shown in Figure 2.19. The CONT (CONTENT) here is structure-shared (as shown by tag [3]) with the RESTIND (restricted index) of the quantifier in QSTORE.
Figure 2.19: Noun Phrase with Relative Clause
The Binding Principles of Pollard and Sag (1994) Chapter 6 are implemented by definite clauses, as shown in Figure 2.20. The constraints are included in the grammar rules by goals, as was shown in Figure 2.3.

binding_principles((HeadDtr, synsem:loc:cat:subcat:Subcat),
    (hd:synsem:Z, tl:Rest)) if
hpSG_pron(Z),
\+ locally_o_commands(_, Z, Subcat), \% ana or ppro, not locally o-commanded
!,
binding_principles(HeadDtr, Rest).
binding_principles((HeadDtr, synsem:loc:cat:subcat:Subcat),
    (hd:synsem:Z, tl:Rest)) if
hpSG_ana(Z),
locally_o_commands(Y, Z, Subcat),
!,
coindexed(Y, Z), \% Binding Principle A: locally o-bound ana.
binding_principles(HeadDtr, Rest).
binding_principles((HeadDtr, synsem:loc:cat:subcat:Subcat),
    (hd:synsem:Z, tl:Rest)) if
hpSG_ppro(Z),
locally_o_commands(Y, Z, Subcat),
!,
coindexed(Y, Z), \% Binding Principle B: locally o-free ppro.
binding_principles(HeadDtr, Rest).
binding_principles((HeadDtr, synsem:loc:cat:subcat:Subcat),
    (hd:synsem:Z, tl:Rest)) if
hpSG_npro(Z),
\+ o_commands(_, Z, Subcat), \% npro, not o-commanded
!,
binding_principles(HeadDtr, Rest).
binding_principles((HeadDtr, synsem:loc:cat:subcat:Subcat),
    (hd:synsem:Z, tl:Rest)) if
hpSG_npro(Z),
o_commands(Y, Z, Subcat),
!,
coindexed(Y, Z), \% Binding Principle C: globally o-free npro.
binding_principles(HeadDtr, Rest).

Figure 2.20: Definite Clauses for Binding Principles

The Binding Theory is based on the distinctions of different subtypes of nominal-object. nom-obj has subtypes pron (pronominal) andupro (nonpronominical). pron has subtypes ana (anaphoric) and ppro (nonanaphoric personal-pronominal). The definite clauses hpSG_pron, hpSG_ana, hpSG_ppro and hpSG_npro simply try each of the possible subtypes.

The definitions of "o-commands" (oblique-commands) and "locally o-commands" in HPSG binding theory approximate to the "o-commands" of Chomsky's binding theory, but the HPSG theory is based on the relative obliqueness of two items on a given subcategorization list. The definite clauses for o_commands and locally_o_commands, which take the two items and the list as parameters, are shown in Figure 2.21.
locally_o_commands(_, _, tl: []) if !, fail.
locally_o_commands(Y, Z, Subcat) if referential_synsem_object(Y), referential_synsem_object(Z), less_oblique(Y, Z, Subcat).
locally_o_commands(Y, Z, Subcat) if referential_synsem_object(Y), referential_synsem_object(Z), member(X, Subcat), subcategorizes_for(X, Z), locally_o_commands(Y, X, Subcat).

Figure 2.21: Definite Clauses for o-Commands

When applied to parsing, the effect of the binding theory is to require the identity (coreference) or non-identity of the referential indices of the nominal objects involved, based on purely syntactic (non-contextual) criteria, in those cases where this is possible. The clauses coindexed and contraindexed in Figure 2.20 use ALE constraints to impose permanent identity or permanent non-unifiability of the variables respectively.

We illustrate the working of the implemented binding theory by showing the results of parsing the sentences the man he saw saw him and the man he saw saw himself in Figures 2.22 and 2.23. In the CONT (CONTENT) feature in Figure 2.22, the SEER (tag [0]) and the SEEN (tag [7]) have different indices. The binding theory implemented in Figure 2.20 has caused these referents to be contraindexed by the contraindexed clause. The permanent non-unifiability of these indices is shown by the explicit inequality [7] != [0] which ALE prints out as part of the result.

In the CONT (CONTENT) feature in Figure 2.23, by contrast, the SEER and the SEEN are coindexed, both having the tag [0] The difference comes from the lexical entries of the pronouns. Personal pronoun him has content of subtype ppro, and reflexive pronoun himself has content of subtype refl (reflexive) which is a subtype of ana. Both are locally o-commanded by the subject (the man he saw), so the clauses of Figure 2.20 cause personal pronoun him to be contraindexed, and reflexive himself to be coindexed, with the subject.
| ?- rec [the, man, he, saw, saw, him].

Figure 2.22: Parsing the man he saw saw him
The text in the image is a representation of a parsing task using a formal grammar or parsing system. It describes the syntactic structure of the sentence "the man he saw saw himself" and includes details about the constituents, their categories, and relationships. The figure at the bottom indicates that the parsing was performed using a specific system, and it highlights the sentence's structure with indices and labels for different parts of the sentence.
Chapter 3

Generation with DCG and SGX: An English Engine

After describing an implementation in Chapter 2 in which the grammar conforms closely to HPSG theory but which can only be used for parsing, in this chapter we describe a framework which can do generation successfully but in which a DCG grammar approximates to HPSG theory to only a limited extent. In Chapter 4 we describe a revision of this framework which implements HPSG theory more closely.

In this chapter we discuss head-driven generation algorithms, and why head-driven generation is a natural approach to generation with HPSG. We introduce the SAX parser and the SGX head-driven generator, which we will use throughout Chapters 3, 4 and 5. We then describe a natural language engine called English Engine, which uses SAX and SGX for parsing and generation with a DCG grammar which includes many of the features and mechanisms of HPSG. Finally we mention an adaptation of English Engine as a natural language dialogue interface, which raises issues about using HPSG in a dialogue framework which we will discuss in Chapter 7.

3.1 Head-Driven Generation Algorithms

Because HPSG is a head-driven grammar, it seems more natural to use head-driven generation than any other approach. How to do head-driven generation with HPSG is therefore one of the central themes of the thesis, starting in this chapter, continuing in Chapter 4 and concluding in Chapter 6. Here, we briefly review the family of head-driven generation algorithms.

An overview of head-driven generation is given by van Noord (1990). Basically, head-driven generation is better than either naive top-down generation or naive bottom-up generation, because head-driven generation gives a way to combine top-down and bottom-up processing advantageously. Top-down generation tends to suffer from problems of non-termination, caused by recursion in grammar rules. Restricting the rules to avoid recursion is unattractive from a linguistic point of view. Bottom-up generation avoids these non-termination problems, but tends to suffer from problems of semantic compositionality and problems of efficiency. To avoid semantic compo-
tion problems with his generator, Shieber (1988) had to restrict the grammars to be semantically monotonic. Efficiency problems arise from left-to-right processing, which is natural for parsing but inappropriate for generation, causing significant nondeterminism. Head-driven generation is designed to avoid all of these problems.

3.1.1 Head-Driven Bottom-Up Generation

Basically, head-driven generation exploits identity of logical form between mothers and daughters in grammar rules. The simplest form is demonstrated by van Noord (1990). His Head-Driven Bottom-Up Generation (BUG) algorithm, shown in Figure 3.1, requires every grammar rule (except rules which simply introduce a lexical item) to have a daughter whose logical form is identical to the logical form of the mother. This daughter is defined to be the semantic head daughter.

```prolog
?-op(500, xfx, '--->').
bug1(Node) :-
    predict_word(Node, Small),
    connect(Small, Node).
connect(Node, Node).
connect(Small, Big) :-
    predict_rule(Small, Middle, Others, Big),
    gen_ds(Others),
    connect(Middle, Big).

gen_ds([]).
gen_ds([Node|Nodes]) :-
    bug1(Node),
    gen_ds(Nodes).
predict_word(node(,_/LF, _), node(S/LF, F)) :-
    ( node(S/LF, F) --&gt; [] ).
predict_rule(Head, Mother, Others, _) :-
    ( Mother --&gt; [Head|Others] ).
```

Figure 3.1: The BUG Algorithm (van Noord, 1990)

When generation starts from a given logical form, the BUG algorithm immediately finds a lexical entry whose logical form can be unified with the given logical form. This lexical entry is the initial "pivot" for the generation process. The algorithm then finds a grammar rule with this logical form in the mother and the semantic head daughter (the pivot), and generates the other daughters in the rule recursively. After generating the sisters of the pivot, the algorithm moves up to the mother and takes the mother as the next pivot, finds a matching rule and generates the sisters of the mother, and works upwards until the generation terminates.

An example grammar for a fragment of English, slightly modified from (van Noord, 1990), is shown in Figure 3.2. With this grammar fragment and the Prolog algorithm in Figure 3.1, strings such as mary calls the boy up can be generated from logical forms such as call_up(mary, boy). As BUG is the simplest head-driven generation algorithm, we will use it with an HPSG grammar in Chapter 6, to clarify fundamental problems in head-driven generation with HPSG.
% Semantic head first in daughters list (e.g., s --- [vp, np].)
% Surface order given by difference list (van Noord, footnote 3).

node(s/LF, PC-PN) --->  
  [ node(vp(Np, □)/LF, P1-PN),  % van Noord: s ---> [np, vp].
    node(Np, P0-P1) ].

node(vp(Subj, T)/LF, PC-PN) --->  
  [ node(vp(Subj, [Obj|T])/LF, P0-P1),
    node(Obj, P1-PN) ].

node(np/LF, PC-PN) --->  
  [ node(n/LF, P1-PN),  % van Noord: np ---> [det, n].
    node(det/the, P0-P1) ].  % van Noord: det/LF

cnode(np/john, [john|X]-X) ---> □.
ncnode(np/marry, [mary|X]-X) ---> □.
cnode(n/boy, [boy|X]-X) ---> □.
cnode(n/girl, [girl|X]-X) ---> □.
cnode(det/the, [the|X]-X) ---> □.  % van Noord: det/_

ncnode(vp(np/Np, □)/sleep(Np), [sleeps|X]-X) ---> □.
ncnode(vp(np/Np, [np/Np2])/kiss(Np, Np2), [kisses|X]-X) ---> □.
nncnode(vp(np/Np, [np/Np2, p/up])/call_up(Np, Np2), [calls|X]-X) ---> □.
nncnode(p/up, [up|X]-X) ---> □.

Figure 3.2: Grammar for a Fragment of English

3.1.2 Semantic Head-Driven Generation

The BUG algorithm imposes a severe restriction on grammars by requiring every rule to have a semantic head daughter whose logical form is identical with the logical form of the mother. This restriction is avoided by the Semantic Head-Driven (SHD) algorithm of Shieber et al. (1989; 1990).

Grammar rules are divided into two types, chain rules and non-chain rules. Chain rules are rules which do have a semantic head daughter whose logical form is identical with the logical form of the mother. Non-chain rules are rules which don’t. Rules which merely introduce a lexical item are non-chain rules, but SHD also allows other types of non-chain rules. Basically, the SHD algorithm uses chain rules bottom-up, and non-chain rules top-down. If all rules in a given grammar have a semantic head (that is, there are no non-chain rules except for lexical items), SHD finds a lexical pivot and works bottom-up exactly like BUG. At the other extreme, if no rules have a semantic head the algorithm cannot be head-driven but works just like a top-down generator. If the grammar has both chain and non-chain rules, the algorithm finds the lowest possible pivot (which will not necessarily be lexical) and generates with a mixture of bottom-up and top-down processing.

In the case of HPSG, it is particularly natural to use a head-driven generation algorithm. “Head-driven Phrase Structure Grammar” is head-driven not only syntactically, but also as far as possible semantically. HPSG’s Head Feature Principle requires the identity of major syntactic features between a phrase and its syntactic head daughter, and HPSG’s Semantics Principle requires the
identity of major semantic features between a phrase and its semantic head daughter. Semantic head-driven generation should therefore be easy to implement.

In the case of HPSG, semantic head-driven generation should also be efficient, because it should basically reduce to the bottom-up BUG algorithm. Head-driven bottom-up generation is geared both to the input logical form (head-driven) and to the information in the lexicon (bottom-up). HPSG is ideally suited to this, because it is highly lexicalist, with rich information in the lexicon, and also because it has a clear definition of semantic head: in head-adjunct phrases, the adjunct daughter is the semantic head; in other headed phrases, the syntactic head daughter is the semantic head. In both cases, the Semantics Principle basically requires the content of the semantic head to be identical to the content of the mother.

If the Semantics Principle is applied strictly, the result is that all HPSG grammar rules are chain rules in the sense of the SHD algorithm. In other words, in the case of HPSG, SHD generation should reduce to BUG generation. We will investigate this issue thoroughly in Chapter 6. At the moment, we are simply pointing out that semantic head-driven generation is the most natural approach to generation with HPSG.

3.1.3 Chart-based Semantic Head-Driven Generation

Even in the original papers on semantic head-driven generation, Shieber et al. (1989; 1990) pointed out that the basic SHD algorithm, although more efficient than non-head-driven algorithms, could give rise to a great deal of unnecessary nondeterminism. They pointed out two particular sources of nondeterminism. One of them is caused by premature lexical choice of specific lexical forms, before enough features have been specified to determine the final form. They suggest this problem can be solved by using a lexicon of word stems during syntactic generation, and delaying the choice of final word form to a post-process. We will investigate this approach in Chapter 4, where we show that in the case of HPSG grammars we can exploit the levels of underspecification and the subsumption ordering in the type hierarchy, to achieve a high degree of determinism in lexical choice.

The other major source of inefficiency pointed out by Shieber et al. (1989; 1990) was the backtracking and recomputation of partial results in the simple form of the SHD algorithm which they presented. They suggested that this could be avoided by using a chart, as in chart parsing.

This suggestion was taken up by Haruno et al. in their Bidirectional Chart Generation algorithm (1993) or Chart-based Semantic Head-Driven generation algorithm (1996). They present a parallel implementation, based on the techniques of the PAX parallel chart parser (Matsumoto, 1986). However, we will use sequential implementations: the SAX parser and the SGX generator.

3.2 SAX and SGX

Although we are concerned with natural language generation rather than parsing, we introduce the SAX parser before describing the SGX generator, because the chart implementation for SGX was derived directly from the implementation technique already used for SAX. The origins of the SAX and SGX systems are shown schematically in Figure 3.3. We also describe SAX because it is used for parsing in the systems in Chapters 3, 4 and 5. The systems in Chapters 3 and 4 are
bidirectional, using SAX for parsing and SGX for generation. The more “theoretical” implementation in Chapter 5 works only for parsing and not for generation, due to the theoretical problems analyzed in Chapter 6.

Figure 3.3: Origins of SAX and SGX

3.2.1 The SAX Chart Parser

After developing the well-known BUP parser (Matsumoto et al., 1983), Matsumoto went on to develop a parallel parsing algorithm PAX (Matsumoto, 1986) for parallel hardware at the Japanese Fifth Generation Computer System project. The SAX parser (Matsumoto and Sugimura, 1987) is an implementation, for ordinary sequential hardware, of the PAX parallel algorithm. SAX is implemented in SICStus Prolog (SICS, 1995) and uses SICStus’s co-routining facilities to implement concurrent processing which approximates to the parallel processing of PAX. The SAX parsing system has since been further developed to include morphological analysers for Japanese and for English, feature structure unification, graphical debugging, and other facilities (Matsumoto et al., 1994).

Like the earlier BUP parser, SAX uses partial execution to compile the grammar for efficient bottom-up parsing. This is done by the SAX Translator program in a separate off-line preprocessing stage, before the parsing process starts, in a similar way to the BUP system. During the parsing process however, instead of building the chart by asserting edges into the Prolog database, SAX implements the chart by creating concurrent processes. For compiled rather than interpreted
Prolog systems, this is a highly efficient form of chart parsing, even on sequential machines. The terminal and non-terminal symbols of the grammar are realized as processes which communicate via streams to build larger structures. A meta-process monitors the streams and controls the whole parsing process (Matsumoto et al., 1993).

**Grammar Restrictions**

Basically, SAX accepts grammars in the form of definite clause grammars (Pereira and Warren, 1980). However, for the sake of efficiency in parsing, some restrictions are imposed. In order to avoid a great deal of copying of variables, a high level of determinism is required within the concurrent processing framework. Therefore, although Prolog code can be added as procedural attachments to DCG rules ("extra conditions") in the usual way, SAX requires all extra conditions to be deterministic. In practice, this means that if an extra condition is not deterministic, the solution which happens to be found first will be taken as the only solution.

Another restriction is that SAX does not permit empty categories in the grammar rules. Since the 1994 textbook version of HPSG uses empty categories, this seems to be a major problem for parsing HPSG with SAX. However, we will see in Chapter 5 that this problem disappears in later versions of HPSG, which do not use empty categories at all. The restriction of SAX, not supporting empty categories, can therefore be considered to be an advantage allowing faster parsing.

If empty categories are really necessary, they can be handled in the SAX concurrent processing system via a meta-process. The general approach is described by Matsumoto (1991) for handling co-ordination, and by Imaichi and Matsumoto (1995) for handling ill-formed inputs. It seems very appropriate that if something is analyzed by positing the existence of an inaudible word, a special mechanism for handling something difficult should be used.

As well as grammars in the form of DCGs with ordinary Prolog terms, SAX can accept grammars which include feature structures. In this case, the SAX UG (Unification Grammar) module must be included, and unification is done by a separate feature structure unification procedure. However, the SAX UG module was not designed to be shared with the SGX generator, and did not support typed feature structures, so I did not use the UG module at all. The grammar described in this chapter is a DCG using ordinary Prolog terms. The grammar in Chapter 4 uses ProFIT to compile HPSG typed feature structures into ordinary Prolog terms, so that when the grammar is input to the SAX Translator, it is in effect a normal DCG.

**Morphological Analysis**

As usual in DCG grammars, lexical items can be introduced by specific rules included in the grammar. Alternatively SAX can be used with CHASEN, a powerful morphological analyzer for Japanese, or with the SAX morph module, a simple morphological analyzer for English. We will only discuss the English one. In this case, lexical items are specified in a dictionary file, separately from the grammar rules.

The morphological analysis takes place in a separate preprocess, before the SAX parser starts syntactic analysis. Because there may be many alternative possible combinations of morphemes, the SAX morph module constructs a lattice of possible morphemes, and the SAX parser uses the
lattice as the starting point for parsing. Robustness can be greatly improved by using the morphological preprocess, because unknown words or misspelt words can be identified in morphological analysis, and can be accepted for syntactic parsing (for example, with some default dictionary information).

3.2.2 The SGX Head-Driven Generator

The SGX generator uses the Chart-based Semantic Head-Driven (CSHD) generation algorithm of Haruno et al. (1993; 1996). The CSHD algorithm follows the Semantic Head-Driven (SHD) generation algorithm of Shieber et al. (1989; 1990) to ensure efficient ordering of the generation process. However, CSHD also implements the suggestion of Shieber et al. that backtracking and recomputation of partial results should be avoided by using a chart, as in chart parsing.

The published CSHD algorithm of Haruno et al. (1996), called Bidirectional Chart Generation algorithm by Haruno et al. (1993), is a parallel implementation in GHC, using the same chart techniques as the PAX parallel chart parser. The SGX generator for sequential machines is a concurrent SICStus Prolog (SICS, 1995) implementation of the parallel CSHD algorithm. Like the SAX parser, SGX implements the chart by concurrent processes and communication streams monitored by a meta-process. Also like the SAX parser, SGX uses partial execution to compile the grammar for efficient generation. This is done by the SGX Translator in a separate off-line preprocessing stage, before the generation process starts. In this stage, SGX also compiles tables of chain rules.

Grammar Restrictions

SGX requires grammars to be in Semantic Head Grammar (SHG) format. SHG format is basically DCG format plus a logical form, separated from the rest of the category by "/\". This is basically the format used by Shieber et al. (1990) to present the SHD semantic head-driven generation algorithm. In addition, SGX requires the semantic head daughter of each chain rule to be explicitly marked by "/\#". This enables the SGX Translator to compile tables of chain rules (it is possible that other daughters in the same rule may also have the same logical form as the mother, but only one of the daughters must be marked as the semantic head).

Like SAX, SGX imposes restrictions for the sake of efficiency. SGX does not permit empty categories in the grammar rules. SGX also does not permit the logical form of the mother of a non-chain rule to be an uninstantiated variable. The SAX UG (Unification Grammar) module for feature structures is not designed for SGX.

SGX requires all extra conditions to be deterministic, in order to avoid copying of variables. In practice, this means that if an extra condition is not deterministic, the solution which happens to be found first will be taken as the only solution. This restriction causes considerable difficulties if lexicon access is performed by an extra condition (which is normal in systems using DCGs), because such lexical access will find only one lexical item. In the case of the SAX parser this problem can be avoided by using the morphological preprocess, which can find all alternative lexical items before parsing starts. There is no equivalent lexical preprocess for the SGX generator, so SGX requires lexical items to be specified by rules included in the DCG grammar. Either every word needs its
own specific rule in the grammar, or a general rule needs to perform lexicon access by an extra condition, which must then be deterministic. This problem, and its partial solution by means of delayed lexical choice, will be investigated in Chapter 4.

**Generalized Chart Algorithm**

When a chart parser begins processing, the chart is initialised using the input words. In its simplest form there is a single sequence of words which form the initial edges in the chart, and the nodes in the chart connected by the initial edges are simply numbered consecutively using the given order of the input words. In the case of the SAX morphological preprocess, several alternative possible sequences of morphemes are linked up in a lattice, instead of a single sequence of words. In the case of generation, however, the sequence of words which will be produced (or even a possible lattice of alternative combinations of morphemes) is not known until the end of processing, so the chart cannot be initialised by this method.

The CSID algorithm used by SGX therefore generalizes the chart algorithm, so that basic nodes in the chart (not only the edges between them) can be introduced incrementally, and instead of a single fixed order of nodes, they can be connected in a many-to-many fashion. This generalization of the chart algorithm was taken further by Den (1994), who showed that it could also be used as the basis for proof procedures for cost-based abduction.

**3.2.3 Combining SAX and SGX**

In order to load the SAX and SGX software together into a single application system, I made some small modifications to use Sicstus Prolog module definitions. With this method it is also possible to compile two object grammars, one for parsing by SAX and one for generation by SGX, and load both grammars together in the same system. Although loaded in the same system, the object grammars are kept distinct by the module definitions.

The object grammar for parsing by SAX must be compiled by the SAX Translator from a source grammar in the form of a DCG. The object grammar for generation by SGX must be compiled by the SGX Translator from a source grammar in the form of an SHG (Semantic Head Grammar). In order to avoid the redundancy of maintaining two grammars of English in closely-related formats, I wrote a procedure which translates SHG grammars into DCG grammars, shown in Figure 3.4.

The basic step is to take the logical form, which is specified after `"/"` in SHG format, and move it into the first argument position inside the parentheses in the DCG format. This procedure is called by the SAX Translator, which expects a user-defined procedure for such purposes. After the SHG grammar has been converted to DCG format, SAX Translator compiles it for parsing. The SGX Translator accepts the grammar directly in SHG format and compiles it for generation. So it is possible, if desired, to develop a single source grammar (in SHG format), which can be compiled separately by SAX for parsing and by SGX for generation.

Although it is desirable to be able to develop a single grammar for use in both parsing and generation, sometimes it is convenient to specify differences between parsing and generation. I implemented this in the SAX translation procedure in Figure 3.4, which accepts any rule which is prefixed by the label `sax` and ignores any rule which is prefixed by the label `sgx`. I made the
% sax_term_expansion(+SHG_Rule, -DCG_Rule)

sax_term_expansion((:- shg_top_node_category(_, Func, Arity)),
                (:- sax_trans:shg_top_node_category(_, Func/NewArity)) :- !,
                NewArity is Arity + 1.

sax_term_expansion((sax Head --> Body), (NewHead --> NewBody)) :- !,
    shg_dcg(Head, NewHead),
    shg_dcg_body(Body, NewBody).

sax_term_expansion((sgx _Head --> _Body), []) :- !.

sax_term_expansion((Head --> Body & Dx), (NewHead --> NewBody & Dx)) :- !,
    shg_dcg(Head, NewHead),
    shg_dcg_body(Body, NewBody).

sax_term_expansion((Head --> Body), (NewHead --> NewBody)) :- !,
    shg_dcg(Head, NewHead),
    shg_dcg_body(Body, NewBody).

shg_dcg(Cat/Sem, NewCat) :-
    Cat =.. [Func|Args],
    NewCat =.. [Func, Sem|Args].

shg_dcg_body((#Cat/Sem, Body), (NewCat, NewBody)) :- !,
    shg_dcg(Cat/Sem, NewCat),
    shg_dcg_body(Body, NewBody).

shg_dcg_body((Cat/Sem, Body), (NewCat, NewBody)) :- !,
    shg_dcg(Cat/Sem, NewCat),
    shg_dcg_body(Body, NewBody).

shg_dcg_body([Extra], Body), ([Extra], NewBody)) :- !,
    shg_dcg_body(Body, NewBody).

shg_dcg_body([Word], Body), ([Word], NewBody)) :- !,
    shg_dcg_body(Body, NewBody).

shg_dcg_body(#Cat/Sem, NewCat) :- !,
    shg_dcg(Cat/Sem, NewCat).

shg_dcg_body(Cat/Sem, NewCat) :- !,
    shg_dcg(Cat/Sem, NewCat).

shg_dcg_body(Extra, Extra) :- !.

shg_dcg_body([Word], [Word]) :- !.

Figure 3.4: Procedure to translate SHG to DCG format.

equivalent SGX translation procedure accept rules labelled sgx and ignore rules labelled sax. Both procedures accept all the shared rules which are not specifically labelled.

3.3 An English Engine (DCG Version)

We now describe English Engine, a system which uses SAX and SGX. English Engine was developed as a free-standing general-purpose natural language engine, like the Core Language Engine (CLE) (Alshawi, 1992). Its functions are orthodox for such an NLE: to parse English keyboard inputs to produce logical forms, and to generate English outputs from logical forms.

3.3.1 Comparison with Core Language Engine

The general design of English Engine was inspired by the design of CLE, perhaps the most successful general-purpose natural language engine implemented in Prolog. This general design, shown in Figure 3.5, is based on a unification-based phrase structure grammar, a quasi-logical form for
semantic representation, a bottom-up chart parser, and a semantic head-driven generator.

English Engine follows the same general design as CLE, but uses newer, equivalent components. The CLE parser was similar to the BUP parser (Matsumoto et al., 1983). English Engine uses the later SAX parser (Matsumoto et al., 1994), described in Section 3.2.1, which combines the BUP techniques with a parallel processing approach. In CLE, generation was based on the semantic head-driven (SHD) algorithm (Shieber et al., 1989). English Engine uses the SGX generator described in Section 3.2.2, based on the CSHD algorithm (Haruno et al., 1996) which combines SHD generation with a chart to eliminate backtracking. The CLE grammar was influenced primarily by GPSG (Gazdar et al., 1985). In English Engine the grammar is influenced primarily by HPSG (Pollard and Sag, 1994). In the version of English Engine described in this chapter, the grammar is written directly as a Prolog DCG, but generally follows HPSG in its choice of features and its basic organizing principles.

The basic differences are shown by comparing Figures 3.5 and 3.6. The CLE grammar was written in a convenient format, which was compiled into Prolog by the CLE Grammar Compiler included in Figure 3.5. Although Figure 3.6 does not show a separate grammar compiler component, the Prolog DCG is compiled separately for parsing by the SAX Translator and for generation by the SGX Translator. In Chapter 4, the ProFIT grammar compiler will be added to the design of Figure 3.6, in the place occupied by the CLE grammar compiler in Figure 3.5, and then the similarity with the CLE design will be clear.

Following GPSG, CLE’s semantic theory was based on Montague semantics, including the lambda calculus. However, the Prolog implementation of CLE was based fully on unification,
and CLE demonstrated that unification-based semantic processing could largely eliminate the use of lambda expressions (Moore, 1989). This had the advantage of enabling the grammar to be used reversibly, for generation as well as parsing.

CLE’s semantic representation differed further from standard Montague semantics by including Davidsonian events, enabling an operator-free first-order representation of adverbial modification. English Engine uses a “neo-Davidsonian” (Parsons, 1990) semantic representation, described in Section 3.3.3, although HPSG generally assumes a semantic theory based on Situation Semantics. The combination of neo-Davidsonian semantics with HPSG was used in the PLUS project (Rentier, 1994) and in Sharp Corporation’s shake and bake machine translation system (Sanfilippo et al., 1994), which both influenced the work described here. The subsequent development of Minimal Recursion Semantics by Copestake et al. (1995; 1997) made it more orthodox to use a neo-Davidsonian semantics with HPSG.

### 3.3.2 Semantic Head-Driven DCG Grammar

The grammar for English Engine is written in SHG format and is compiled into separate object grammars for parsing and generation as described in Section 3.2.3. Some example phrase structure rules in SHG format are given in Figure 3.7. The rules use the normal DCG arrow notation (→) between mother and daughters. As these rules are for verb phrases (VPs), the mother’s category functor name is vp. The semantic head daughter in all these examples is a verb, whose category functor is v. The various complement daughters have their own category functors (np, ap, pp). Because the verb is the semantic head daughter, it is marked by “#” as required by SGX.

After the vp category functor, the first argument inside the parentheses specifies the head Features in the form hf:BF, where BF is a variable which unifies the head features of the mother with the head features of the head daughter, as required by the HPSG Head Feature Principle. The format hf:BF is used in the style of an attribute:value pair, but in fact it is simply a Prolog term in this implementation. The second argument specifies the subcategorization list in the form sc:SC, where SC is a variable, or sc:[…] with an actual list of subcategorized categories. The third argument specifies the Slash feature in the form slash:Slash or slash:[]. After the three arguments inside the parentheses, the logical form is specified in the form /LF where LF is a variable, or /….&…&…. where semantic terms are separated by ampersands.

### 3.3.3 Event-based Logical Form

The logical form used in English Engine is a simple event-based representation known as ELF (Event-based Logical Form), based on the “neo-Davidsonian” semantic theory of Parsons (1990). In ELF, verbs have event indexes, NPs fill thematic roles, PPs can be attached to event indexes, and adjectives have “state” indexes. The choice of this general style of semantics was partly motivated by its use with HPSG at Sharp Labs of Europe (Sanfilippo et al., 1994), but the specific details of ELF in English Engine were developed independently.

In ELF, the logical form starts with a group of 3 contextual indices, which appear in the example VP rules as variables in the form Cs&Ca&Ct. These represent the 3 standard HPSG contextual indices for speaker (Cs), addressee (Ca) and utterance-time (Ct). In this implementation, it is
% Intransitive, or slash complement
vp(hf:HF, sc:SC, slash:Slash)
    /LF
--> # v(hf:HF, sc:SC, slash:Slash)
     /LF.

% NP subject, NP object
vp(hf:HF, sc:[np(Case)/index(A,AgrA) & TermA], slash:[])
    /Cs&Ca&Ct & PredE & tense(E,T)
    & RoleA & index(A,AgrA) & TermA
    & RoleB & index(B,AgrB) & TermB
--> # v(hf:HF, sc:[np(Case)/index(A,AgrA) & TermA],
    np(acc)/index(B,AgrB) & TermB],
    slash:[])
    /Cs&Ca&Ct & PredE & tense(E,T)
    & RoleA & index(A,AgrA) & TermA
    & RoleB & index(B,AgrB) & TermB,
    np(acc, que:[])
    /Cs&Ca&Ct & index(B,AgrB) & TermB.

% NP subject, predicative AP complement
vp(hf:HF, sc:[np(Case)/index(A,AgrA) & TermA], slash:[])
    /Cs&Ca&Ct & state(E,Adj) & tense(E,T)
    & theme(E,A) & index(A,Agr) & TermA
--> # v(hf:HF, sc:[np(Case)/index(A,Agr) & TermA],
    ap(prd:y)/state(E,Adj) & tense(E,[])
    & theme(E,E) & index(A,Agr) & TermA],
    slash:[])
    /Cs&Ca&Ct & state(E,Adj) & tense(E,T)
    & theme(E,A) & index(A,Agr) & TermA,
    ap(hf:[prd:y, mod:none], sc:[np(ncm)/index(A,Agr) & TermA])
    /Cs&Ca&Ct & state(E,Adj) & tense(E,[])
    & theme(E,E) & index(A,Agr) & TermA.

Figure 3.7: Some VP phrase structure rules in SHG format

assumed that these contextual indices will remain constant throughout an utterance. A more
sophisticated representation of contextual indices in HPSG is discussed in Chapter 6.

After the 3 contextual indices, the ELF logical form specifies the main predicate term and its
tense. For example, the rule for a predicative adjective phrase (AP) in Figure 3.7 specifies by
state(E, Adj) & tense(E,T) that the main predicate is of type state, has the index variable E,
has a value Adj which will be instantiated by a specific adjective, and has a tense T.

After the main predicate and tense, the ELF logical form specifies a number of thematic roles.
Each thematic role is specified by a role name (agent, theme, etc.), an index term, and a further
term. For example, in the rule for a predicative AP in Figure 3.7 the thematic role is theme, and
it is specified by theme(E, A) & index(A,Agr) & TermA where TermA is not yet instantiated (it
will be unified with TermA in the semantics of the subject).

In ELF, NP indexes carry agreement features as in HPSG. The Prolog term for an NP index
looks like index(X, AgrX) where X is the referential index (which may be instantiated to a specific
discourse referent identifier) and AgrX is a variable for X’s agreement features (which may be instantiated to specific features such as 3-sg). As in HPSG, NP indexes are specified by verbs in their subcategorization list, for subject-verb agreement.

3.3.4 Lexicon and Lexical Rules

As English Engine uses the SAX morphological analyzer described in Section 3.2.1, the lexicon takes the form of a file of lexical items, separate from the grammar. An example lexical item is shown in Figure 3.8.

dict(be, v, [_Cs&_Ca&_Ct & state(E,Adj) & tense(E,[])
 & theme(E,A) & index(A,Agr) & TermA,
   hf:[bse,aux;y,inv:n],
   sc:[np(nom)/index(A,Agr) & TermA,
     ap(prd:y)/state(E,Adj) & tense(E,[])
     & theme(E,A) & index(A,Agr) & TermA],
   slash:[])).

Figure 3.8: Base form lexical entry for “be” with predicative AP complement

The entries in the lexicon have the functor dict with 3 arguments: the word’s form, the word’s category, and a list of further specifications within that category. When the word be is input, the SAX morphological analyzer finds the lexical entry in Figure 3.8 (among others), and inserts into the morpheme lattice a category with functor v and the 4 arguments from the list (the logical form, the head features, the subcategorization list, and the slash list). This category will successfully match the head daughter (marked with “#”) in the rule for VP with predicative AP complement in Figure 3.7. Note that the logical form, which appears after “/” in the VP rule, will be moved to the first argument position by the SAX Translator as described in Section 3.2.3.

The lexical entry in Figure 3.8 is the base form of the verb be, subcategorizing for a predicative adjective phrase. The verb’s head features are specified as a list in hf:[bse, aux:y, inv:n]. The features closely follow HPSG. The first item in the list is the verb’s VFORM feature, which has the value bse (base form). aux:y specifies that the AUX (auxiliary) feature has the value y, equivalent to “AUX +” in HPSG. inv:n specifies that the INV (inverted) feature has the value n, equivalent to “INV -” in HPSG.

Following HPSG, new lexical items can be derived from existing lexical items by lexical rules. Figure 3.9 shows a lexical rule to derive the 3rd person singular form is from the base form be. The lexical rule is actually a Prolog inference rule, which states in effect that if any base form be exists, with some complements specified in Comps, then a finite form is also exists, with the same complements, but its subject must be 3rd person singular.

Note that although the lexical rule in Figure 3.9 specifies a relationship between one lexical item and another (that one can be derived from the other), this style of rule is processed quite differently from the lexical rules in the ALE system in Chapter 2. In ALE, lexical rules are applied during lexicon compilation, to expand the lexicon to include all the derived items as well as the existing items. In English Engine, there is no expansion of the lexicon during compilation. Lexical rules
are compiled as Prolog inference rules, and are used during the SAX morphological preprocess to prove the existence of the derived items and add them to the morpheme lattice.

3.3.5 Robust Parsing and Generation

Although the same grammar is used for both parsing and generation, English Engine handles some details in different ways in order to support robust parsing to a limited extent, while maintaining correct generation. We illustrate this by the handling of the indefinite articles *a* and *an*, whose lexicon entries are shown in Figure 3.10.

```
dict(Det, det, [Quant, PhonAgr, SyntAgr]) :-
    det(Det, PhonAgr, SyntAgr)/Quant.

det(a, consonant, 3-sg)/indef.
det(an, vowel, 3-sg)/indef.
det(the, _PhonAgr, 3-_N)/def.
```

Figure 3.10: Lexicon entries for *a*, *an* and *the*

All determiners are given a phonetic agreement feature, which is left uninstantiated as variable _PhonAgr_ in most determiners, as can be seen in the case of the definite article *the* in Figure 3.10. For the indefinite article *a*, _PhonAgr_ is instantiated to the atomic value consonant, while in the case of *an*_PhonAgr_ is instantiated to the atomic value vowel_. This phonetic agreement feature is used in the phrase structure rules for noun phrases, some of which are shown in Figure 3.11.

The first rule in Figure 3.11 is labelled sax, so it is used only for parsing and not for generation, as described in Section 3.2.3. This rule accepts an NP consisting of an indefinite determiner followed by an Nbar (category n1), without requiring any phonetic agreement between them. The determiner’s first feature is a free variable _DetPhon_, which is not unified with anything and can take any value.

The second rule in Figure 3.11 is labelled sgx, so it is used only for generation and not for parsing. This rule generates an NP consisting of an indefinite determiner followed by an Nbar (category n1), and requires phonetic agreement between them. The determiner’s first feature is the variable _PhonInitial_. The Nbar’s first feature is _Phon_, which holds the initial part of the surface string value of the Nbar in the variable _Phon_. The rule has a procedural attachment which
calls the procedure phon_initial, giving the values of Phon from the Nbar and Phon_initial from the
determiner as parameters. This procedure enforces phonetic agreement of the determiner with
the Nbar by instantiating Phon_initial in the determiner to either consonant or vowel according
to the actual string value of Phon in the Nbar. The procedure includes both general rules about
vowels and consonants, and specific exceptions such as hour (vowel) and unified (consonant).

The third rule in Figure 3.11 ensures that when an Nbar is made from an Adjective Phrase
(category ap) followed by a Noun, the ph:Phon value of the Nbar (the initial part of the surface
string, which will control the phonetic agreement of the determiner in the NP) is that of the AP
rather than that of the Noun (as in NP *a red apple not *an red apple). The fourth rule handles
the simple Nbar without AP (in NP an apple not *a apple). The first two rules, labelled sax and
sgx respectively, ensure that both the starred and unstarred forms will be accepted by SAX, but
only the unstarred forms will be generated by SGX.

3.3.6 A Dialogue Interface

We now describe an adaptation of English Engine for use as a natural language interface to PLUS,
a Pragmatics-based Language Understanding System (Black et al., 1993). We will not discuss
internals of PLUS, as the only relevant part is the interface between the Natural Language Engine
(NLE) and the Dialogue Manager (DM). Although HPSG was the preferred grammar formalism in PLUS, it was used only for parsing and not for generation. Instead of HPSG, the PLUS surface generator (Black and Cunningham, 1992) used categorial grammar. One reason was that the generator, implemented at UMIST, was based on an existing generator developed at UMIST by Phillips (1991) which used categorial grammar. For PLUS, the reliability of re-using a known approach outweighed the redundancy of having two different grammars for parsing and generation. In Chapter 7 we will show that Phillips' generation algorithm can in fact be used with HPSG.

![Diagram](https://example.com/diagram.png)

**Figure 3.12:** English Engine and Dialogue Manager.

English Engine was developed as a free-standing NLE, and was then adapted to serve as an NLE interface to the PLUS DM, as an alternative to the original PLUS NLE. This required major extensions to the grammar, to handle pragmatic, elliptical and fragmentary dialogue phenomena which were not treated at all in the initial, more theoretical HPSG-like grammar. It also required translation between language-oriented logical forms in the NLE and application-oriented logical forms in the DM. The basic arrangement is shown in Figure 3.12.

In order to handle the requirements of the dialogue system, the grammar had to be extended to cover many practical problems for which solutions have not yet been proposed in HPSG theory. These included pragmatic particles such as "Yes" and "Thanks", the use of numbers, and elliptical fragments. Figure 3.13 shows an example of the type of dialogue handled in PLUS. The pragmatics-based processing of such dialogues is described by Jokinen (1994). All the dialogue utterances in Figure 3.13 can be parsed and generated by the adapted English Engine.

HPSG syntactic theory deals mainly with single complete grammatical sentences. For dialogues, the grammar must handle both smaller units (phrases and fragments) and larger units (combinations of phrases or sentences). The adaptation therefore extended the grammar beyond HPSG by defining grammatical units (GUs) and discourse units (DUs). GUs include phrases ("in Bolton") and elliptical fragments ("to rent"). DUs combine GUs with punctuation marks ("To rent?") and pragmatic features (mood = y/n question). This approach was further extended by defining conjunctions of GUs, for example "To buy or to rent?" (mood = wh-question). Another fundamental problem is how to represent elliptical fragments in logical form. As an ad hoc solution, ELF logical form was extended to include specific terms for zero anaphora.

A basic problem in any natural language generation system is how to make lexical choices in a specific natural language. The ELF logical form uses English words as logical predicates, e.g. "company(C)", but the PLUS database uses domain-specific concepts, e.g. "carHireCompany(C)".

51
I need a car.
To buy or to rent?
To rent.
Where?
In Entwistle.
Where is Entwistle?
In Bolton.
There are 12 carhire companies. Would you like to see the list?
Yes.

Figure 3.13: Example of a PLUS dialogue.

In order to use English Engine for generation with the PLUS system, the database concepts must be translated to English words. The database syntax, e.g. "carHireEvent(E,A,B,C)" must also be translated to ELF syntax, e.g. "event(E,renting(E)) & agent(E,A) & theme(E,B)". The PLUS project used a conceptual lexicon to translate concepts to words, and the adaptation of English Engine also involved implementing a conceptual lexicon to translate between the concepts used in PLUS and ELF logical forms, but we will not discuss this further.

3.3.7 DCG and HPSG

The grammar in English Engine is basically a DCG, written in the SHG variant of the DCG format, but the specific features and mechanisms in the grammar follow HPSG where possible. In particular, head features are passed between mother and head daughter by unification, and grammar rules follow the subcategorization principle, so the grammar is certainly head-driven as well as being a phrase structure grammar. The grammar is not only syntactically head-driven by the percolation of head features, it is also explicitly semantically head-driven by the sharing of logical forms between mothers and semantic head daughters. Many other HPSG mechanisms are included, for example unbounded dependencies are implemented using the flash feature.

The adaptation of English Engine as an interface for PLUS was intended to show that a grammar based at least partly on HPSG could be used in a practical dialogue interface, for generation as well as parsing. In fact the result was that the grammar was extended in various ad hoc ways to meet the requirements of the dialogues, using the typical DCG approach of adding further specific rules rather than the HPSG approach of capturing generalizations within the theory. We will return to some of the issues in using HPSG in a pragmatics-based dialogue framework in Chapter 7.

While the ALE implementation in Chapter 2 could only do parsing, the English Engine described in this chapter succeeds in doing both parsing and generation with an "HPSG-like" grammar. However, the use of atomic categories such as vp and v in the DCG style means that there are many specific phrase structure rules, instead of a few general schemata as in HPSG. Although these grammars can be processed efficiently by SAX and SGX, they are difficult to develop and to debug. The absence of a type hierarchy and an inheritance mechanism means that many aspects
of HPSG theory cannot be implemented at all in such a DCG-based approach. In Chapter 4, these problems will be solved by adding a typed feature structure system to the SAX/SGX framework described here.
Chapter 4

Generation with HPSG and ProFIT: Delayed Lexical Choice

Chapter 3 described a natural language engine which performs efficient parsing and generation using SAX and SGX. However, the DCG grammar (in SHG format) approximates to HPSG theory only to a limited extent. It is not an HPSG grammar because, unlike the grammar in Chapter 2, it is not based on a typed feature structure representation, which is essential for HPSG.

In this chapter we describe a completely revised version of English Engine, which again performs efficient parsing and generation using SAX and SGX, but combines them with an HPSG grammar based on a typed feature structure representation, implemented using the ProFIT typed feature system. Where required to make the difference clear, the old version of English Engine described in Chapter 3 will be called DCG English Engine, and the new version described in this chapter will be called HPSG English Engine. We begin by briefly describing the ProFIT system.

4.1 The ProFIT Typed Feature System

ProFIT stands for “Prolog with Features, Inheritance and Templates”. The ProFIT system, which was developed by Erbach (1995), is an extension of Prolog which supports inheritance-based typed feature structures. The type hierarchy is declared in a signature, which defines the subtypes and appropriate features of every type. Terms which include typed feature structures can then be used in Prolog procedures alongside normal Prolog terms. In a separate preliminary compilation, the ProFIT system compiles the typed feature structures into normal Prolog terms, using the given signature declarations. The resulting normal Prolog terms can then be compiled by the normal Prolog system, which in the present case is SICStus Prolog. Like ALE (Chapter 2), and also SAX and SGX (Chapter 3), ProFIT itself is implemented in Prolog. The last released version was ProFIT 1.54 for SICStus Prolog 2.9. I made a trivial modification to ProFIT in order to run it with SICStus 3.5 (SICS, 1995).

Once the typed feature structures have been compiled into normal Prolog terms, these terms can be used with any software which accepts Prolog terms. This includes any parser or generator.
which accepts grammars in Prolog or in Prolog DCG formats. ProFIT can therefore easily be used with existing parsers or generators, and does not provide a built-in parser or generator of its own. This is a different approach from the ALE system described in Chapter 2, which provides not only a built-in parser but also built-in mechanisms for lexical entries, lexical rules, and so on. By contrast, ProFIT provides no mechanisms for such things, but leaves the user free to use any preferred format and any mode of processing (which the user must implement).

In HPSG English Engine, ProFIT is used in order to implement an HPSG grammar, and in order to combine the HPSG grammar with the SAX parser and the SGX generator. However, apart from being easy to use with existing parsers or generators, ProFIT can be used with the general Prolog system, to include typed feature structures in any kind of procedures the user wishes to develop. For example, in HPSG English Engine ProFIT is used to implement HPSG lexical rules in the form of Prolog procedures, which are used at run-time during lexicon lookup by the SAX morphological preprocessor. This is quite different from the ALE built-in mechanism for lexical rules, which are applied off-line during lexicon compilation.

ProFIT uses some of the same compilation techniques as the Core Language Engine (CLE) grammar compiler (Alshawi, 1992), or revised versions of the same techniques. Typed feature structures are compiled by ProFIT into Prolog terms, in the same way that CLE compiles feature structures into terms, so that relatively slow unification of feature structures is replaced by relatively fast unification of terms. Also like CLE, ProFIT uses the technique of Mellish (1988) for compiling finite domains such as index agreement into boolean vectors for fast unification. However, while CLE used a sort hierarchy only for semantic selectional restrictions, HPSG uses a sort hierarchy also for syntactic restrictions, which means that in HPSG English Engine the use of ProFIT for typed feature structures is more pervasive than the use of sorts in CLE.

ProFIT also provides a very useful facility for defining templates, which are basically source language macros or parameterized abbreviations. Templates are expanded during compilation wherever they are invoked. They are used extensively in HPSG English Engine.

ProFIT does not support either dynamic constraint checking or set-valued features. Some of the techniques we will use (template expansion and difference lists) are only partial substitutes for these facilities. The CL-ONE system (Manandhar, 1994) sets out to provide these facilities, and has been partially integrated with the ProFIT system. Unfortunately, a full implementation of the CL-ONE extensions of ProFIT to support set constraints is not available.

4.2 An HPSG English Engine

In the HPSG English Engine, like the DCG English Engine described in Chapter 3, the linguistic descriptions in the grammar and lexicon are shared resources which are used, in appropriately compiled forms, for both parsing and generation. The influence of the CLE is now even clearer. In addition to a unification-based phrase structure grammar, a logical form representation, a bottom-up chart parser and a semantic head-driven generation algorithm, HPSG English Engine also includes a grammar compiler which compiles features into efficient representations for term unification.
The overall design of HPSG English Engine is shown in Figure 4.1, which can be compared with Figure 3.5 (Core Language Engine) and Figure 3.6 (DCG English Engine) in Chapter 3. Where Figure 3.6 does not show a separate grammar compiler component (though the Prolog DCG is compiled for parsing by the SAX Translator and for generation by the SGX Translator), the ProFIT grammar compiler is added to the design in Figure 4.1, in the same place as the CLE grammar compiler in Figure 3.5.

### 4.2.1 ProFIT with SAX and SGX

The main processing algorithms of HPSG English Engine have already been described in Chapter 3. We recall from Section 3.2.1 that the SAX parser is a concurrent SICStus Prolog implementation of the PAX parallel parsing algorithm of Matsumoto and Sugimura (1987), and that like the earlier BUP parser SAX uses partial execution to compile the grammar for efficient bottom-up parsing, but instead of building the chart by asserting edges into the Prolog database, SAX implements the chart by creating concurrent processes. The SGX generator (Section 3.2.1) is a concurrent SICStus Prolog implementation of the CSHD parallel chart-based semantic head-driven generation algorithm of Haruno et al. (1996), which uses partial execution to compile the grammar for efficient generation, and combines the semantic head-driven generation algorithm of Shieber et al. (1990) with a chart to avoid recomputation of partial results. Like SAX, SGX implements the chart by concurrent processes and communication streams monitored by a meta-process.

We also recall from Chapter 3 that SAX and SGX accept definite clause grammars (in SHG format for SGX) with certain restrictions. Empty categories are not supported, and SGX does not allow non-chain rules with uninstantiated logical forms. Prolog code can be added to DCG rules as extra conditions, but the extra conditions must be deterministic. This latter restriction will be important in this chapter, in connection with delayed lexical choice in Section 4.4.

The grammar in HPSG English Engine is an HPSG grammar, based on a sort hierarchy definition and typed feature structure representations implemented using the ProFIT system. However, the grammar is written in a format based on the DCG format, not in ID/LP format: the order of daughters in a phrase is not specified by separate Linear Precedence rules, as in GPSG and HPSG theory, but is specified by the left-to-right order of daughters in phrase structure rules. This does not implement HPSG theory correctly, but in this respect it is no worse than the HPSG implementation in Chapter 2, in which the ALE system similarly requires the order of daughters to be specified by their left-to-right order in the rules.
In other respects, the grammar follows HPSG theory closely, and is clearly superior to a traditional DCG-based grammar. Large DCG-based grammars typically have many rules, many categories, and many arguments per category. Such grammars can be efficiently processed by SAX and SGX, as in the case of the DCG English Engine in Chapter 3, but they are difficult to develop and debug. This is one of the reasons for adopting HPSG grammatical theory, and for using typed feature structures.

Since HPSG collects all linguistic features into a single structured sign, the many arguments in a traditional DCG category can be replaced in an HPSG category by a single argument representing the sign. Moreover, as HPSG generalizes from category-based rules (for S, NP, etc.) to schemata for phrasal signs, the many DCG rules are replaced by a few generalized rules using only the basic categories word and phrase. Since we need to specify a separate logical form in the Semantic Head Grammar (SHG) format for generation with SGX, the categories in the grammar have the format word(Sign)/LF and phrase(Sign)/LF. Here word and phrase are the basic categories, Sign is a single HPSG sign represented as a single typed feature structure, and LF is a logical form.

\[
\text{phrase}(\text{synsem}!\text{loc}!(\text{cat}!(\text{head}!\text{HF} & \\
\text{subcat}!\text{@list}1(\text{SubjSynsem})) & \\
\text{cont}!\text{Cont})))/\text{lf}(\text{Cont})
\]

\[
\#	ext{ word}(\text{synsem}!\text{loc}!(\text{cat}!(\text{head}!\text{HF} & \\
\text{subcat}!\text{@list}1(\text{SubjSynsem})) & \\
\text{cont}!\text{Cont})))/\text{lf}(\text{Cont}).
\]

Figure 4.2: The basic form of a grammar rule

Figure 4.2 shows the basic form of a grammar rule. This is a simplified version of HPSG Schema 2 with zero complement daughters. word and phrase are the basic categories. The HPSG signs inside the parentheses of the categories, beginning synsem!loc..., are represented as typed feature structures in the ProFIT notation, and lf(Cont) is a logical form. In the ProFIT notation, features are written as attribute!value. The exclamation mark is a compromise between the vertical bar used in some notations such as in (Pollard and Sag, 1994) and the colon used in others such as the ALE system. @list1 is a template which expands to a list with one member.

The # symbol is required by SGX to identify the semantic head of a chain rule for semantic head-driven generation. Note that lf(Cont) is a logical form, and Cont is a variable which is unified with the value of the CONTENT feature cont!Cont. In this grammar we follow Shieber et al. (1990) and Pollard and Sag (1994) in equating logical form with semantic content. Therefore a separate logical form is redundant, as the CONTENT feature could be used to control generation. However, we will see in Chapter 6 that this is in fact a simplistic approach.

4.2.2 Two-stage Grammar Compilation

The ProFIT system compiles the typed feature formalism into Prolog terms, which can be used with any appropriate parser or generator. We therefore use ProFIT in order to combine HPSG grammar with the SAX parser and the SGX generator, by compiling the grammar in two separate stages. The stages of processing are shown in Figure 4.3, which includes both lexicon compilation
(on the left side of the figure) and grammar compilation (on the right side of the figure). Lexicon compilation is described in Section 4.2.3.

![Diagram of lexicon and grammar compilation](image)

**Figure 4.3: Compiling the lexicon and the grammar**

In the first stage of grammar compilation, the typed feature structures in the grammar rules are compiled by ProFIT into Prolog terms. By this process, DCG rules containing HPSG signs are compiled into DCG rules containing only Prolog terms. Figure 4.4 shows the example rule of Figure 4.2 after compilation by ProFIT. (The compiled format is not intended to be read by the human grammar writer).

```
phrase('sign'(R,'phrase','synsem'(Q,'local'(F,'cat'(C,'head'(H,G),
    'list'('nelist'(E,'list'('elist'))),'cont'(B,A))))))
```

**Figure 4.4: The rule after compilation by ProFIT**

In the second stage, the resulting grammar is compiled separately by the SAX translator for parsing and by the SGX translator for generation. Since the grammar contains only Prolog terms after compilation by ProFIT, it has in effect become a normal DCG and is therefore perfectly acceptable to SAX and SGX. This second stage of grammar compilation is the same as in DCG...
English Engine. We recall that grammar rules can be labelled to be compiled only by SAX or only by SGX, and that the SAX Translator uses partial execution to produce efficient code for bottom-up chart parsing and the SGX Translator compiles tables of chain rules and also uses partial execution to produce efficient code for generation.

4.2.3 Lexicon Compilation

Unlike the ALE system (Chapter 2), we do not compile the lexicon off-line into a static list of HPSG lexical signs. Instead, the existence of an HPSG lexical sign is proved on-line by lexical rules. The lexical rules are compiled by ProFIT, so that lexical rules containing HPSG signs are compiled into lexical rules containing only Prolog terms, as shown on the left side of Figure 4.3. The basic lexicon takes the form of a lexical database, which is a list of relatively simple Prolog terms. Full HPSG lexical entries for the words in the input string are derived from entries in the lexical database by means of the lexical rules, during the SAX morphological preprocess. The derived HPSG lexical entries are inserted into the morph lattice to be passed to the SAX parser.

We will describe later in connection with delayed lexical choice (Section 4.4) how access to the lexicon is organised on two levels, making a distinction between a morphological lexicon and a syntactic-semantic lexicon. The main lexicon is the morphological lexicon, for which we specify a lexicon access interface

\[
\text{morph\_lex}(\text{Form}, \text{Cat}, [\text{LF}, \text{Sign}])
\]

where \text{Form} is a specific morphological form, \text{Cat} is the basic category (normally \text{word}), \text{LF} is a logical form and \text{Sign} is a typed feature structure. A lexical inference rule is shown in simplified form in Figure 4.5. In ProFIT, sorts are written as <sort, and features as feature!value.

\[
\begin{align*}
\text{morph\_lex}(\text{Vbse}, \text{word}, [\text{lf(Cont)}, \text{synsem=loc!}(\text{cat!}(\text{head!}(\text{vform!<bse \& aux!<n \& inv!<n}) \& \text{subcat}@list1(\text{loc!}(\text{cat!}(\text{head!<n\text{<noun} \& \text{subcat!<elist}) \& \text{cont!}(\text{Subj \& index!<ref})))) \& \\
\text{cont!}(\text{Cont \& <poca \& quants!<elist \& nucleus!(\text{rnf!Reln \& Role!Subj}))}))]) \\
\end{align*}
\]

\[
\text{verb}(\text{Vbse}, \text{Reln}, [\text{np/Role}])
\]

Figure 4.5: A morph\_lex rule for a verb base form

We use lexical inference rules to derive full HPSG lexical signs from a database of simple Prolog clauses. The example assumes a lexical entry such as

\[
\text{verb(walk, walk1, [np/agent])}
\]

specifying a verb with base form \text{walk} and sense \text{walk1}, which subcategorizes for a noun phrase subject assigned to a thematic role \text{agent}. The lexical inference rule in Figure 4.5 derives the full HPSG lexical sign for the verb base form from this Prolog clause. These lexical inference rules are therefore rather different from HPSG lexical rules, which derive new HPSG lexical signs only
from other existing HPSG lexical signs. Our lexical inference rules can re-use available non-HPSG lexical information.

However, we also use lexical rules which are like normal HPSG lexical rules, to derive new signs from other lexical signs. We use these lexical rules as in standard HPSG theory, for morphological derivations, complement extraction and so on. We have no automatic defaults, so these rules must be written carefully. The simplified example in Figure 4.6, which derives a 3rd singular verb form from a base form, instantiates nominative and 3rd singular in the first subcat item, and copies the rest of subcat by unification.

\[
\text{morph\_lex(V3sg, word, [if(Cont),}
\begin{align*}
\text{synsem\_loc!}(\text{cat!}(\text{head!}(\text{vform!}<\text{fin \& aux!}\langle n \& inv!\langle n') \& \\
\text{subcat!}(\text{first!loc!}(\text{cat!}(\text{head!}\langle \text{case!}<n\& \text{cm} \& \\
\text{subcat!}<\text{elist}) \& \\
\text{cont!}(\text{Subject \& \\
\text{index\_agr!}(3\&sg))) \& \\
\text{rest\_rest}) \& \\
\text{cont!(Cont \& nucleus\_reln!(Reln)\)])})
\end{align*}
\]

\[
:-
\text{morph\_lex(Vbse, word, [if(Cont),}
\begin{align*}
\text{synsem\_loc!}(\text{cat!}(\text{head!}(\text{vform!}<\text{bse \& aux!}\langle n \& inv!\langle n') \& \\
\text{subcat!}(\text{first!loc!}(\text{cat!}(\text{head!}\langle \text{noun} \& \\
\text{subcat!}<\text{elist}) \& \\
\text{cont!(Subject \& \\
\text{index\_agr!}(3\&sg))) \& \\
\text{rest\_rest}) \& \\
\text{cont!(Cont)\)])},
\end{align*}
\]

\[
\text{morph\_infl(verb\_3sg, Vbse, Reln, V3sg)}. \\
\]

Figure 4.6: Lexical rule for 3rd singular verb form

The typed feature structures in the lexical rules are compiled by ProFIT into Prolog terms. The resulting rules are then compiled by SICStus Prolog, together with the database of simple lexical entries. The compiled lexicon, both database and rules, is used during morphological analysis by the SAX morph preprocess, as shown on the left side of Figure 4.3.

### 4.2.4 Advantages of ProFIT/SAX over ALE

Using the ALE system to develop an English grammar as described in Chapter 2, it is possible to implement successfully almost everything in (Pollard and Sag, 1994). Unfortunately, in practice the development process can be hindered by slow compilation, especially the time required for lexicon compilation. By contrast, using the SAX parsing system with a DCG-based grammar as described in Chapter 3, both compiling and parsing are much faster than ALE. However, without an inheritance-based typed feature system it is not possible to implement HPSG theory properly. The combination of ProFIT with SAX, described in this chapter, makes it possible to combine HPSG theory with fast compiling and parsing.

Instead of using a monolithic engine such as ALE, this is a modular approach using different tools for different purposes. The tools are the SAX parsing system (a grammar translator, a morphological preprocess, and a concurrent chart parser), the SGX generation system (a grammar translator and a concurrent chart generator), the ProFIT typed feature system and SICStus Prolog.
The ProFIT inheritance-based typed feature system enables us to implement HPSG theory, and has the advantage that it can be combined with any appropriate parser or generator and with SICStus Prolog. We combine ProFIT with the SAX parser and the SGX generator, and use ProFIT with SICStus Prolog to implement lexical rules as run-time inference rules. In this section, we summarize some of the advantages of this approach over ALE.

Efficient Parsing

In the first stage of grammar compilation, the typed feature structures are compiled by ProFIT into Prolog terms, so that relatively slow unification of feature structures (as in ALE) is replaced by relatively fast unification of terms. Finite domains such as index agreement are compiled as boolean vectors for fast unification.

In the second stage, the grammar is compiled by the SAX translator, using partial execution to produce efficient code for bottom-up chart parsing. During parsing, the SAX parser implements the chart by concurrent processes monitored by a meta-process. For compiled Prolog systems, this is a highly efficient form of chart parsing, even on sequential machines.

A Finite Sort Hierarchy

In the ALE system, every individual lexical predicate must be declared as an atomic sort. Even for a modest-size lexicon, this requires a massive sort hierarchy, consisting primarily of atomic sorts. In the ProFIT system, atomic sorts for individual lexical predicates do not need to be declared. The sort hierarchy can therefore consist entirely of sorts which contribute to syntactic and semantic classification.

In order to avoid declaring specific semantic roles in the sort hierarchy for every individual verb, we specify semantic roles primarily as thematic roles, rather than as verb-specific roles as in Pollard and Sag (1994). We therefore only need a small but extensible set of thematic role declarations in the sort hierarchy, rather than a massive set of verb-specific semantic roles.

The requirement for atomic sort declarations in ALE means that whenever a new word is added to the lexicon, its atomic sort must also be added to the sort hierarchy. Since any modification of the sort hierarchy requires the whole system to be recompiled, lexicon development involves frequent recompilation.

As ProFIT does not require atomic sort declarations, new words can be added to the lexicon without extension of the sort hierarchy, provided they are further instances of existing syntactic and semantic sorts. Our use of thematic roles instead of verb-specific semantic roles means that new verbs can be added to the lexicon without extension of the sort hierarchy, provided they have an existing combination of thematic roles. In these cases only the lexicon needs to be recompiled.

Compiling Lexical Rules

The ALE system applies lexical rules “off-line” during lexicon compilation, to derive new lexical entries from existing ones. The new lexical entries are collected together with the existing entries, and all of them are compiled to produce a finite collection which is the lexicon. During parsing, each input word is simply looked up in this static lexicon. The aim of off-line lexical rule expansion, like partial evaluation, is to speed up parsing by shifting some of the processing from parse-time
to compile-time.

This approach is fine if the lexicon is small and interaction between lexical rules is limited, but it is inadequate for a realistic system. The problem is that every possible alternative sign for every possible morphological form of every possible word must be generated and added to the list. Interaction between lexical rules tends to cause an exponential explosion in the number of lexical entries which must be added. On a practical level, lexicon compilation is unacceptably slow.

There is also a theoretical problem, pointed out by (van Noord and Bouma, 1994), that it is unclear how recursive lexical rules can be treated, as they can lead to an infinite number of lexical entries. This problem is similar to the problem of infinite recursion in bottom-up parsing or generation when a grammar contains left-recursive rules, which is solved by adopting a bottom-up approach in parsing or generation. We adopt a bottom-up approach also for lexical rules, by taking the actual input as the basis of processing, and applying lexical rules during retrieval of lexical information for the actual input.

In the ProFIT/SAX system, lexical rules remain rules. The HPSG signs in the rules are compiled from typed feature structures into Prolog terms by ProFIT, so that the compiled rules can be used as normal Prolog inference rules. During lexical lookup, the rules are used deductively to prove the existence of a lexical item. Thus we do not have a static lexicon in the form of a list, rather we have a dynamic lexicon in the form of an interface. During parsing, lexicon lookup is performed by the SAX morphological preprocess, which builds a morpheme lattice to be passed to the SAX parser for syntactic parsing. The Prolog lexical inference rules are used during this preprocess.

The practical advantage of this approach is that there is no lexical rule expansion process during compilation. Each lexical rule is simply compiled into a Prolog rule, and there is no interaction between the rules at compile-time. Lexicon compilation is therefore very fast.

**Eliminating Empty Categories**

ALE supports empty categories, always inserting every empty category at every node in the chart, whether the actual input requires them or not. This is clearly inefficient. SAX does not support them (though they could be implemented by a meta-process if necessary). Like Sag (1997), we believe that theories which posit no invisible entities are a priori preferable, and that the traces and null relativizers of Pollard and Sag (1994) are undesirable.

In the ProFIT/SAX implementation, empty traces are eliminated by a Complement Extraction Lexical Rule, as proposed by Pollard and Sag (1994) in their Chapter 9. This approach would be awkward in ALE, as it would massively increase the size of the static lexicon, with a correspondingly massive increase in lexicon compilation time. In our implementation, lexicon compilation is not affected by adding CELR. During parsing, where transitive substantives appear in the actual input, the SAX morphological preprocess uses the rule to infer the existence of both the slashed and unslashed signs, and inserts them both into the morpheme lattice.

The disadvantage of not permitting empty categories is that the analysis of relative clauses presented by Pollard and Sag (1994) cannot be implemented, as it uses empty relativizers. However, a better analysis of English relative clauses is proposed by (Sag, 1997), which eliminates empty relativizers as well as empty traces. A ProFIT/SAX implementation of this revised analysis is
described in Chapter 5.

**Flexible Lexical Rules**

ALE lexical rules can only derive a new lexical entry from an existing entry. This is awkward, e.g. attributive and predicative adjectives cannot be derived from each other because there are different exceptions in both classes. Our lexical rules are not restricted in this way, so they can derive a lexical entry from data which is not part of the lexicon. They can therefore conditionally derive either alternative adjective entry, or both, from a basic list of adjectives with explicit exceptions.

This approach to lexical rules can be extended in other ways. For example, we could develop rules to access an existing source of lexical information and construct HPSG signs from whatever form of information was available.

### 4.3 An HPSG English Grammar

We now describe the grammar used by HPSG English Engine. First we show the sort hierarchy on which the whole grammar is based, and explain why some aspects of semantic representation differ from the textbook (Section 4.3.1). Then we describe the use of ProFIT templates to allow convenient abbreviations such as NP and VP when writing the grammar (Section 4.3.2). The rules for ID schemata, however, use only the general categories *word* and *phrase* (Section 4.3.3). Some of the HPSG principles are enforced by procedural attachments (Section 4.3.4).

#### 4.3.1 Sort Hierarchy

The sort hierarchy is shown in Figure 4.7. In ProFIT, the symbol > is used to show the subtypes of a supertype, equivalent to `sub` in ALE. Thus the type `bool` has exactly two subtypes `y` and `n` (equivalent to the usual Boolean `+` and `−`).

Note that the syntactic part of the sort hierarchy shown in Figure 4.7 (beginning with `cat`) follows the revisions to HPSG theory proposed in Chapter 9 (Reflections and Revisions) of the Pollard and Sag (1994) textbook. The `subcat` list is replaced by a `valency` feature `val`, which has separate lists for subjects `subj`, specifiers `spr` and complements `comps`. This grammar also includes some rather *ad hoc* additions to handle punctuation, which are not part of HPSG theory at all. They are fairly self-explanatory, and can be simply ignored.

The semantic part of the sort hierarchy shown in Figure 4.7 (beginning with `cont`) includes several significant differences from the semantic representation used by Pollard and Sag (1994). Following the experience of Section 3.3.6, a sort `prag` is included for *pragmatic particle*, in addition to the standard HPSG semantic sorts `nominal-object`, parameterized state of affairs `psoa`, and `quantifier quant`.

Following standard HPSG theory, the agreement features (gender, number and person) are carried by the `index` feature which is part of the semantic content of nominal objects. However, here we take advantage of ProFIT’s efficient compilation of finite domains using the technique of Mellish (1988). The agreement features are declared as components of a finite domain (`fin-dom`) named `agr`, which is part of the index. The index feature itself has three subtypes, referential `ref`, pleonastic `there` and pleonastic `it`, which is standard HPSG theory.
Figure 4.7: Sort Hiearchy
For the specification of nuclear semantic content as quantifier-free psoas, we take advantage of ProFIT's multi-dimensional inheritance (Erbach, 1994). qfpsa is defined as having three distinct dimensions. The first specifies the basic type of psoa, and in the case of verbal psoas distinguishes those with AGENT or EXPER (experimenter) roles. The second dimension distinguishes, quite independently, whether the psoa includes a THEME role or not. The third dimension distinguishes, also independently, whether the psoa includes a SOA_ARGUMENT argument (for example, a sentential complement) or not. The use of thematic relations (agent, experimenter, theme) rather than lexically-specified semantic roles (see, seen, etc) does not follow the standard HPSG textbook theory. It has the practical advantage that it is not necessary to define specific semantic roles in the sort hierarchy for every verb, but the system would work just as well if the textbook-style semantic roles were used.

A more important difference from the textbook theory is that when an argument fills a thematic role, the argument's entire semantic content is assigned directly within the head's content, not just the index as in the textbook. In Figure 4.7, note that the attributes AGENT, EXPER and THEME are declared as type nom_obj, not type index. These thematic roles therefore include RESTRICT as well as INDEX. This is a significant deviation from Pollard and Sag (1994).

Another important difference from the textbook theory is that this grammar does not attempt to implement storage or retrieval of quantifiers. In Figure 4.7, the type sign introduces only the attribute SYNSEM, there is no QSTORE attribute. Quantifiers are located directly in QUANTS, which is part of CONTENT. This is a second significant deviation from Pollard and Sag (1994).

Similarly, the system does not attempt to handle contextual background. In the part of the sort hierarchy in Figure 4.7 beginning with conx, only the attribute CINDS is introduced, there is no BACKGR attribute. In this grammar, background conditions such as being the bearer of a proper name are put in RESTRICT (semantic restrictions) which is part of CONTENT. This is a third significant deviation from Pollard and Sag (1994).

These three deviations are necessary in order to enable head-driven generation to work. In the case of quantified NPs, the quantifier is in QUANTS which is part of the psoa on the RESTRICT list. In the case of non-quantificational NPs, proper nouns have a name condition on the RESTRICT list (instead of on the BACKGR list), while pronouns have an empty RESTRICT list (which is in accordance with the textbook) and are distinguished from each other by their agreement features. The assignment of the RESTRICT list to thematic roles, as well as the INDEX, therefore makes available all the quantifier and background information required for generation.

In Chapter 5 we will describe an implementation which follows the textbook semantics in assigning only the INDEX (and not RESTRICT) of a semantic role within CONTENT. Because the required quantifier and background information is unavailable, that implementation is unable to perform head-driven generation. In Chapter 6 we will describe revisions to HPSG theory which will resolve this problem, enabling head-driven generation to work while following the textbook semantics in assigning only the INDEX of a semantic role within CONTENT.
4.3.2 Templates

The ProFIT facility for defining templates is illustrated in Figure 4.8. Templates are macros or parameterized abbreviations, which are defined and used in the source form of the grammar. A template is defined using the symbol `:=`, and is invoked using the symbol `@`. When the grammar is compiled by ProFIT, the templates are expanded wherever they are invoked.

```
list1(X) := (first !X & rest !<elist).
list2(X,Y) := (first !X & rest !(first !Y & rest !<elist)).
list3(X,Y,Z) := (first !X & rest !(first !Y & rest !(first !Z & rest !<elist))).
```

```
nlocal_free := synsem!nonloc!(inher!(que !<elist &
   rel !<elist &
   to_bind!(que !<elist &
   rel !<elist &
   slash !<elist)).
```

```
inher_free := synsem!nonloc!inher!(que !<elist &
   rel !<elist &
   slash !<elist).
```

```
to_bind_free := synsem!nonloc!to_bind!(que !<elist &
   rel !<elist &
   slash !<elist).
```

Figure 4.8: Some Useful Templates

Figure 4.8 shows the definitions of some useful templates, which are used simply as abbreviations of frequently occurring structures. The templates in Figure 4.9, on the other hand, are intended to help the grammar writer by allowing the use of familiar terms such as NP, AP in the source grammar. These terms are not actual categories in the grammar, which uses only the general categories word and phrase in abstract ID schemata. The templates in Figure 4.9 are simply abbreviations for the appropriate cluster of features, and are thus used in exactly the same way as the abbreviations S, NP, VP etc. are used by Pollard and Sag (1994).

Note that templates can be nested, by being invoked inside the definition of other templates. For example, the template list1 defined in Figure 4.8, and the template detp defined in Figure 4.9 are both invoked by spr !elist !detp & loc !cont !Quant) in the definition of nbar, to specify SPR as a list containing just one DetP. The templates in Figure 4.9 are used extensively to specify subcategorization in lexical entries.
Figure 4.9: Templates for DetP, Nbar, NP, AP, PP, VP
4.3.3 ID Schemata

The Head-Subject Schema and the Head-Specifier Schema are shown in Figure 4.10. In both cases, the head is marked as the semantic head by the symbol #, as required by SGX. The logical form is specified in SHG format (separated from the syntactic category by ‘/’), as required by SGX.

```
phrase(Phrase & <hd_val_ph &
   synsem!loc!(cat!(head!HF &
      val!(subj!<elist &
         spr!<elist &
         comps!<elist)) &
      cont!HeadCont))
   /lf(HeadCont)
   -->
   phrase(Subj & <hd_val_ph &
      synsem!(SubjSynsem & loc!cont!SubjCont))
   /lf(SubjCont),
   # phrase(Head & <hd_val_ph &
      synsem!loc!(cat!(head!HF &
         val!(subj!<elist &
            spr!<elist &
            comps!<elist)) &
         cont!HeadCont) &
      @to_bind_free)
   /lf(HeadCont),
   {inv_condition(Phrase, inv:n)},
   {subj_condition(Subj, Head)},
   {conx_consistency(Phrase, [Head,Subj])},
   {nonlocal_features(Phrase, [Head,Subj])}.

phrase(Phrase & <hd_val_ph &
   synsem!loc!(cat!(head!HF &
      val!(subj!<elist &
         spr!<elist &
         comps!<elist)) &
      cont!HeadCont))
   /lf(HeadCont)
   -->
   phrase(Spr & <hd_val_ph &
      synsem!loc!(cat!(head!spec!HeadSynsem &
         cont!SprCont))
   /lf(SprCont),
   # phrase(Head & <phrase &
      synsem!(HeadSynsem &
      loc!(cat!(head!HF &
         val!(subj!<elist &
            spr!<elist &
            comps!<elist)) &
         cont!HeadCont) &
      @to_bind_free)
   /lf(HeadCont),
   {conx_consistency(Phrase, [Head,Spr])},
   {nonlocal_features(Phrase, [Head,Spr])}.
```

Figure 4.10: Head-Subject and Head-Specifier Schemata
Head-Complements Schemata for 0, 1 and 2 complements are shown in Figures 4.11 and 4.12. As the SAX and SGX grammar translators are designed for DCG format grammars, with rules in which there are a fixed number of daughters, we specify the Head-Complements Schema separately for 0 complements, 1 complement, and 2 complements. Note that in each case the head daughter has the category word as required by HPSG theory, and it is also marked by # as the semantic head as required by SGX.

```
phrase(Phrase & <hd_val_ph &
    synsem!loc!(cat!(head!HF &
        val!(subj@list1(SubjSynsem) &
            spr!Sprs &
            comps!<elist)) &
        cont!HeadCont))
   /If(HeadCont)
-->
   # word(Head & <word &
    synsem!loc!(cat!(head!HF &
        val!(subj@list1(SubjSynsem) &
            spr!Sprs &
            comps!<elist)) &
        cont!HeadCont)
    @to_bind_free)
   /If(HeadCont),
{conx_consistency(Phrase,    [Head])},
{nonlocal_features(Phrase,   [Head])}.
```

Figure 4.11: Head-Complements Schema (Zero Complements)

The other schemata, for Head-Subj-Comps (Schema 3), Head-Adjunct, and Head-Filler, are omitted as they are basically similar to those shown.

### 4.3.4 HPSG Principles

In all of the schemata, the Head Feature Principle is implemented by the sharing of the variable HF in the head feature head!HF of the mother and the head daughter. The Subcategorization Principle is implemented by the items on the head daughter’s comps list (for example, Comp1Synsem and Comp2Synsem) being shared with the synsem values of the complement daughters. The Semantics Principle is implemented by the sharing of the variable HeadCont in the CONTENT feature cont!HeadCont of the mother and the semantic head daughter (and also in their logical forms).

The ID schemata in this implementation include procedural attachments for the Principle of Contextual Consistency and for the Nonlocal Feature Principle but not for the Quantifier Inheritance Principle, as can be seen in Figures 4.10 to 4.12. There is no BACKGR attribute in this implementation, as noted in Section 4.3.1, so the conx_consistency procedure merely unifies the CJNDS values. No amalgamation of background conditions or of unscoped quantifiers is involved, so the difficulties for semantic head-driven generation discussed in Chapter 6 do not arise here.
Figure 4.12: Head-Complements Schemata (One and Two Complements)
The Nonlocal Feature Principle is implemented by means of a DCG extra condition, which is attached to the schemata in the form \(\text{ncnlocal\_features(Phrase, [Head|Dtrs])}\), where Phrase is the mother, and \([\text{Head}|\text{Dtrs}]\) is a list of all the daughters with the head daughter Head first in the list, followed by the non-head daughters Dtrs. The implementation of the extra condition is shown in Figure 4.13.

\[
\text{ncnlocal\_features(synsem\!nonloc\!inher\!(que\!\&\!rel\!\&\!slash\!)(\text{Phrase}, [\text{Head}|\text{Dtrs}]))} \\
  \text{:- que\_union([\text{Head}|\text{Dtrs}], \text{Inher\_Que}), !,} \\
  \text{rel\_union([\text{Head}|\text{Dtrs}], \text{Inher\_Rel}), !,} \\
  \text{slash\_union([\text{Head}|\text{Dtrs}], \text{Inher\_Slash}), !,} \\
  \text{subtract\_que(\text{Inher\_Que}, \text{Head}, \text{Que}), !,} \\
  \text{subtract\_rel(\text{Inher\_Rel}, \text{Head}, \text{Rel}), !,} \\
  \text{subtract\_slash(\text{Inher\_Slash}, \text{Head}, \text{Slash}), !.}
\]

\[
\text{que\_union([], <elist) :- !.} \\
\text{que\_union([synsem\!nonloc\!inher\!que\!\&\!\text{Rest}], \text{QueUnion})} \\
  \text{:- que\_union(\text{Rest}, \text{Que\_Rest}), !,} \\
  \text{union\_FIT(\text{Que\_Union}), !.}
\]

\[
\text{rel\_union([], <elist) :- !.} \\
\text{rel\_union([synsem\!nonloc\!inher\!rel\!\&\!\text{Rest}], \text{RelUnion})} \\
  \text{:- rel\_union(\text{Rest}, \text{Rel\_Rest}), !,} \\
  \text{union\_FIT(\text{Rel\_Union}), !.}
\]

\[
\text{slash\_union([], <elist) :- !.} \\
\text{slash\_union([synsem\!nonloc\!inher\!slash\!\&\!\text{Rest}], \text{SlashUnion})} \\
  \text{:- slash\_union(\text{Rest}, \text{Slash\_Rest}), !,} \\
  \text{union\_FIT(\text{Slash\_Union}), !.}
\]

\[
\text{subtract\_que(\text{Inher\_Que}, synsem\!nonloc\!to\_bind\!que\!\&\!\text{To\_Bind\_Que}, \text{Que})} \\
  \text{:- subtract\_FIT(\text{Inher\_Que}, \text{To\_Bind\_Que}, \text{Que}).}
\]

\[
\text{subtract\_rel(\text{Inher\_Rel}, synsem\!nonloc\!to\_bind\!rel\!\&\!\text{To\_Bind\_Rel}, \text{Rel})} \\
  \text{:- subtract\_FIT(\text{Inher\_Rel}, \text{To\_Bind\_Rel}, \text{Rel}).}
\]

\[
\text{subtract\_slash(\text{Inher\_Slash}, synsem\!nonloc\!to\_bind\!slash\!\&\!\text{To\_Bind\_Slash}, \text{Slash})} \\
  \text{:- subtract\_FIT(\text{Inher\_Slash}, \text{To\_Bind\_Slash}, \text{Slash}).}
\]

\[
\text{union\_FIT(<elist, \text{Ys}, \text{Ys}) :- !.} \\
\text{union\_FIT(\text{Ys}, <elist, \text{Ys}) :- !.} \\
\text{union\_FIT(\text{first}!\text{X} \& \text{rest}!\text{Xs}, \text{Ys}, \text{first}!\text{X} \& \text{rest}!\text{Zs})} \\
  \text{:- union\_FIT(\text{Xs}, \text{Ys}, \text{Zs}).}
\]

\[
\% \text{subtract\_FIT (only for singleton sets as lists)} \\
\text{subtract\_FIT(<elist, <elist, <elist) :- !.} \\
\text{subtract\_FIT(\text{X}, <elist, \text{X}) :- !.} \\
\text{subtract\_FIT(\text{X}, \text{X}, <elist) :- !.}
\]

Figure 4.13: Nonlocal Feature Principle

The procedure implements the Nonlocal Feature Principle for each of the three nonlocal features QUE, REL and SLASH by first forming the unions of all the daughters’ INHER sets for each of the three features and then subtracting from the unions any items from the head daughter’s TO-BIND sets for each of the three features.
4.4 Delayed Lexical Choice

We recall from Section 3.2.2 that SGX requires all extra conditions to be deterministic, and if an extra condition is not in fact deterministic, the solution which happens to be found first will be taken as the only solution. This restriction causes difficulties for lexicon access, which is most conveniently performed by means of an extra condition. The problem is simple but serious: such lexical access will find only one lexical item, when there might be many items which should be considered. Which item is actually found will depend on irrelevant matters such as the order in which specific rules happen to be written down.

This problem does not arise with the SAX parser, because the SAX morphological preprocess can find all alternative lexical items before parsing starts. There is no equivalent lexical preprocess for the SGX generator, so SGX requires lexical items to be specified by rules included in the DCG grammar. Either every word needs its own specific rule in the grammar, which is clearly impractical except for extremely small lexicons, or a general rule in the grammar must perform lexicon access by an extra condition, which must then be deterministic. We now describe a partial solution for this problem, by means of delayed lexical choice.

4.4.1 Monotonicity and Subsumption

Delayed lexical choice is an established technique in natural language generation. When a backtracking algorithm is combined with a lexicon of morphological forms, there is considerable nondeterminism during syntactic generation, because features required for a deterministic choice of morphological form are not yet instantiated. With delayed lexical choice, a lexicon of stems is used during syntactic generation, and the choice of morphological form is delayed to a postprocess. Instead of producing a string of word forms, syntactic generation produces a string of lexical items. The morphological postprocess converts the lexical items to final lexical forms, when all required syntactic features have become instantiated.

In the original papers on semantic head-driven generation, Shieber et al. (1990) suggested the use of delayed lexical choice to improve the efficiency of the algorithm. Describing the implementation of delayed lexical choice in the MiMo2 system, they pointed out that only monotonic choices (which further instantiate the feature structure of a lexical item but do not change it) can be delayed. For example, the choice of singular or plural verb form can be delayed until after the subject has been generated, by performing syntactic generation with a lexical item based on the verb stem, which does not specify singular or plural.

By contrast, the standard HPSG lexical rule for passivization which changes the order of items on the subcat list is nonmonotonic. Both the active and the passive variants must be made available as distinct lexical items during syntactic generation. However, in an HPSG-style inheritance-based typed feature formalism, many aspects of monotonicity are "built in", due to the subsumption relation in the sort hierarchy. A sort subsumes its subsorts, which may further instantiate its features, but cannot change them. We therefore exploit the monotonicity of subsumption in the sort hierarchy in our implementation of delayed lexical choice.
4.4.2 Syntactic-Semantic Lexicon

The main lexicon, described in Section 4.2.3, is the morphological lexicon. This is a lexicon of specific word forms, with the interface

\[
morph\_lex(\text{Form}, \text{Cat}, [\text{LF}, \text{Sign}])
\]

where \text{Form} is a specific word form. Clearly, if this interface is used for generation to find a word form \text{Form} from a given logical form \text{LF}, there will be many alternative word forms for most logical forms, and lexicon access will be highly nondeterministic.

The established approach to delayed lexical choice is to replace the morphological lexicon with a lexicon of stems for the syntactic generation stage. This was done, for example, in the MiMo2 system referred to by Shieber et al. (1990). However, we adopt a slightly different approach which extends and generalises the basic idea, and is more flexible. In place of a lexicon of stems, we specify a syntactic-semantic lexicon. One advantage of this approach is that it can be applied to classes of words which do not have any kind of stems, such as pronouns. Another advantage is that it can be applied in order to delay choices for any kind of feature, not only for morphological forms. We discuss these possibilities in Section 4.4.4.

Like the morphological lexicon, the syntactic-semantic lexicon is not a static list of signs, but a dynamic lexicon interface which proves the existence of a lexical item using lexical rules. The syntactic-semantic lexicon has the interface

\[
synsem\_lex(\text{Lex}, \text{Cat}, [\text{LF}, \text{Sign}])
\]

where \text{Lex} has no significance for generation, and the other parameters are the same as for the morphological lexicon. Entries in the syntactic-semantic lexicon are derived by a small number of lexical rules from entries in the morphological lexicon. Like the morph\_lex rules, the synsem\_lex rules are compiled first by ProFIT and then by SICStus Prolog.

To implement delayed lexical choice, we use the synsem\_lex interface during syntactic generation, and then use the morph\_lex interface in the morphological postprocess. We must therefore ensure that the delayed morph\_lex rules will be monotonic, in the sense that they may further instantiate the features given by the synsem\_lex interface, but they will not change them. We do that by ensuring that the synsem\_lex entries subsume the morph\_lex entries from which they are derived.

\[
synsem\_lex(\text{Lex}, \text{word}, [\text{lf}(<\text{psca})],
\text{synsem}\_loc!(\text{cat}(!\text{head}(!\text{vform}!<\text{vform} & \text{aux}!<\text{n} & \text{inv}!<\text{n}) & \text{subcat}(!\text{Subcat}) & \text{cont}(!\text{Cont})))
\]

\[
:-
morph\_lex(\text{Lex}, \text{word}, [\text{lf}(!\text{Cont})],
\text{synsem}\_loc!(\text{cat}(!\text{head}(!\text{vform}!<\text{bse} & \text{aux}!<\text{n} & \text{inv}!<\text{n}) & \text{subcat}(!\text{Subcat}) & \text{cont}(!\text{Cont}))).
\]

Figure 4.14: A synsem\_lex rule for verbs

Figure 4.14 shows a simplified form of a synsem\_lex rule for verbs. The rule derives the synsem\_lex entry from the morph\_lex base form entry, in which \text{vform} has a value of sort <\text{bse}. The \text{subcat} of the synsem\_lex entry is unified with the \text{subcat} of the morph\_lex entry, so that
the synsem_lex entry subcategorizes for the appropriate syntactic complements. The morph_lex base form entry is used so that the agreement features of the subject will not be restricted. The content values are also unified, so that the synsem_lex entry includes the appropriate semantic roles. However, the head features are not unified. The synsem_lex vform has a value of sort <vform, which is the immediate supersort of the morph_lex vform sort <bse. Instead of full unification, the synsem_lex head features subsume those of the morph_lex entry.

4.4.3 Generation with Delayed Lexical Choice

In DCG-based systems, the interface between the grammar and the lexicon can often be specified by a DCG rule which accesses the lexicon by means of an extra condition. In our framework, such a rule might be:

```
word(Sign)/LF -->
[Word],
{morph_lex(Word, word, [LF, Sign])}.
```

However, since our concurrent processing algorithms require extra conditions to be deterministic, such a rule would find only one lexical entry (the first unifiable one), which would depend on the order of lexical rules and lexical entries. In order to implement delayed lexical choice, we use a modified form of the rule:

```
sgx word(Sign)/LF -->
[Sign],
{synsem_lex(Word, word, [LF, Sign])}.
```

The label sgx shows that the rule is to be compiled only by SGX, not by SAX (using the label mechanism described in Section 3.2.3). It differs from the previous rule not only by accessing the syntactic-semantic lexicon instead of the morphological lexicon, but also by specifying that the lexical item is [Sign] instead of [Word]. That is, the output of syntactic generation is not a string of words but a string of HPSG signs.

When syntactic generation begins, the semantic head-driven generation algorithm uses drain rules (like the rule in Figure 4.2) to identify the pivot, the semantic head of the sentence. The synsem_lex entry for the pivot is then accessed by the extra condition in the DCG rule above.

Since the synsem_lex entry for verbs (Figure 4.14) does not specify subject agreement or vform subtype, but does specify subcategorization and semantic roles, it can be used equally well as the semantic head to drive syntactic generation of, say, a 3rd-singular finite clause or an infinitival complement. Since a single entry can be used in this way, the extra condition can be deterministic, as required.

If the verb is the head of an infinitival complement, its vform becomes instantiated to <bse from subcategorization by the auxiliary to. If the verb is the head of the main clause, its vform becomes instantiated to <fin (finite) by the rule for grammatical units in our grammar, which we describe next.

After syntactic generation, the string of HPSG signs is converted to a string of word forms by a morphological postprocess, which unifies the signs with entries in the morphological lexicon. As the signs are fully instantiated during syntactic generation, this postprocess is also deterministic.
In order to support processing of units other than complete sentences, we define a unit called
\texttt{gram\_unit} (grammatical unit), which is intended to be a unit which can be uttered by itself. The
idea is derived from the earlier work on a dialogue interface, described in Section 3.3.6. The rule
for \texttt{gram\_unit} is shown in Figure 4.15.

\begin{verbatim}
gram\_unit(Phrase & <phrase &
   synsem!(loc!(cat!(head!(HF & <subst & mod!<none) &
   val!(subj!<elist &
   spr!<elist &
   comps!<elist)) &
   cont!Cont)) &
   nonloc!inher!slash!<elist))
   /lf(Cont)

   # phrase(Head & <phrase &
   synsem!loc!(cat!(head!HF &
   val!(subj!<elist &
   spr!<elist &
   comps!<elist)) &
   cont!Cont) &
   @to\_bind\_free)
   /lf(Cont),
   mark(Punc & <word &
   synsem!loc!(cat!(head!<punct &
   val!(subj!<elist &
   spr!<elist &
   comps!<elist)) &
   cont!Cont))
   /lf(Cont),
   {gram\_unit\_condition(Phrase),
   {conx\_consistency(Phrase, [Head]),
   {nonlocal\_features(Phrase, [Head])}.

gram\_unit\_condition(synsem!loc!cat!head!(<verb & vform!<fin & mod!<none)) :- !.
gram\_unit\_condition(synsem!loc!cat!head!(case!<acc & mod!<none)) :- !.
gram\_unit\_condition(synsem!loc!cat!head!(<prep & prd!<y & mod!<none)) :- !.
gram\_unit\_condition(synsem!loc!cat!head!(<prag\_part)) :- !.
\end{verbatim}

Figure 4.15: Grammatical Units

We take advantage of the HPSG framework by requiring \texttt{gram\_unit} to be a phrase which is
\textit{saturated} (no unsatisfied valencies) and \textit{substantive} (\texttt{head} subtype \texttt{subst}, not \texttt{func}). In addition to
these general requirements, we impose specific conditions on \texttt{gram\_unit}: \textit{finite} VP, \textit{accusative} NP, \textit{predicative} PP, or pragmatic particle. An alternative rule for \texttt{gram\_unit} without final punctuation
is also defined for SAX only (using the label mechanism described in Section 3.2.3), to accept
inputs which don’t end with any punctuation.

4.4.4 Reversible Delayed Lexical Choice

Most forms of robust parsing are based on constraint relaxation. Our approach to delayed lexical
choice is based on using \textit{less instantiated} signs from the syntactic-semantic lexicon, rather than
the more instantiated signs from the morphological lexicon. This can be viewed as equivalent
to constraint relaxation. It therefore seems reasonable to consider \textit{reversing} the approach, using
delayed lexical choice for parsing.

Constraint relaxation in parsing typically has a two-pass approach. Strict parsing is attempted with normal grammar rules and the normal parsing algorithm. If strict parsing fails to produce a parse, relaxed parsing is attempted, using a modified algorithm or modified grammar rules. With a lexicalist grammar like HPSG it seems more appropriate to use modified lexical rules, as in our syntactic-semantic lexicon.

However, in our approach to delayed lexical choice we do not start with strict constraints and then relax them. On the contrary, we start with relaxed constraints from less instantiated signs and then further instantiate the signs as other constraints become available. Our approach is therefore incremental description refinement (Mellish, 1988) rather than constraint relaxation.

When the syntactic-semantic lexicon is used for generation, the logical form is the retrieval key, and the name of the lexeme is irrelevant. In the interface syxsem_lex(Lex, word, [LF, Sign]), the variable Lex does not need to be unified with the name of the morpheme in the morph_lex entry, and could be given another value, such as “verb”. However, if we use the syntactic-semantic lexicon for parsing, the value of this variable will be the retrieval key. If the value is taken directly from the words of the input string, it will not necessarily unify with the name of the morpheme in the morph_lex entry.

In the case of verbs (Figure 4.14), where the input word may be an inflected form but the synsem_lex entry uses the morph_lex entry for the base form, we must first use the morphological preprocess to obtain the “root” form of the word, which is the same as the base form. We then use the root form instead of the input form as the retrieval key. In the case of pronouns, which take different forms according to case and reflexivity but have no natural root form, the input form is used directly as the retrieval key (Section 4.4.5).

Since the synsem_lex entry for verbs in Figure 4.14 does not restrict subject agreement, an ill-formed input with incorrect subject-verb agreement is parsed in exactly the same way as a well-formed input. The subject agreement in the verb’s sign remains uninstantiated until the subject and the verb phrase are combined by Schema 1, when the agreement features are instantiated to those of the subject. So “she swim” is accepted, but only “she swims” is generated in a finite clause. The synsem_lex entry in Figure 4.14 also does not restrict vform, which remains uninstantiated until the verb phrase is combined into a larger phrase. So “she can swimming” is accepted, but only “she can swim” is generated, since “can” subcategorizes for a VP with vform of sort <bse.

4.4.5 Underspecification and Pronouns

Of course, different specifications in the rules for the syntactic-semantic lexicon produce different effects. In the synsem_lex entry for pronouns in Figure 4.16, instead of unifying the head feature case with the morph_lex entry, the head is specified only as sort <noun, leaving the case unspecified. There are distinct morph_lex entries for nominative and accusative forms of personal pronouns, but it is irrelevant which one happens to be found when the rule is executed, because the rule does not unify the head features which include case. So the synsem_lex entry can be used deterministically for syntactic generation, leaving the case to be instantiated from subcategorization by a verb or preposition.
In parsing, the effect of this form of the rule is that the case of an input pronoun is ignored. Whether this is good or bad depends on both the language and the level of relaxation desired. This form of the rule would clearly be unsuitable for free word order languages, but seems useful for English, accepting “for you and I” but generating “for you and me”.

```
synsem_lex(Lex, word, [if(SynsemCont),
    synsem_loc!(cat!(head!<noun &
        subcat!<elist) &
        cont!(SynsemCont & <pron &
            index!Index &
            restr!<elist))])
```

```
morph_lex(Lex, word, [if(MorphCont),
    synsem_loc!(cat!(head!<noun &
        subcat!<elist) &
        cont!(MorphCont & <pron &
            index!Index &
            restr!<elist))])
```

Figure 4.16: A synsem_lex rule for pronouns

In Figure 4.16, the synsem_lex *content* value is not unified with the morph_lex *content* value. Only the *index* values are unified, including the gender, number and person features essential for pronouns (the *restr* values are empty lists). The *content* values are constrained only to be of sort <pron> (pronominal). In the sort hierarchy, <pron> has subsorts <ana> (anaphoric) and <ppro> (personal-pronominal), and <ana> has its own subsorts <refl> (reflexive) and <recp> (reciprocal). HPSG binding theory is based on these sortal distinctions, which are part of the *content* value.

Again, there are distinct morph_lex entries for reflexive and personal-pronominal forms, but it is irrelevant which one happens to be found when the rule is executed, because the rule does not unify the *content* values. Therefore the synsem_lex entry can be used deterministically for syntactic generation before the sort becomes instantiated to <ana> or <ppro> by the binding principles.

The effect of this form of the rule is to relax the binding constraints in parsing, accepting “I saw me” but generating “I saw myself”. Of course the distinction between “They saw themselves” (co-indexed) and “They saw them” (contra-indexed) is also lost in parsing with this version. The binding constraints can be re-instated simply by unifying the *content* values in the rule, but the above version is not necessarily bad, for example in parsing non-native English. The rule could be improved by having alternative forms which distinguish 3rd and non-3rd person.

This thesis is not concerned with robust parsing. However, the idea of using delayed lexical choice in reverse makes it possible, without modifying the parsing and generation algorithms, to parse certain types of ill-formed inputs and to generate corresponding well-formed outputs, using the same shared linguistic descriptions.
4.5 Summary

In this chapter we have described how HPSG English Engine uses SAX and SGX like DCG English Engine (in Chapter 3) to perform efficient parsing and generation, but combines them with an HPSG grammar based on a typed feature structure representation implemented using the ProFIT typed feature system.

The HPSG grammar described in Section 4.3 follows the Pollard and Sag (1994) textbook very closely as far as syntactic theory is concerned. However, a number of deviations from the semantic representation used by Pollard and Sag (1994) are discussed in Section 4.3.1. These deviations are necessary in order to enable head-driven generation to work. Specifically, in the case of quantified NPs the quantifier is in QUANTS which is inside the RESTR list, and in the case of non-quantificational NPs any background conditions are in the RESTR list instead of the BACKGR list. Crucially, the assignment of the RESTR list to thematic roles makes available the quantifier and background information required for generation.

A range of examples of parsing and generation with this system are shown in Appendix A. However, there are two major restrictions in this implementation which arise from the combination of SAX and SGX with HPSG grammatical theory. First, SAX and SGX do not support empty categories, whereas Pollard and Sag (1994) use empty categories both for empty traces and for empty relativizers. The problem of empty traces was solved by means of a complement extraction lexical rule as proposed in Chapter 9 of Pollard and Sag (1994). This implementation can therefore successfully handle unbounded dependencies, as shown in Appendix A (Kim we know Sandy claims Dana hates). However, the need for empty relativizers means that this implementation can neither parse nor generate English relative clauses.

The second restriction is that this implementation cannot generate sentences involving modification by adjuncts, although it is able to parse them as shown in Appendix A (I would like to hire a big red car). This is an implementational problem, rather than a theoretical one. Although SAX and SGX are extremely efficient, they impose restrictions on the internal copying of variables which can have the effect that some linguistically plausible rule formulations will not work. It is also significant that the SAX parser is a mature system which has been extensively tested and is widely used in Japan, whereas the SGX generator has never moved beyond the status of an experimental prototype. In the case of modification by adjuncts, the rule formulation which works successfully with SAX does not work with SGX.

Solutions for both of these restrictions will be developed in Part II of the thesis. In Chapter 5 we describe an implementation which follows the proposals of Sag (1997) in eliminating all empty categories (empty relativizers as well as empty traces) from HPSG theory. This makes it possible to handle English relative clauses successfully using the SAX parser, which does not support empty categories. The system in Chapter 5 is intended to be a faithful implementation of the theoretical analysis, so (unlike the system in this chapter) it does not deviate from the standard HPSG semantic representation. It follows the textbook semantics in assigning only the INDEX (and not RESTR) of a semantic role within CONTENT. Because this means that the required quantifier and background information is unavailable, the system in Chapter 5 is unable to perform any kind
of head-driven generation at all. It can only perform parsing.

In Chapter 6 we describe further revisions to HPSG theory which solve this problem, enabling head-driven generation to work successfully while following the textbook semantics in assigning only the INDEX of a semantic role within CONTENT. In order to demonstrate the feasibility of the methods proposed, we describe a prototype implementation using a head-driven generation algorithm which is much simpler and clearer than SGX.

We also show in Chapter 6 how adjuncts can be specified so that they can serve as semantic heads, as proposed by Pollard and Sag (1994). We illustrate how adjuncts can incorporate the required quantifier and background information obtained from the syntactic heads which they modify. We include examples of sentences involving modification by adjuncts in a demonstration of head-driven generation using the HPSG textbook semantic representation, which is made possible by the theoretical revisions proposed in Chapter 6.
Part II

Problems and Solutions in HPSG
(Post-1994 Revisions)
Chapter 5

Eliminating Empty Categories: Relative Clause Constructions

All the implementations described in Chapters 2, 3 and 4 were based on the HPSG theory presented by Pollard and Sag (1994). In Chapters 5, 6 and 7 we will investigate some problems in the 1994 theory and implement some major revisions which have been proposed in the period 1994-98. In this chapter we describe an implementation of the proposals of Sag (1997), in which empty categories are completely eliminated from HPSG theory. These proposals include a revised analysis of English relative clauses.

5.1 Problems in the 1994 HPSG Theory

The HPSG English Engine described in Chapter 4 can successfully parse and generate sentences using HPSG. The grammar follows the 1994 textbook version of HPSG and is implemented with an appropriate typed feature structure formalism. However, there are at least two major problems with the grammar, which we have not yet discussed.

First, the grammar in Chapter 4 does not handle relative clauses. The reason for this is that the analysis of relative clauses presented by Pollard and Sag (1994) involves empty relativizers. Two distinct relativizers are posited, both of which are phonetically null empty categories, like the empty traces used in unbounded dependencies. The grammar in Chapter 2 specifies these traces and relativizers using the special ALE lexical entry format for empty categories, described in Sections 2.3.4 and 2.3.5. The grammar in Chapter 4, however, cannot do this as it is used with SAX and SGX which do not permit empty categories. The empty traces are therefore eliminated by means of lexical rules, following the alternative "traceless analysis" proposed by Pollard and Sag (1994) in their Chapter 9 revisions of HPSG theory. The analysis of relative clauses with empty relativizers cannot be directly implemented with SAX and SGX. It would require the use of a meta-process, as described by Matsumoto (1991) for handling co-ordination and by Imaichi and Matsumoto (1995) for handling ill-formed inputs. We do not consider this to be a fault in SAX and SGX, but a fault in the 1994 HPSG theory which included empty categories taken over from earlier theories.
Second, the grammar in Chapter 4 does not handle quantifier scoping. The reason for this is that the analysis of quantifier scoping presented by Pollard and Sag (1994) involves Cooper storage. Quantifiers are considered as starting out in storage, and are then retrieved at some suitable point during the analysis. This mechanism causes great difficulties for semantic head-driven generation, as the identity of the semantics between a mother and the semantic head daughter is disrupted whenever a quantifier retrieval takes place. We will investigate this problem further in Chapter 6, together with a related problem in the handling of contextual background information.

5.2 Revisions to HPSG Theory since 1994

We now briefly summarize some of the main revisions to HPSG theory which have been proposed and generally accepted in the period from 1994 to 1998. These revisions will be discussed separately in the chapters of Part II which describe their implementation.

Argument Structure and Valency

In the main theory of Pollard and Sag (1994), which we implemented in ALE as described in Chapter 2, subcategorization information is specified in the SUBCAT list. The binding theory is also specified in relation to the SUBCAT list, as we described in Section 2.3.6. One of the revisions accepted by Pollard and Sag (1994) in their Chapter 9 was the proposal of Borsley (1989) to specify subcategorization separately for subjects, specifiers and complements. In this revision, these are given separate lists within a new VALENCY feature, as in our implementation in Chapter 4. The combined SUBCAT list is, however, not abolished but is retained by Pollard and Sag (1994) as the concatenation of the three valency lists, in order to preserve the binding theory unchanged.

In a subsequent revision proposed by Manning and Sag (1995), published later as (Manning and Sag, 1999), the retained SUBCAT list is renamed ARGUMENT-STRUCTURE (ARG-ST). The ARG-ST list continues to provide the basis for the binding theory, but it is no longer necessarily the concatenation of the three valency lists. Reasons for possible dissociations between the ARG-ST list and grammatical relations in the VALENCY lists are given by Manning and Sag (1995). We will see in Chapter 6 that ARG-ST can serve not only as the locus of binding theory but also as the locus of lexical constraints for nonlocal features, quantifier scoping, and contextual features.

Given the dissociation between ARG-ST and VALENCY, constraints on linking between the members of ARG-ST and the members of VALENCY were investigated by Davis (1996). His terminology for semantic relations (ACT for actor, UND for undergoer, etc.) has been adopted in recent work. In this chapter we adopt the use of ARG-ST and VALENCY, but use older names for thematic relations (agent, theme, etc.).

Constructions and Construction Grammar

Construction Grammar (Fillmore and Kay, 1993; Kay, 1995) has been a major influence on HPSG theory recently. Construction Grammar has "grammatical constructions as basic building blocks" (Fillmore, 1999). Inspired by this approach, Sag (1997) developed a complete analysis of relative clause constructions in HPSG which does not posit the existence of any empty relativizers. In Section 5.3 we describe an implementation of Sag's analysis. The use of constructions has also
strongly influenced the way in which Minimal Recursion Semantics has developed (see below).

**Quantifier Scoping**

As we mentioned, the grammar in Chapter 4 does not handle quantifier scoping, because the mechanism based on Cooper storage in Pollard and Sag (1994) causes difficulties for semantic head-driven generation. However, this is by no means the only problem.

Aside from generation, the mechanism for quantifier scoping in Pollard and Sag (1994) cannot produce the correct scopings, in some cases, even from the point of view of analysis. Proposals for revisions in the theory of quantifier storage and retrieval are made by Pollard and Yoo (1995). The first step is to make QSTORE a local feature within SYNSEM, rather than an independent feature of the sign. This change is also proposed by Frank and Reyle (1995) in an alternative analysis based on Underspecified Discourse Representation Structures.

Following the ideas of Pollard and Yoo (1995), further revisions by Manning et al. (in press) have led to a proposal for the lexicalization of quantifier scoping. We will adopt this proposal in Chapter 6 (Section 6.1.3). In this chapter however, QSTORE remains a feature at the outer level of the sign.

**Minimal Recursion Semantics**

The approach to semantics presented by Pollard and Sag (1994) is based on Situation Semantics. Like the syntactic representations in HPSG, which make use of typed feature structures and unification, the semantic representations are also typed feature structures. The structures represent a version of Situation Semantics in which the operation of semantic composition is implemented by unification of the semantic features of the component parts.

With this kind of representation, large and complex semantic structures can be built up, consisting of feature structures nested within feature structures, to arbitrary levels of nesting. Such structures can be processed successfully by unification, and can be used in a head-driven approach to generation, as we have shown in Chapter 4. However, we have only been dealing with monolingual analysis and generation processes.

In the case of bilingual or multilingual processing, such as in the transfer stage of a machine translation system, it is arguable that such complex, deeply nested structures are difficult to handle. Flat, list-based semantic representations tend to be preferred, with list concatenation as the basic operation rather than unification. In order to use HPSG in a machine translation application, Copestake et al. (1995) developed Minimal Recursion Semantics (MRS) to serve as such a flat, list-based semantic representation. In order to integrate MRS with HPSG’s syntactic representations, however, MRS also makes use of typed feature structure representations within each element in the flat lists.

The development of MRS semantics coincided with a movement towards construction-based theories in HPSG. As a result of this, MRS has been developed in a way which reduces the significance of semantic heads in HPSG, so that MRS is more suitable for non-head-driven approaches to generation. We will investigate the significance of semantic heads in HPSG in Chapter 6, and we will discuss MRS and non-head-driven approaches to generation in Chapter 7. In this chapter,
we continue to use unification-based nested feature structures as semantic representations.

5.3 Implementing Sag 1997

We now describe an implementation of the revisions proposed by Sag (1997), which completely eliminate empty categories from HPSG. We consider these revisions to be a significant improvement over the theory of Pollard and Sag (1994) purely in terms of linguistic theory. However, they are especially welcome because they also eliminate the problem of how to handle relative clauses in the framework described in Chapter 4.

Since SAX and SGX do not permit empty categories in normal processing, the theory of Pollard and Sag (1994) would require special handling of extraction and relative clauses by means of a meta-process. The use of a meta-process in a concurrent process framework, as proposed by Matsumoto (1991), seems to be appropriate in cases where some distinction between “normal” and “abnormal” processing is motivated. It seems particularly suitable for handling ill-formed inputs, as described by Imaiichi and Matsumoto (1995). In the cases of extraction (Sag and Fodor, 1994) and relative clauses (Sag, 1997), it is not evident that such special processing is motivated.

5.3.1 Eliminating Traces by Lexical Rule

The elimination of empty categories has two parts: the elimination of empty traces and the elimination of empty relativizers. We begin with the elimination of empty traces.

```
morph_lex(Vbse, word, [lf(Cont), @verb(bse) &
    phon![Vbse] &
    synsem!(loc!((cat!(head!(<verb & vform!<bse &
        aux!<minus & inv!<minus & mod!<none) &
        val!(subj![[Subj] &
        spr![] &
        comps![])) &
        arg_s![[Subj,
        Comp & QGAP]) &
        cont!Cont &
        conx!backgr!Backgr) &
        nonlocal!NonLoc) &
        qstore![]))

:=
  morph_lex(Vbse, word, [lf(Cont), @verb(bse) &
    synsem!(loc!((cat!(head!(<verb & vform!<bse &
        aux!<minus & inv!<minus & mod!<none) &
        val!(subj![[Subj] &
        spr![] &
        comps![] &
        arg_s![[Subj,
        Comp]) &
        cont!Cont &
        conx!backgr!Backgr) &
        qstore![]))]

nonlocal_amalgam([Subj, Comp], NonLoc).
```

Figure 5.1: Complement Extraction Lexical Rule

Adopting the alternative analysis proposed in Chapter 9 of Pollard and Sag (1994) as a revision
of HPSG theory, we eliminate empty traces by means of lexical rules. Figure 5.1 shows a ProFIT implementation of the Complement Extraction Lexical Rule. This version of the rule only performs extraction of the first complement on the COMPS list of the base form of a verb. Additional complement extraction lexical rules perform extraction of the second complement, and extraction of the complement of other parts of speech. Other lexical rules, taking the output of this rule as their input, derive “complement-extracted” finite and other verb forms from the “complement-extracted” base form.

The implementation described in Chapter 4 also used a Complement Extraction Lexical Rule. However, the formulation of the rule shown in Figure 5.1 differs from the previous one, due to the introduction of the ARG-ST feature (argu), which was not present in the previous implementation. Although the extracted complement disappears from the COMPS list in the VALENCY feature (val), it remains on the ARG-ST list. The complement in ARG-ST is specified in Figure 5.1 to be ”@’GAP’, using a ProFIT template defined as shown in Figure 5.2. The template expands to specify type <gap, with its own LOCAL value on its SLASH list.

\[ \text{Figure 5.2: Template for } \text{gap} \]

This use of a ProFIT template is an approximation to the type constraint which Sag (1997) formulates as shown in (5.1). (In these representations, the constraints specified on the right apply to the type specified on the left).

\[ (5.1) \quad \text{gap-synsem} \Rightarrow \begin{bmatrix} \text{LOCAL} & \text{gap} \\ \text{SLASH} & \{\text{gap}\} \end{bmatrix} \]

The ProFIT implementation of the rule shown in Figure 5.1, in combination with the other complement extraction lexical rules to perform extraction of second complements and extraction of the complement of other parts of speech, is an approximation to the general form of CELR, which Sag (1997) formulates as shown in (5.2).

\[ (5.2) \quad \text{Complement Extraction Lexical Rule (CELR):} \]

\[ \begin{bmatrix} \text{word} \\ \text{COMPS} & \bigcirc & \langle \text{gap} \rangle \end{bmatrix} \Rightarrow \begin{bmatrix} \text{COMPS} & \text{gap} \end{bmatrix} \]

Here \( \bigcirc \) designates the sequence union operation of Reape (1994). The idea is that the COMPS list of the input includes an element (at some unspecified position) which is not in the COMPS list of the output. This element is constrained to be of type \( \text{gap} \) (\( \text{gap-synsem} \)). In the output, this element of type \( \text{gap} \) will appear only on the ARG-ST list. The ProFIT rule in Figure 5.1 is an approximation to this general CELR.
5.3.2 Lexicalization of Nonlocal Features

One of the proposals of Sag (1997) changes the way nonlocal features are amalgamated. This is implemented in the rule in Figure 5.1 by means of a procedural attachment, in the line

\[ \text{nonlocal\_amalgam([Sub], Comp], NonLoc).} \]

In Pollard and Sag (1994), the Nonlocal Feature Principle requires each of the INHERITED features QUE, REL and SLASH of a phrase to be the set union of the equivalent feature of all the daughters, minus any elements of the equivalent TO-BIND sets of the head daughter. This constraint is specified on phrases: the values of the nonlocal features of all the daughters of a phrase are amalgamated by set union (minus any elements to be bound off) to give the values of the nonlocal features of the phrase.

In the revision proposed by Sag (1997), nonlocal feature amalgamation is specified by constraints on words instead of constraints on phrases. This is done by using ARG-ST. As a word’s arguments are lexically specified in its ARG-ST list, the word’s nonlocal features can be defined as amalgamations of the nonlocal features of its arguments. This form of amalgamation, specified by lexical constraints, is called \textit{lexical amalgamation}. We will investigate it extensively in Chapter 6.

For the lexical amalgamation of nonlocal features, Sag (1997) specifies lexical amalgamation of SLASH as in (5.3), where \( \perp \) designates disjoint set union and \( \Rightarrow \) designates contained set difference.

(5.3) \[ \text{SLASH Amalgamation Constraint:} \]

\[
\begin{align*}
\text{word} & \Rightarrow \\
& \begin{cases}
\text{BIND} & \perp \\
\text{ARG-ST} & \left\{ \text{SLASH} \perp, \ldots, \text{SLASH} \perp \right\} \\
\text{SLASH} & \perp \ldots \perp \Rightarrow \perp
\end{cases}
\end{align*}
\]

This SLASH Amalgamation Constraint allows a simplification of the mechanism for inheriting SLASH values. In particular, the rather clumsy duplication, distinguishing QUE, REL and SLASH within an INHERITED feature from QUE, REL and SLASH within a TO-BIND feature, is no longer necessary. Sag states a new SLASH Inheritance Principle (SLIP) as phrasal constraint (5.4). Here \( \textit{hd-nexus-ph} \) means the head-nexus-phrase type from the phrase type hierarchy (Section 5.3.3) and \( \Rightarrow \) indicates a default value. The combination of (5.3) and (5.4) means that a phrase inherits the SLASH values of its daughters indirectly, via the head daughter.

(5.4) \[ \text{SLASH Inheritance Principle (SLIP):} \]

\[
\begin{align*}
\text{hd-nexus-ph} & \Rightarrow \\
& \begin{cases}
\text{SLASH} & \perp \\
\text{HD-DTR} & \left\{ \text{SLASH} \perp \right\}
\end{cases}
\end{align*}
\]

Sag also introduces lexical amalgamation of QUE and REL, and a Wh-Inheritance Principle (WHIP) in which QUE and REL are inherited via a phrase’s head daughter. The combination of SLIP and WHIP replaces the Nonlocal Feature Principle of Pollard and Sag (1994).

Returning to the ProFIT implementation shown in Figure 5.1, the procedural attachment calls the Prolog procedure \texttt{nonlocal\_amalgam}, which is shown in Figure 5.3. This procedure performs
the amalgamation for each of the three nonlocal features QUE, REL and SLASH. These sets are represented as Prolog lists, processed by tail recursion. Instead of using the standard `append` as the basic operation, the procedure `wappend` (wait-append) is used, so that if the variables representing the lists are not yet instantiated, `wappend` will wait until they are instantiated.

```prolog
nonlocal_amalgam(Arg_S, que!Que & rel!Rel & slash!Slash) :-
    que_amalgam(Arg_S, Que),
    rel_amalgam(Arg_S, Rel),
    slash_amalgam(Arg_S, Slash).

que_amalgam(Arg_S, []) :-
    Arg_S == [], !.
que_amalgam([nonloc!que!HQue|T], Que) :-
    que_amalgam(T, TQue),
    wappend(HQue, TQue, Que).

rel_amalgam(Arg_S, []) :-
    Arg_S == [], !.
rel_amalgam([nonloc!rel!HRel|T], Rel) :-
    rel_amalgam(T, TRel),
    wappend(HRel, TRel, Rel).

slash_amalgam(Arg_S, []) :-
    Arg_S == [], !.
slash_amalgam([nonloc!slash!HSlash|T], Slash) :-
    slash_amalgam(T, TSlash),
    wappend(HSlash, TSlash, Slash).

wappend(L0,L1,L) :-
    ( L0 == L -> L1 = []
    ; L1 == L -> L0 = []
    ; wappend0(L0,L1,L) )
.
:- block wappend0(-,-,-).

wappend0(L0,L1,L) :-
    ( L1 == [] -> L0 = L
    ; wappend1(L0,L1,L) )
.
:- block wappend1(-,?,-).

wappend1([],X,X).

wappend1([H|T],X,[H|T2]) :-
    wappend(T,X,T2).
```

Figure 5.3: Amalgamation of Nonlocal Features

### 5.3.3 Phrase and Clause Type Hierarchy

The key innovation made by Sag (1997) is the specification of a hierarchy of phrase types and an orthogonal hierarchy of clause types. The types and subtypes are organized hierarchically within the overall sort hierarchy, and in addition a number of constraints are specified, which are attached to specific phrase types and clause types.

In the ProFIT implementation, the two hierarchies can be specified directly in the overall sort hierarchy as ProFIT supports multi-dimensional inheritance, but the attached constraints can only
top > [bool, sign, mod_synsem, local, nonlocal,
category, p-o-s, case, vform, valence,
content, index, agr, qfposa,
context, c_ind].

bool > [plus, minus].

sign > [word, phrase]
intro [phon, synsem:canonical, qstore].
phrase > [hd_ph, non_hd_ph] * [clause, non_clause].
hd_ph > [hd_adj_ph, hd_nexus_ph]
intro [hd_dtr:sign].
hd_adj_ph > [simp_hd_adj_ph, hd_rel_ph]
intro [adj_dtr:sign].
hd_nexus_ph > [hd_fill_ph, hd_subj_ph, hd_spr_ph, hd_comp_ph].
hd_fill_ph > [fin_hd_fill_ph, inf_hd_fill_ph]
intro [fill_dtr:sign].
fin_hd_fill_ph > [fin_hd_fill_decl_cl, fin_wh_fill_rel_cl].
inf_hd_fill_ph > [inf_wh_fill_rel_cl].
hd_subj_ph > [fin_hd_subj_ph, hd_subj_non_cl]
intro [subj_dtr:sign].
fin_hd_subj_ph > [hd_subj_decl_cl, wh_subj_int_cl,
wh_subj_rel_cl, bare_rel_cl].
hd_spr_ph intro [spr_dtr:sign].
hd_comp_ph > [hd_comp_non_cl, yes_no_int_cl,
simp_inf_rel_cl, red_rel_cl]
intro [comp_dtrs].
clause > [imp_cl, decl_cl, int_cl, rel_cl].
decl_cl > [hd_subj_decl_cl, hd_fill_decl_cl].
int_cl > [wh_subj_int_cl, yes_no_int_cl].
rel_cl > [wh_rel_cl, non_wh_rel_cl, red_rel_cl].
wh_rel_cl > [wh_subj_rel_cl, wh_fill_rel_cl].
wh_fill_rel_cl > [fin_wh_fill_rel_cl, inf_wh_fill_rel_cl].
non_wh_rel_cl > [bare_rel_cl, simp_inf_rel_cl].
non_clause > [hd_subj_non_cl, hd_comp_non_cl, hd_adj_ph].

mod_synsem > [none, synsem].
synsem > [pro, gap, canonical]
intro [loc:local, nonloc:nonlocal].
local intro [cat:category, cont:content, conx:context].
nonlocal intro [que, rel, slash].
category intro [head:p-o-s, val:valence, arg].
p-o-s > [verbal, noun, adj, prep, det]
intro [prd:bool, mod:mod_synsem].
verbal > [verb, comp]
intro [vform:vform, aux:bool, inv:bool, mc:bool].
noun intro [case:case].
prep intro [pform].
det intro [spec:synsem].

case > [nom, acc].
vform > [bse, fin, ger, inf, pas, prp, psb].
valence intro [subj, spr, compa].

Figure 5.4: Sort Hierarchy with Phrase Types and Clause Types
be approximated by means of templates. The hierarchies of phrase types and clause types in the overall sort hierarchy is shown in Figure 5.4.

The hierarchy for semantic sorts is identical to the one in Chapter 4 (shown in Figure 4.7, so not repeated here), except that the roles AGENT, EXPER and THEME are declared as sort ref instead of sort nom_obj. That is to say, the roles are assigned only INDEX as in Pollard and Sag (1994), and not RESTR as in the non-standard semantic representation in Chapter 4.

Note that the type gap which is used in the Complement Extraction Lexical Rule in Figure 5.1 (via the template in Figure 5.2) is specified in Figure 5.4 as a subtype of synsem by the line

```
synsem > [pro, gap, canonical]
```

All (visible) words have SYNSEM of subtype canonical. The subtypes gap and pro apply only "within the system". There are no lexical entries for traces or any other empty (invisible) words.

The main point in Figure 5.4 is that phrases are classified along two distinct dimensions (phrase type and clause type) simultaneously. This is implemented using the ProFIT facility for multi-dimensional inheritance (Erbach, 1994). The specification

```
phrase > [hd_ph, non_hd_ph] * [clause, non_clause]
```

means that phrases are classified in the phrase type dimension as headed phrases (hd_ph) or non-headed phrases (non_hd_ph), and are also separately classified in the clause type dimension as clause or non_clause. Within the two dimensions, local hierarchies of phrase subtypes and clause subtypes are specified in the usual way.

### 5.3.4 Phrase and Clause Type Constraints

Before describing the implementation of the constraints on specific phrase types, we need to show how some more general constraints are specified. Figure 5.5 shows ProFIT template definitions for some HPSG principles, which should be fairly self-explanatory.

```prolog
/* Head Feature Principle */
'HFP' := synsem!loc!cat!head!HF &
   hd_dtr!synsem!loc!cat!head!HF.

/* Valency Principle */
'VALP'(subj) := synsem!loc!cat!val!subj!Subj &
   hd_dtr!synsem!loc!cat!val!subj!Subj.
'VALP'(spr) := synsem!loc!cat!val!spr!Spr &
   hd_dtr!synsem!loc!cat!val!spr!Spr.
'VALP'(comps) := synsem!loc!cat!val!comps!Comps &
   hd_dtr!synsem!loc!cat!val!comps!Comps.

/* Empty Comps Constraint */
'ECC' := synsem!loc!cat!val!comps![].

/* Semantics Principle */
'SEMP'(head) := synsem!loc!cont!Cont &
   hd_dtr!synsem!loc!cont!Cont.
'SEMP'(adjunct) := synsem!loc!cont!Cont &
   adj_dtr!synsem!loc!cont!Cont.
```

Figure 5.5: Templates for some HPSG Principles
The templates defined in Figure 5.5 are invoked inside the further definitions of templates for the constraints on specific phrase types, shown in Figure 5.6. At the highest level of the phrase type hierarchy, the template for headed phrases defined by the line

\[
\text{hd\_ph} := \text{hd\_ph} & \text{@HFP} & \text{@ECC}.
\]

means that all headed phrases are of type \text{hd\_ph}, they conform to the Head Feature Principle by invoking the template \text{@HFP}, and they conform to the Empty COMPS Constraint by invoking the template \text{@ECC}, defined in Figure 5.5.

\[
\text{hd\_ph} := \text{hd\_ph} & \text{@HFP} & \text{@ECC}.
\]

\[
\text{hd\_adj\_ph} := \text{hd\_adj\_ph} & \text{@hd\_ph} & \text{@VALP'(subj)} & \text{@VALP'(spr)} & \text{@VALP'(comps)} & \text{hd\_dtr!synsem!HeadSynsem} & \\
& \text{hd\_dtr!synsem!loc\!cat\!head!mod!HeadSynsem}.
\]

\[
\text{hd\_nexus\_ph} := \text{hd\_nexus\_ph} & \text{@hd\_ph} & \text{@SEMP'(head)}.
\]

\[
\text{hd\_subj\_ph} := \text{hd\_subj\_ph} & \text{@hd\_nexus\_ph} & \text{@VALP'(spr)} & \text{@VALP'(comps)} & \text{hd\_dtr!synsem!loc\!cat\!val!([subj!SubjSynsem]} & \\
& \text{hd\_dtr!synsem!loc\!cat\!val!([subj!spr![], & \\
& \text{subj\_dtr!synsem!SubjSynsem}.
\]

\[
\text{fin\_hd\_subj\_ph} := \text{fin\_hd\_subj\_ph} & \text{@hd\_subj\_ph} & \text{synsem\!loc\!cat\!head!([verb & vform!<fin].}
\]

\[
\text{hd\_spr\_ph} := \text{hd\_spr\_ph} & \text{@hd\_nexus\_ph} & \text{@VALP'(subj)} & \text{@VALP'(comps)} & \text{hd\_dtr!synsem!loc\!cat\!val!([spr![SprSynsem]]} & \\
& \text{hd\_dtr!synsem!loc\!cat\!val!([subj!spr![], & \\
& \text{spr\_dtr!synsem!SprSynsem}.
\]

\[
\text{hd\_comp\_ph} := \text{hd\_comp\_ph} & \text{@hd\_nexus\_ph} & \text{@VALP'(subj)} & \text{@VALP'(spr)}.
\]

\[
\text{hd\_fill\_ph} := \text{hd\_fill\_ph} & \text{@hd\_nexus\_ph} & \text{@VALP'(subj)} & \text{@VALP'(spr)} & \text{@VALP'(comps)} & \text{hd\_dtr!synsem!nonloc\!slash\!Slash} & \\
& \text{hd\_dtr!synsem!loc\!cat\!head!([verbal & \\
& \text{nonloc\!slash\!Slash[FillerLoc|Slash]} & \\
& \text{fill\_dtr!synsem!loc\!FillerLoc}.
\]

\[
\text{fin\_hd\_fill\_ph} := \text{fin\_hd\_fill\_ph} & \text{@hd\_fill\_ph} & \text{hd\_dtr!synsem!loc\!cat\!head!([verb & vform!<fin] & \\
& \text{val!subj[]}])}.
\]

\[
\text{inf\_hd\_fill\_ph} := \text{inf\_hd\_fill\_ph} & \text{@hd\_fill\_ph} & \text{hd\_dtr!synsem!loc\!cat\!head!([comp & vform!<inf] & \\
& \text{val!subj![@PRC'}}).
\]

Figure 5.6: Templates for Phrase Types

Headed phrases (\text{hd\_ph}) are classified by Sag (1997) as head-adjunct phrases (\text{hd\_adj\_ph}) or
head-nexus phrases ($hd_{nexus\_ph}$), as in Figure 5.4. The definitions of the templates for $hd_{adj\_ph}$ and $hd_{nexus\_ph}$ therefore both invoke the template for headed phrases by $\Phi{hd_{ph}}$, which automatically invokes both $\Phi{HFP'}$ and $\Phi{ECC'}$. This nesting of templates within templates is an approximation, in ProFIT, to the inheritance of constraints attached to specific sorts.

The template for $hd_{adj\_ph}$ implements the Valency Principle by invoking all three parts of $\Phi{VALP'}$. The template for $hd_{nexus\_ph}$ implements the Semantics Principle by $\Phi{SEMP'}(head)$, but leaves the various parts of the Valency Principle to be implemented by its subtypes $hd_{subj\_ph}$, $hd_{spr\_ph}$ and $hd_{comp\_ph}$. The remaining templates for phrase types in Figure 5.6 implement the various constraints which Sag (1997) attaches to specific phrase types.

Note that the template for a head-filler phrase ($hd_{fill\_ph}$) specifies that its head daughter’s category must be $<verb\_al\_type>$. The type $verb\_al\_type$ is introduced by Sag (1997) as a common supertype of both $verb$ and $comp$ (complementizer), as shown in the sort hierarchy in Figure 5.4. Head-filler phrases have two distinct subtypes, finite head-filler phrases and infinitival head-filler phrases. The templates for these subtypes of head-filler phrase constrain their head daughters to be $verb$ and $comp$ respectively, the two subtypes of $verb\_al\_type$.

The template for infinitival head-filler phrase ($inf_{hd_{fill\_ph}}$) also specifies that its head daughter’s subject must be of type $PRO$ by invoking the template $\Phi{PRO'}$. $PRO$ is a subtype of $synsem$ which corresponds to unexpressed subjects in control constructions. The ProFIT template definition for $\Phi{PRO'}$ is shown in Figure 5.7. $PRO$ is always accusative (so it can never be the subject of a finite verb), always reflexive (so its binding meets ‘Visser’s Generalization’) and always has a referential index (so it never combines with VPs requiring expletive subjects).

\begin{align*}
\text{PRO'} := & <pro \& loc!(cat!head!case!<acc \& cont!(<refl \& index!<ref>))>.
\end{align*}

Figure 5.7: Template for $PRO$

The sort hierarchy in Figure 5.4 includes the clause type hierarchy proposed by Sag (1997) as well as the phrase type hierarchy. There are not so many constraints attached to specific clause types. Sag formulates the general constraints governing all clauses as shown in (5.5).

\begin{equation}
(5.5) \quad \text{clause} \Rightarrow \begin{bmatrix}
\text{SUBJ} & \text{list}(PRO) \\
\text{HEAD} & \text{MOD} / \text{none} \\
\text{QUE} & \{\} \\
\text{REL} & \{\}
\end{bmatrix}
\end{equation}

The ProFIT templates to implement the various constraints on clauses are shown in Figure 5.8. These templates achieve the same effect as the general constraint (5.5) by slightly different means. Whereas SUBJ is specified in (5.5) as list(PRO), a list all of whose members are of type PRO, it is specified in Figure 5.8 as subj!([] or [\Phi{PRO'}]), a disjunction of an empty list or a list containing just one item of type PRO. Whereas the HEAD is specified in (5.5) as MOD '/' none,
a default value of \textit{none}, it is specified in Figure 5.8 as \texttt{mod!\textless none\textgreater} in all clause subtypes except relative clauses (\texttt{rel\_cl}), the exception to the default.

The templates in Figure 5.8 collectively implement the general and specific constraints specified by Sag (1997). Four basic subtypes of clause are specified: declarative (\texttt{decl\_cl}), interrogative (\texttt{int\_cl}), imperative (\texttt{imp\_cl}), and relative (\texttt{rel\_cl}).

\[
\text{clause} := \langle \text{clause} \& \text{synsem!(loc!cat!val!subj![]} \text{ or } \text{[@'PRC']} \rangle \& \text{nonloc!(que![]} \& \text{rel![]} \& \text{slash![]} \rangle.\% \text{slash![]} \text{GW}.
\]

\[
\text{decl\_cl} := \langle \text{decl\_cl} \& \text{@clause} \& \text{synsem!loc!cat!head!(<verb \& vform!<fin \& mod!\textless none\textgreater)} \rangle.
\]

\[
\text{int\_cl} := \langle \text{int\_cl} \& \text{@clause} \& \text{synsem!loc!cat!head!mod!<none}\rangle.
\]

\[
\text{imp\_cl} := \langle \text{imp\_cl} \& \text{@clause} \& \text{synsem!loc!cat!head!mod!<none}\rangle.
\]

\[
\text{rel\_cl} := \langle \text{rel\_cl} \& \text{@clause} \& \text{synsem!loc!cat!head!(inv!<minus \& mc!<minus \& mod!loc!cat!head!<noun)}\rangle.
\]

\[
\text{wh\_rel\_cl} := \langle \text{wh\_rel\_cl} \& \text{@rel\_cl} \& \text{synsem!loc!cat!head!mod!@np(<case, <nom\_obj}>\rangle.
\]

\[
\text{non\_wh\_rel\_cl} := \langle \text{non\_wh\_rel\_cl} \& \text{@rel\_cl} \& \text{synsem!(loc!cat!head!mod!@nbar(Index, _) \& nonloc!slash![]} \rangle \& \text{hd_dtr!synsem!nonloc!slash![cont!index!Index]}\rangle.
\]

Figure 5.8: Templates for Clause Types

Two subtypes of relative clause are specified: \textit{Wh}-relatives (\texttt{wh\_rel\_cl}) and \textit{non-\textit{Wh}} relatives (\texttt{non\_wh\_rel\_cl}). For all relative clauses the HEAD’s MOD value is always of type \textit{noun}, specified in the ProFIT template by \texttt{mod!loc!cat!head!<noun}. For the reasons given by Sag (1997), this is further restricted to NP by the template \texttt{@np} in the case of \texttt{wh\_rel\_cl}, and to Nbar by the template \texttt{@nbar} in the case of \texttt{non\_wh\_rel\_cl}.
5.3.5 Relative Clause Constructions

We can now describe how the various constraints on phrase types and clause types are used to build relative clause constructions without empty relativizers. The intention of Sag (1997) is that the specific constraints on phrase types and clause types interact with the more general constraints on ID schemata and the general HPSG principles, to license precisely the relative clauses which are OK, and block ill-formed ones. However, ProFIT has no general constraint mechanism, so we can only approximate to this intention by achieving the same effect by other means.

ProFIT implementations of the general ID schemata were specified in Chapter 4, in Section 4.3.3. The basic method which we adopt is to provide distinct forms of the general ID schemata for distinct constructions. Each distinct construction must include both a specification of its phrase type and a specification of its clause type, using the templates which we have described.

We start with the general ID schema for head-subject phrases, which was shown in Figure 4.10. In the analysis of Sag (1997), there are four distinct subtypes of head-subject phrase (more precisely, four distinct subtypes of finite head-subject phrase), as shown in the sort hierarchy in Figure 5.4. These are head-subject declarative clauses \(<hd_subj_{decl}_{cl}\>\), Wh-subject interrogative clauses \(<wh_{subj}_{int}_{cl}\>\), Wh-subject relative clauses \(<wh_{subj}_{rel}_{cl}\>\), and bare relative clauses \(<bare_{rel}_{cl}\>\).

The rule which builds a head-subject declarative clause construction is shown in Figure 5.9. It is derived from the general ID schema for head-subject phrases in Figure 4.10.

```
phrase(Phrase & <hd_subj_{decl}_{cl} & @fin_hd_subj_{ph} & @decl_{cl} & @'SLIP' & @'WHIP'(que) & @'WHIP'(rel) & synsem!loc!cat!head!inv!<minus & hd_dtr!(Head & <phrase & synsem!loc!cont!HeadCont) & subj_dtr!(Subj & <phrase & synsem!loc!cont!SubjCont)) /lf(HeadCont)
-->
phrase(Subj) /lf(SubjCont), # phrase(Head) /lf(HeadCont),
{phon_inheritance(Phrase, [Subj,Head])},
{conx_consistency(Phrase, [Head,Subj])},
{quantifier_inheritance(Phrase, [Head,Subj])}.
```

Figure 5.9: Head-Subject Declarative Clause Construction

Note that the type of the mother is \(<hd_subj_{decl}_{cl}\>\). The distinctive constraints are imposed by the two templates for the construction’s phrase type and its clause type. The line

```
@fin_hd_subj_{ph} &
```

imposes all the constraints attached to finite head-subject phrases by the template \(fin_{hd_subj_{ph}}\) defined in Figure 5.6. At the same time, the line

```
@decl_{cl} &
```


imposes all the constraints attached to declarative clauses by the template decl_cl, as defined in Figure 5.8. In this way, the ProFIT implementation can make an approximation to the general constraints intended in the theoretical analysis of Sag (1997).

The rule which builds a Wh-subject relative clause construction is shown in Figure 5.10. It is also derived from the general ID schema for head-subject phrases in Figure 4.10, but the type of the mother is <wh_subj_rel_cl>. Again, the distinctive constraints are imposed by the two templates for the construction’s phrase type and its clause type. The line

@fin_hd_subj_ph &

is the same as for the head-subject declarative clause construction, imposing the constraints attached to finite head-subject phrases. By contrast, the line

@wh_rel_cl &

imposes the different set of constraints attached to Wh-relative clauses by the template wh_rel_cl in Figure 5.8. The rule also unifies the index of the modified NP with the index in the REL feature.

```
phrase(Phrase & <wh_subj_rel_cl &
 @fin_hd_subj_ph &
 @wh_rel_cl &
 @'SLIP' & @'WHIP'(que) &
 synsem!loc!cat!head!mod!&mp(<case, index!Index) &
 hd_dtr!(Head & <phrase &
   synsem!(loc!cnt!HeadCont &
   nonloc!rel!1[Index]) &
 subj_dtr!(Subj & <phrase &
   synsem!(loc!cnt!SubjCont &
   nonloc!rel!1[Index])))
/lf(HeadCont)
-->
phrase(Subj)
/lf(SubjCont),
# phrase(Head)
/lf(HeadCont),
 phon_inheritance(Phrase, [Subj,Head]),
 (conx_consistency(Phrase, [Head,Subj])),
 (quantifier_inheritance(Phrase, [Head,Subj])).
```

Figure 5.10: Wh-Subject Relative Clause Construction

The rule which builds a bare relative clause construction is shown in Figure 5.11. Again, the distinctive constraints are imposed by the two templates for the construction’s phrase type and its clause type. The template for @fin_hd_subj_ph is the same as before, but the line

@non_wh_rel_cl &

imposes a different set of constraints attached to non-Wh-relative clauses.
So far, all the constructions we have looked at are (subtypes of) head-subject phrases, so they all include the same template @fin_hdsubj_ph for their phrase type. The differences between them have so far been made only by the different templates for their clause types. We now compare the rule for bare relative clause constructions, which is shown in Figure 5.11, with the rule for simple infinitival relative clause constructions, which is shown in Figure 5.12.

Again, the distinctive constraints are imposed by the two templates for the construction’s phrase type and its clause type. However, this time the template for the clause type, @nonwh_rel_cl, is the same in both constructions, and it is the template for the phrase type which distinguishes them. For simple infinitival relative clauses, the phrase type is head-complement phrase, and the distinctive constraints are imposed by the template @hd_comp_ph defined in Figure 5.6.
Figure 5.13: Finite Wh-Filler Relative Clause Construction

We will not exhaustively describe every distinct relative clause construction. However, we show the rules for finite and infinitival Wh-filler relative clauses in Figures 5.13 and 5.14. In Chapter 6 we will discuss finite Wh-filler relative clauses, as part of an analysis of case assignment and register restrictions within Sag’s analysis of relative clauses. We will propose an alternative approach using lexical constraints instead of constructional constraints.

Figure 5.14: Infinitival Wh-Filler Relative Clause Construction

Sag (1997) treats nonsubject relative clauses such as whose bagels I like and from whom I bought these bagels in terms of a single construction type fin-wh-fill-rel-cl. He specifies the constraints on this construction type as shown in (5.6).
This states that the filler daughter must be either an NP or a PP. Sag says that “the constraints on this type again have little work to do, though perhaps more than the simplified formulation shown”. We will return to this point in Chapter 6.

We have not yet described how the relative clauses (of whatever type) are combined within noun phrases with the nouns they modify. Sag (1997) proposes that head-relative phrases should be treated as a distinct type of construction, which is a subtype of head-adjunct phrase. Other head-adjunct phrases which are not relative clauses are therefore called simple head-adjunct phrases, shown in Figure 5.15, while head-relative phrases are shown in Figure 5.16.

There is an important difference in the way the semantic content of the phrases is made up from the content of the daughters. In the case of the simple head-adjunct phrase, the mother’s semantic content is identical to that of the adjunct daughter, as required by the basic formulation of the Semantics Principle. This is imposed here by the template @"SEMP"(adjunct), which was defined in Figure 5.5.

In the case of the head-relative phrase, the psot (parameterized state of affairs) which is the semantic content of the adjunct daughter is combined with the RESTR (restriction) list from the head daughter to create the RESTR list of the mother. This mechanism does not conform to the Semantics Principle, and neither the head daughter nor the adjunct daughter is the semantic head. In general, the use of constructions in HPSG has weakened the role of semantic heads in the organization of the grammar. We will discuss this issue further in Chapter 6.
phrase(Phrase & <hd_rel_ph &
   @hd_adj_ph &
   Not SEMP
   Not SLIP, Not WHIP
   synsem!(loc!cnt!(Cont &
      index!Index &
      restr![Noun,Psoa|Rest] &
      nonloc!NonLoc) &
   hd_dtr!(Head & <phrase &
      synsem!loc!cnt!(HeadCont &
      index!Index &
      restr![Noun|Rest]) &
   adj_dtr!(Adjunct & <rel_cl &
      synsem!(loc!cnt!(Psca & <psca) &
      nonloc!NonLoc))
   /lf(Cont)
)-->
phrase(Head)
/lf(HeadCont),
phrase(Adjunct)
/lf(Psca),
{phon_inheritance(Phrase, [Head,Adjunct])},
{conx_consistency(Phrase, [Head,Adjunct])},
{quantifier_inheritance(Phrase, [Head,Adjunct])}.

Figure 5.16: Head-Relative Phrase

5.4 Summary

We have described a ProFIT implementation of the proposals of Sag (1997). With this implementation, the wide variety of English relative clause constructions discussed by Sag can be parsed, using the SAX parser in the same way as in Chapter 4, even though SAX does not support empty categories. A demonstration execution trace showing parsing (but not generation) with this grammar is given in Appendix B.

Note that the grammar implemented here follows Sag (1997) and does not include restrictions on case assignment and register variation. Therefore two incorrect forms, the man at who we looked and the man at that we looked, are accepted as well as the correct form the man at whom we looked.

Methods for implementing the required restrictions on case assignment and register variation will be proposed in Chapter 6.

The implementation described in this chapter solves the problem of empty categories, which prevented the system in Chapter 4 from handling relative clauses. Unfortunately, although this enables SAX to parse relative clauses successfully, the system described here cannot generate them with SGX. This is not due to a specific problem with relative clauses, but is part of a more general problem with the handling of set-valued features in HPSG, which means that the system described in this chapter cannot perform generation at all. The reasons for this will be discussed in Chapter 6, where a solution will be proposed and demonstrated.
Chapter 6

Head-Driven Generation: Lexicalization of Context

This chapter presents a proposal for a further revision in HPSG theory, which we shall call the *lexicalization of context*. Instead of *phrasal* amalgamation of contextual information from a phrase’s daughters, as specified by the Principle of Contextual Consistency of Pollard and Sag (1994), we propose *lexical* amalgamation of context from a word’s arguments by means of lexical constraints. The Principle of Contextual Consistency is replaced by a Contextual Head Inheritance Principle, in which a phrase’s CONTEXT feature is token-identical to that of its contextual head daughter. We offer motivations for the proposal on theoretical, computational and linguistic levels.

The chapter has four main parts. Section 6.1 shows that the proposal naturally follows other recent revisions in HPSG theory concerned with set-valued features, and brings the handling of context features into line on a theoretical level with the lexicalization of nonlocal features and of quantifier storage. Section 6.2 argues that the lexicalization of context, combined with the lexicalization of quantifier scoping, allows the idea of semantic heads to regain its original significance, which otherwise is lost when quantifier retrieval and background conditions are involved. Section 6.3 then sketches a linguistic analysis which combines the lexicalization of context with the lexicalization of nonlocal features, in an approach to register variation within Sag’s analysis of English relative clauses. Section 6.4 describes a first attempt at implementing the lexicalization of context, using ProFiT with a very simple head-driven generator.
6.1 Set-Valued Features

We start by comparing two alternative approaches to the amalgamation of set-valued features: the phrasal amalgamation approach of Pollard and Sag (1994) and the lexical amalgamation approach of more recent proposals.

6.1.1 Phrasal Amalgamation

Three principles - the Nonlocal Feature Principle, the Quantifier Inheritance Principle, and the Principle of Contextual Consistency - specify constraints on certain set-valued features of a phrase and the equivalent set-valued features of the daughters. In Pollard and Sag (1994), these constraints are specified on phrases, that is, the values of the relevant features of all the daughters of a phrase are amalgamated by set union (possibly with subtraction of certain elements) to give the value of the relevant feature of the phrase. We therefore refer to this phrase-based amalgamation of set-valued features as *phrasal amalgamation*.

The Nonlocal Feature Principle requires each of the INHERITED features QUE, REL and SLASH of a phrase to be the set union of the equivalent feature of all the daughters, minus any elements of the equivalent TO-BIND sets of the head daughter. The Quantifier Inheritance Principle requires the QSTORE feature of a phrase to be the set union of the QSTOREs of all the daughters, minus any quantifiers in the phrase’s RETRIEVED list. The Principle of Contextual Consistency simply requires the BACKGROUND feature of a phrase to be the set union of the BACKGROUND sets of all the daughters.

In recent revisions of HPSG theory, phrasal amalgamation has been divided into two distinct parts: the amalgamation part and the inheritance part. While the inheritance part is still specified by constraints on phrases, the set-valued feature amalgamation part is now specified by constraints on words.

6.1.2 Lexical Amalgamation

Following the proposals of Manning and Sag (1995), a word’s arguments are lexically specified in its ARGUMENT-STRUCTURE (ARG-ST) list. A word’s set-valued features can now be defined in terms of the amalgamation of the equivalent set-valued features of its arguments. This form of amalgamation, specified by lexical constraints, is referred to as *lexical amalgamation*.

\[(6.1) \quad \text{SLASH Amalgamation Constraint:} \]

\[
\begin{align*}
\text{word} \Rightarrow & \left[ \begin{array}{c} \text{BIND} & [0] \\
\text{ARG-ST} & \left[ \text{SLASH} [n], \ldots, \text{SLASH} [m] \right] \\
\text{SLASH} & (\cap [n] \ldots \cap [m]) - [0] \end{array} \right] 
\end{align*}
\]

Advantages of lexical amalgamation over phrasal amalgamation have already been proposed for nonlocal features and for quantifier storage. We recall from Chapter 5 that the lexicalization of nonlocal features is described by Sag (1997), who specifies lexical amalgamation of SLASH as in (6.1), where \(\cap\) designates disjoint set union and ‘-’ designates contained set difference.
(6.2) SLASH Inheritance Principle (SLIP):

\[ \text{hd-nexus-ph} \Rightarrow \begin{bmatrix} \text{SLASH} & [\text{hd-nexus-ph}] \\ \text{HD-DTR} & [\text{SLASH}] \end{bmatrix} \]

This allows a simplification of the mechanism for inheriting SLASH values. Sag (1997) states a new SLASH Inheritance Principle (SLIP) as phrasal constraint (6.2), where `hd-nexus-ph` is the head-nexus-phrase type from Sag's phrase type hierarchy and `/'` indicates a default value. The combination of (6.1) and (6.2) means that a phrase inherits the SLASH values of its daughters indirectly, via the head daughter.

Sag (1997) also introduces the lexicalization of QUE and REL, and a Wh-Inheritance Principle (WHIP) in which QUE and REL are inherited via a phrase's head daughter. The combination of SLIP and WHIP replaces the Nonlocal Feature Principle of Pollard and Sag (1994).

### 6.1.3 Lexicalization of Quantifier Scoping

The lexicalization of quantifier scoping follows a similar approach to the lexicalization of nonlocal features. Adopting the proposals of Pollard and Yoo (1995), QSTORE is moved within the structure of the sign. Instead of being a sign-level feature as in Pollard and Sag (1994), QSTORE becomes a local feature which can be included in the features subcategorized for by a lexical head, and can therefore be lexically amalgamated in that head. These proposals have been extended to include lexicalization of quantifier retrieval by Manning et al. (in press), who specify a Quantifier Amalgamation Constraint (actually a constraint on word stems) as in (6.3), where `[ ]` is the set of retrieved quantifiers.

(6.3) Quantifier Amalgamation Constraint:

\[ \text{stem} \Rightarrow \begin{bmatrix} \text{ARG-ST} \left( \left[ \text{QSTORE} [\square], \ldots, \text{QSTORE} [\square] \right] \right) \\ \text{QSTORE} \left[ \Pi \ldots \Pi \right] - [\square] \\ \text{CONT} \left[ \text{QUANTS} \ \text{order} [\square] \right] \end{bmatrix} \]

Given the Quantifier Amalgamation Constraint, unscoped quantifiers are no longer inherited from all daughters but only from the semantic head daughter. This is stated in (6.4) as a revised Quantifier Inheritance Principle (QUIP).

(6.4) Quantifier Inheritance Principle (QUIP):

\[ \text{hd-nexus-ph} \Rightarrow \begin{bmatrix} \text{QSTORE} & [\square] \\ \text{HD-DTR} & [\text{QSTORE} [\square]] \end{bmatrix} \]

\[ \text{hd-adjunct-ph} \Rightarrow \begin{bmatrix} \text{QSTORE} & [\square] \\ \text{ADJ-DTR} & [\text{QSTORE} [\square]] \end{bmatrix} \]
6.1.4 Lexicalization of Context

We now propose the lexicalization of contextual features, following the same approach as the lexicalization of nonlocal features and the lexicalization of quantifier scoping. For the set-valued feature BACKGROUND (BACKGR), we introduce the Background Amalgamation Constraint (6.5), in which a word’s BACKGR set is the set union of the BACKGR sets of its arguments.

\[(6.5) \quad \text{Background Amalgamation Constraint:} \]

\[
\left[ \text{ARG-ST} \left( \left\{ \text{BACKGR} \right\}_1, \ldots, \left\{ \text{BACKGR} \right\}_n \right) \right]
\]

\[
\text{BACKGR} \bigcup \ldots \bigcup \text{BACKGR}
\]

Amalgamation of CONTEXTUAL-INDICES (C-INDICES) depends on how they are defined. As Pollard and Sag (1994) say, “each part of an utterance (at least each lexeme) has its own C-INDICES value.” This suggests set-valued C-INDICES, amalgamated in phrases by set union. However, as Pollard and Sag also say, it is typical of discourse situations that the contextual indices are uniform throughout an utterance. In a coarse-grained analysis, a phrase’s C-INDICES can be simply the unification of the C-INDICES of its daughters. The same simplification could be specified by the lexical constraint (6.6), in which a word’s C-INDICES feature is the unification of the C-INDICES of its arguments.

\[(6.6) \quad \text{Lexical Amalgamation of C-INDICES (simplified version):} \]

\[
\text{word} \Rightarrow \left[ \text{ARG-ST} \left( \left\{ \text{C-INDS} \right\}_1, \ldots, \left\{ \text{C-INDS} \right\}_n \right) \right]
\]

\[
\text{C-INDS} \bigcup \ldots \bigcup \text{C-INDS}
\]

If a more fine-grained analysis is required, the C-INDICES should be set-valued. For a single word the indices will usually be singleton sets, but for phrases these sets need to be amalgamated by set union. The three standard indices of Pollard and Sag (1994) then need to be specified by separate constraints, as in (6.7)

\[(6.7) \quad \text{Lexical Amalgamation of C-INDICES (set-valued version):} \]

\[
\text{word} \Rightarrow \left[ \text{ARG-ST} \left( \left\{ \text{SPEAKER} \right\}_1, \ldots, \left\{ \text{SPEAKER} \right\}_n \right) \right]
\]

\[
\text{SPEAKER} \bigcup \ldots \bigcup \text{SPEAKER}
\]

\[
\text{word} \Rightarrow \left[ \text{ARG-ST} \left( \left\{ \text{ADDRESSEE} \right\}_1, \ldots, \left\{ \text{ADDRESSEE} \right\}_n \right) \right]
\]

\[
\text{ADDRESSEE} \bigcup \ldots \bigcup \text{ADDRESSEE}
\]

\[
\text{word} \Rightarrow \left[ \text{ARG-ST} \left( \left\{ \text{U-LOC} \right\}_1, \ldots, \left\{ \text{U-LOC} \right\}_n \right) \right]
\]

\[
\text{U-LOC} \bigcup \ldots \bigcup \text{U-LOC}
\]

---

\(^3\)Here, set union should not be disjoint. There may be duplicate conditions with the same index in examples such as She saw herself.
Given the lexical amalgamation of BACKGR and C-INDICES, their values can be passed up to higher levels by a new Contextual Head Inheritance Principle (CHIP), in which a phrase’s CONTEXT is by default token-identical to that of its contextual head daughter. We will assume that contextual heads are defined in the same way as semantic heads: in a head-adjunct-phrase the adjunct daughter is the contextual head, and in a head-nexus-phrase the syntactic head is the contextual head. The principle is stated in (6.8).

\[
\begin{align*}
\text{hd-nexus-ph} & \Rightarrow \left[ \begin{array}{c}
\text{CONTEXT} / \uparrow \\
\text{HD-DTR} \left[ \text{CONTEXT} \downarrow \uparrow \right]
\end{array} \right] \\
\text{hd-adjunct-ph} & \Rightarrow \left[ \begin{array}{c}
\text{CONTEXT} / \uparrow \\
\text{ADJ-DTR} \left[ \text{CONTEXT} \downarrow \uparrow \right]
\end{array} \right]
\end{align*}
\]

The combination of (6.5) and (6.8) ensures that a phrase inherits the BACKGR values of its daughters, not directly but via the contextual head daughter. This combination replaces the Principle of Contextual Consistency of Pollard and Sag (1994).

6.2 Semantic Heads

We now link the lexicalization of context to the role of semantic heads in head-driven grammar. Though HPSG is fundamentally head-driven by syntactic heads, it is also to a secondary degree head-driven by semantic heads. The definition of semantic head in Pollard and Sag (1994) is clear: in head-adjunct phrases the adjunct is the semantic head, and in other headed phrases the syntactic head is the semantic head. The definition is intended to work together with the Semantics Principle, so that the major semantic features of a phrase are inherited from the semantic head, while the major syntactic features are inherited from the syntactic head by the Head Feature Principle.

In Pollard and Sag (1994), when an NP is assigned as an argument of a verb, only the NP’s index is directly assigned to the verb’s content. Other semantic features from the NP are distributed to other features of the VP by phrasal amalgamation. If the NP is a quantificational NP, its unscooped quantifiers are added to the VP’s QSTORE. If the NP is non-quantificational (a pronoun or proper noun), its background conditions are added to the VP’s BACKGR. In both cases, which we will look at in turn, the VP includes major semantic features which it does not inherit from the verb which is its semantic head according to the definition. In effect, phrasal amalgamation “by-passes” the semantic head, which loses its intended significance.

6.2.1 Quantificational-Semantic Heads

The original form of the Semantics Principle in Pollard and Sag (1994), Chapter 1, equates semantic content with the CONTENT feature and simply says that a phrase has the same CONTENT as its semantic head. However, the principle is reformulated in Chapter 8 to cater for quantifier storage and retrieval, because only scoped quantifiers are included in the QUANTS list within CONTENT, while unscooped quantifiers are stored in the QSTORE set which is not part of CONTENT.
TENT. In this approach to quantifier scoping, a quantifier may be retrieved from storage at any suitable syntactic node. A quantifier retrieved at a particular node is a member of the QSTORE set, but not the QUANTS list, of a daughter of that node, and due to the retrieval it is a member of the QUANTS list, but not the QSTORE set, of the mother node. As QUANTS is part of CONTENT, the effect of retrieval is that the phrase and the semantic head have different CONTENT values.

The reformulated Semantics Principle therefore makes a distinction between quantificational content and nuclear content, and requires only nuclear content (the NUCLEUS feature) to be shared between a VP and its head verb. This clearly reduces the significance of semantic heads. Though the verb is the semantic head according to the definition, it only passes one part of its semantic content up to the phrase.

In the lexicalized approach to quantifier scoping, with the Quantifier Amalgamation Constraint (6.3), this problem does not arise. Retrieval is located in the lexicon, inside the verb’s lexical entry, and does not cause a difference in either QSTORE or QUANTS between a VP and its head verb. The phrase and the semantic head have identical QSTORE, identical QUANTS and identical NUCLEUS.

The identity of QUANTS and NUCLEUS between a phrase and its semantic head is full identity of CONTENT. We can therefore return to the original form of the Semantics Principle, in which a phrase inherits the full CONTENT of the semantic head. We restate this, renamed as the Semantic Head Inheritance Principle (SHIP) in the style of QUIP and CHIP in (6.9).

\[(6.9)\]

\[
\text{Semantic Head Inheritance Principle (SHIP):} \\
\begin{cases}
hd\text{-nexus}\text{-ph} & \Rightarrow [\text{CONTENT} / \text{□} \\
\text{HD-DTR} [\text{CONTENT \text{□}}] \\
hd\text{-adjunct}\text{-ph} & \Rightarrow [\text{CONTENT} / \text{□} \\
\text{ADJ-DTR} [\text{CONTENT \text{□}}]
\end{cases}
\]

The identity of QSTORE between a phrase and its semantic head was stated earlier as a revised Quantifier Inheritance Principle (QUIP) in Section 6.1.2. The combination of SHIP and QUIP means that a phrase inherits all of its nuclear and quantificational content from the semantic head.

6.2.2 Contextual-Semantic Heads

Non-quantificational NPs such as pronouns or proper nouns generally have contextual background conditions. The way these are handled parallels the way a quantificational NP’s unscoped quantifiers are handled. In both cases, only the NP’s index is assigned to the verb’s NUCLEUS. With phrasal amalgamation, a phrase inherits background conditions as well as unscoped quantifiers from all daughters. In general therefore, a phrase and its semantic head will have the same NUCLEUS feature but will have different BACKGR and QSTORE features.

Figure 6.1 shows the standard analysis of *She saw Kim* with phrasal amalgamation. *She* has a non-empty contextual BACKGR set (shown by tag [□]), stating a pragmatic requirement that
the referent is female. This background condition is passed up from NP to S by the Principle of Contextual Consistency. Similarly, Kim has a background condition (shown by tag [4]) that the referent bears this name. This condition is also passed from NP to VP, and from VP to S.

As there are no quantifiers, V is the semantic head of VP and VP is the semantic head of S not merely by definition but also in the sense of the original Semantics Principle, since S, VP and V all share the same CONTENT (shown by tag [1]). However, VP includes the BACKGR condition shown by tag [5] which it does not inherit from V, and S includes the BACKGR condition shown by tag [4] which it does not inherit from VP. If semantic features are understood in a wider sense, not restricted to the CONTENT feature, then in both cases the phrase includes a major semantic feature which is not inherited from its semantic head.

With lexical amalgamation, by contrast, the BACKGR sets of she and Kim are amalgamated in the verb’s lexical entry by the Background Amalgamation Constraint (6.5). So the empty BACKGR set of saw in Figure 6.1 is changed from BACKGR {} to BACKGR {4, 5}. This set is inherited by VP from V and by S from VP by the Contextual Head Inheritance Principle (6.8). Since the contextual heads and the semantic heads are the same, all major semantic features (nuclear, quantificational and contextual) are inherited via the semantic heads. In this way, semantic heads play a full role in the organization of the grammar.
6.2.3 Semantic Heads and Generation

The role of semantic heads is clear in semantic head-driven generation, which requires the identity of logical forms between phrases and their semantic heads. Though logical form is not a separate level in HPSG, we could use a logical form consisting of CONTENT, QSTORE and CONTEXT. These features are all needed to include sufficient information for generation.

In order to achieve the required identity of logical forms between phrases and semantic heads, we need to combine lexicalization of quantifier scoping and lexicalization of context, so that SHIP ensures identity of CONTENT, QUIP ensures identity of QSTORE, and CHIP ensures identity of CONTEXT, as shown in (6.10).

\[
\text{SHIP + QUIP + CHIP:}
\]

\[
\text{hd-nexus-ph \Rightarrow CONTENT / \square \quad CONTEXT / \square \quad QSTORE / \square} \quad \left[ \begin{array}{c}
\text{CONTENT / \square} \\
\text{CONTEXT / \square} \\
\text{QSTORE / \square}
\end{array} \right]
\]

\[
\text{hd-adjunct-ph \Rightarrow CONTENT / \square \quad CONTEXT / \square \quad QSTORE / \square} \quad \left[ \begin{array}{c}
\text{CONTENT / \square} \\
\text{CONTEXT / \square} \\
\text{QSTORE / \square}
\end{array} \right]
\]

We will investigate in Section 6.4 how to implement semantic head-driven generation using such a logical form, as part of a first attempt to implement the lexicalization of context. This will require dealing with the problems of quantifier scoping and background conditions which were raised in Section 6.2.1 and Section 6.2.2. First, however, we will review the way in which these problems were treated in the two previous chapters.

In Chapters 4 and 5 we did not discuss the mechanisms for handling quantifiers or contextual information. Now that we have distinguished phrasal and lexical amalgamation (Section 6.1), and described the difficulties in handling quantifier scoping (Section 6.2.1) and contextual background features (Section 6.2.2) in a semantic head-driven framework, we can review the mechanisms used in the earlier chapters for quantifiers and context. In particular, we can now see why the system in Chapter 4 can perform generation successfully whereas the system in Chapter 5 can only parse and cannot generate.

The two systems differ considerably. The system in Chapter 4 does not follow the treatment of quantifiers and context described by Pollard and Sag (1994), but deviates from it significantly in order to perform generation. The system in Chapter 5 does follow Pollard and Sag (1994)
in the treatment of quantifiers and context, while following Sag (1997) in the lexicalized treatment of nonlocal features. As we have seen, the phrasal treatment of quantifiers and context by Pollard and Sag (1994) is basically incompatible with the Semantics Principle, which is essential for semantic head-driven generation. In other words, the reason the system in Chapter 5 cannot generate is, remarkably, that it follows the textbook theory.

The system in Chapter 4 does not attempt to implement storage or retrieval of quantifiers. We saw that in Figure 4.7 the type sign introduces only the attribute SYNSEM, there is no QSTORE attribute. Quantifiers are located directly in QUANTS, which is part of CONTENT. Similarly, the system does not attempt to handle contextual background. We also saw that in Figure 4.7 the type conx introduces only the attribute CINDS, there is no BACKGR attribute. Background conditions are put in RESTR which is part of CONTENT. When an argument fills a thematic role the argument’s entire semantic content, including RESTR, is assigned directly within the head’s content, not just the INDEX as in the textbook. This crucially makes available all the quantifier and background information required for generation.

We also saw that the ID schemata in Chapter 4 include procedural attachments for the Principle of Contextual Consistency and for the Nonlocal Feature Principle but not for the Quantifier Inheritance Principle. As there is no BACKGR attribute in this implementation, the Principle of Contextual Consistency merely unifies the CINDS values. Since no phrasal amalgamation is necessary, either of background conditions or of unscoped quantifiers, there is no difficulty for semantic head-driven generation in the system of Chapter 4.

By contrast, the system described in Chapter 5 does perform phrasal amalgamation, following the theory of Pollard and Sag (1994) closely. In Figure 5.4, the type sign introduces the QSTORE attribute as well as SYNSEM. Quantifiers are located in QSTORE, which is not part of CONTENT. In the semantic sort hierarchy, the type conx introduces BACKGR as well as CINDS. Background conditions are located in BACKGR, which is not part of CONTENT. Whereas in the semantic sort hierarchy in Chapter 4 shown in Figure 4.7, the attributes AGENT, EXPER and THEME are declared as type nomobj and therefore include RESTR as well as INDEX, in Chapter 5 they are declared as type mref (referential index) and therefore include only INDEX and not RESTR. In these respects the system of Chapter 5 follows Pollard and Sag (1994) faithfully.

The ID schemata in Chapter 5 have procedural attachments for the Principle of Contextual Consistency and for the Quantifier Inheritance Principle (but not for the Nonlocal Feature Principle which is lexicalized), as shown in Figures 5.9 to 5.15. The procedure for the Principle of Contextual Consistency performs phrasal amalgamation of background conditions, and the procedure for the Quantifier Inheritance Principle performs phrasal amalgamation of unscoped quantifiers. Both of these involve the difficulties for semantic head-driven generation which we have discussed in this chapter. Therefore, as we already noted, it is precisely because the system in Chapter 5 follows the textbook theory closely, that it can only parse and cannot generate.

However, there is also a more specific technical problem. Both SAX and SGX require procedural attachments to be deterministic. In the case of parsing, the procedures for phrasal amalgamation perform set union, which is deterministic. In the case of generation the equivalent operation is set
partition, which is non-deterministic. Therefore the procedural attachments work with SAX, but fail with SGX. This is a further reason why the system described in Chapter 5 can parse with SAX but cannot generate with SGX.

6.3 Register Variation

To introduce the lexicalization of context in linguistic analysis, we now briefly sketch a simple approach to register variation. We combine the lexicalization of context with the lexicalization of non-local features.

To keep a representation for register variation as simple as possible, we assume that CONTEXT has an additional attribute REGISTER (REGSTR), with value of sort register, which has only two subsorts, formal and informal. To maintain the lexical amalgamation of context, we introduce a Register Amalgamation Constraint (6.11).

(6.11) Register Amalgamation Constraint:

\[ \text{word } \Rightarrow \begin{cases} \text{ARG-ST} \left( \left(\text{REGSTR} \right), \ldots, \left(\text{REGSTR} \right) \right) \\ \text{REGSTR} \end{cases} \]

The combination of the Register Amalgamation Constraint and the Contextual Head Inheritance Principle (6.8) ensures that a phrase inherits the REGSTR values of its daughters via the contextual head daughter. In the same style as (6.6) for C-INDICES, this representation does not cater for register-switching, but assumes that register will typically be uniform throughout an utterance.

6.3.1 Relative Pronouns

In Chapter 5 we described an implementation of the proposals made by Sag (1997) for a set of revisions to HPSG theory, including a new analysis of English relative clauses. In his revised analysis, Sag argues for treating relative that as a pronominal, rather than a complementizer, as the only real obstacle is that it disallows pied piping (6.12a), and this property is shared with relative who (6.12b) in many varieties of English.

(6.12)

a. *The person [with that we were talking] . . .

b. *The person [with who we were talking] . . .

c. The person [with whom we were talking] . . .

Observing that in such varieties the only pied-piped relative pronouns are whose, which and whom (6.12c), Sag comments that “the constraints on this variation have to do with case assignment, register restrictions, or both”. We will look at two ways of specifying such constraints, first as clausal constraints, and then as lexical constraints.

We begin by noting that, while whom (6.13) is always accusative and formal, there appears to be systematic covariation of case assignment and register restrictions in who and that. Relative who is either nominative and unrestricted (REGSTR value register) as in (6.14) or accusative and informal as in (6.15). The same covariation seems to occur in relative that, and also in interrogative who.
(6.13) whom (accusative):

\[
\begin{array}{ll}
\text{PHON} & \langle \text{whom} \rangle \\
\text{CAT} & \text{NP}[\text{acc}] \\
\text{REGSTR} & \text{formal}
\end{array}
\]

(6.14) who (nominative):

\[
\begin{array}{ll}
\text{PHON} & \langle \text{who} \rangle \\
\text{CAT} & \text{NP}[\text{nom}] \\
\text{REGSTR} & \text{register}
\end{array}
\]

(6.15) who (accusative):

\[
\begin{array}{ll}
\text{PHON} & \langle \text{who} \rangle \\
\text{CAT} & \text{NP}[\text{acc}] \\
\text{REGSTR} & \text{informal}
\end{array}
\]

### 6.3.2 Clausal Constraints

We recall from Chapter 5 that nonsubject relative clauses such as *whose bagels I like* and *from whom I bought these bagels* are treated by Sag (1997) in terms of a single construction type `fin-wh-fill-rel-cl`. He hints that the constraints on this type have more work to do than the simplified formulation shown in (6.16), which states only that the filler daughter must be an NP or a PP.

(6.16) `fin-wh-fill-rel-cl ≜ [FILLER-DTR [HEAD noun ∨ prep]]`

We can associate register restrictions with construction types. For example, a relative clause with a PP filler such as (6.12c) is formal. We could specify this by splitting the construction type (6.16) into two distinct subtypes `fin-wh-np-fill-rel-cl` (6.17) and `fin-wh-pp-fill-rel-cl` (6.18).

(6.17) `fin-wh-np-fill-rel-cl ≜ [FILLER-DTR [HEAD noun]]`

(6.18) `fin-wh-pp-fill-rel-cl ≜ [FILLER-DTR [HEAD prep ∨ REGSTR formal]]`

Now we can describe the constraints in (6.12). In (6.12a) and (6.12b), *with* assigns accusative case to its arguments *who* and *that*. Given that accusative *who* is informal register in (6.15), the Register Amalgamation Constraint unifies its informal register with that of *with*. The value *informal* is then passed up to the PP by CHIP (6.8). The formal register restriction in (6.18) would then prevent the informal PP from being the filler of a relative clause. The clausal constraint would in this way block examples (6.12a) and (6.12b) but allow (6.12c) as required.
6.3.3 Lexical Constraints

It could be argued that the PP with who in (6.12b) is both formal and informal. That is, it violates some constraints on case and register consistency. These constraints should apply at the level of the PP, not only at the higher level of the relative clause in (6.18).

This could be done by specifying PP construction subtypes, putting the register restrictions on them instead of on relative clause subtypes. However, we will explore the lexicalization of context and the lexicalization of nonlocal features to show that the same result can be produced by lexical constraints. We specify systematic covariation between register and nonlocal features of prepositions. These covariances are stated in (6.19)–(6.21) as constraints on lexical subtypes.

\[
\text{(6.19)} \quad \text{rel-prep} \Rightarrow \begin{bmatrix}
\text{HEAD} & \text{prep} \\
\text{QUE} & \{\} \\
\text{REL} & \{\square\} \\
\text{SLASH} & \{\} \\
\text{REGSTR} & \text{formal}
\end{bmatrix}
\]

\[
\text{(6.20)} \quad \text{que-prep} \Rightarrow \begin{bmatrix}
\text{HEAD} & \text{prep} \\
\text{QUE} & \{\square\} \\
\text{REL} & \{\} \\
\text{SLASH} & \{\} \\
\text{REGSTR} & \text{formal}
\end{bmatrix}
\]

\[
\text{(6.21)} \quad \text{slash-prep} \Rightarrow \begin{bmatrix}
\text{HEAD} & \text{prep} \\
\text{QUE} & \{\} \\
\text{REL} & \{\} \\
\text{SLASH} & \{\square\} \\
\text{REGSTR} & \text{informal}
\end{bmatrix}
\]

Lexical constraint (6.19) requires prepositions with non-empty REL to have formal register. Similarly, (6.20) requires prepositions with non-empty QUE to have formal register. By contrast, (6.21) requires prepositions with non-empty SLASH to have informal register. Prepositions whose nonlocal features are all empty have no register restriction.

The point of (6.19) is that it requires a rel-prep preposition to take as argument a relative pronoun which is formal. The non-empty REL requires the argument to be a relative pronoun, as its REL value is acquired by lexical amalgamation of nonlocal features. The formal register requires the argument also to be formal, as its REGSTR value is acquired by lexical amalgamation of context.
6.3.4 Interaction of Constraints

We can see in Figures 6.2 and 6.3 how the lexical constraints, interacting with the lexicalization of nonlocal features and the lexicalization of context, provide an alternative way to block \textit{with who} at PP level, while allowing \textit{with whom}, as required.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{interaction_of_constraints}
\caption{Interaction of Constraints in \textit{with whom}}
\end{figure}

In Figure 6.2, \textit{with} assigns accusative case to its argument \textit{whom}, which is lexically specified as accusative in (6.13) anyway. The SLASH Amalgamation Constraint requires \textit{with} to amalgamate the non-empty REL of \textit{whom} in its own REL. As the preposition has thereby a non-empty REL, constraint (6.19) requires it to have formal register. As \textit{whom} is lexically specified as formal register in (6.13), the Register Amalgamation Constraint simply requires the two formal registers to be unified.
In Figure 6.3, *with* also assigns accusative case to its argument *who*. Lexical constraint (6.15) thereby requires accusative *who* to have \( \text{in\text{-}formal} \) register. As in Figure 6.2, the SLASH Amalgamation Constraint requires *with* to amalgamate the non-empty REL of *who* in its own REL, and as the preposition has thereby a non-empty REL, constraint (6.19) requires it to have \( \text{formal} \) register. The Register Amalgamation Constraint is therefore violated, as the informal register of accusative *who* cannot be unified with the formal register of *rel-prep with.*
6.4 Implementing Lexicalization of Context

We now describe a first implementation of the theoretical proposals for the lexicalization of context. The aim of this implementation is to be as simple and clear as possible, largely ignoring questions of efficiency.

We are concerned primarily with head-driven generation, which we take to be the most natural approach to surface realization with HPSG. In Section 6.4.1 we recall the family of head-driven generation algorithms, specifically BUG, SHD and CSHD, which we described in Chapter 3. These algorithms have been presented as Prolog algorithms for use with DCG grammars. In Section 6.4.2 we briefly summarize our method of using ProFit to compile an HPSG grammar, implemented as a PSG with typed feature structures, into a DCG with normal Prolog terms. The point is that this approach enables an HPSG grammar to be used with the existing Prolog algorithms for head-driven generators.

Such a combination of head-driven grammar and head-driven generator works well, provided the semantics is strictly head-driven. However, we show in Section 6.4.3 that if we implement quantifier storage and contextual background conditions as described by Pollard and Sag (1994), the notion of semantic head becomes unclear and this approach no longer works. In fact, head-driven generation of even simple phrases such as “Kim walks” (Chapter 1 of the HPSG textbook) raises fundamental difficulties.

To use a semantic head-driven algorithm, we must adopt the recent HPSG proposals to put quantifier scoping and contextual background inside semantic heads. We have described these proposals in Section 6.2. In Section 6.4.4 we show how they can be implemented in the ProFit HPSG grammar. We conclude that head-driven generation with HPSG is possible, but there are some difficulties in implementing this approach.

6.4.1 The BUG Bottom-Up Generator

We assume that generation starts from some kind of logical form, and that for HPSG the logical form may be represented as a typed feature structure. In fact, logical form is not recognised as a separate linguistic level by HPSG, but is more or less equated with semantic content. We therefore assume, for the purposes of this section, that the starting logical form for generation will be a semantic feature structure which will be identical to the CONTENT feature of the top-level HPSG sign to be generated.

We now briefly summarise the description of semantic head-driven generation algorithms which was presented in Chapter 3. Head-driven generation algorithms assume that most grammar rules have a semantic head daughter whose logical form is identical to the logical form of the mother. The basic head-driven bottom-up generation (BUG) algorithm of van Noord (1990) requires every rule to have such a head (except lexical entries). The semantic head-driven (SHD) algorithm of Shieber et al. (1990) relaxes this, dividing rules into chain rules with such a head which are processed bottom-up, and non-chain rules which are processed top-down. If all rules have a semantic head, SHD becomes equivalent to BUG. The chart-based semantic head-driven (CSHD) algorithm of Haruno et al. (1996) increases the efficiency of the SHD algorithm by using a chart to eliminate
recomputation of partial results.

To make a first implementation of the lexicalization of context, we prefer the simplest possible algorithm. We will therefore use the most basic form of the Semantics Principle, to ensure that all rules have a semantic head. We can then use the simplest algorithm, van Noord's Head-Driven Bottom-Up Generator, BUG.

The BUG algorithm was presented by van Noord (1990) as a Prolog generator, BUG1. We described BUG1 in Chapter 3, in Section 3.1.1, where we showed the Prolog algorithm in Figure 3.1. That form of the algorithm was suitable for use with a Prolog DCG-style grammar, as shown in Figure 3.2. In order to use BUG1 with an HPSG grammar, we will use ProFIT.

6.4.2 BUG with HPSG and ProFIT

We prefer to make the grammar as simple as possible. Figure 6.4 shows the style of grammar we will use. ProFIT templates are used for principles such as the Head Feature Principle (‘HFP’) and Semantics Principle (‘SemP’). We also use templates for constraints on phrase types, as we did in Chapter 5 to implement the phrase type hierarchy of Sag (1997). Note that the hd_nexus_ph template invokes general constraints on headed phrases by @hd_ph, and invokes the non-adjunct Semantics Principle by @’SemP’.

```
‘HFP’ := synsem!loc!cat!head!HF &
       hd_dtr!synsem!loc!cat!head!HF.
‘SemP’ := synsem!loc!cont!Cont &
       hd_dtr!synsem!loc!cont!Cont.
‘SemP’(adjunct) := synsem!loc!cont!Cont &
                 adj_dtr!synsem!loc!cont!Cont.

hd_ph := <hd_ph & @’HFP’ &
         synsem!loc!cat!val!comps[]>

hd_nexus_ph := <hd_nexus_ph & @hd_ph &
               @’SemP’>.

hd_subj_ph := <hd_subj_ph & @hd_nexus_ph &
              @’VALP’(spr) & @’VALP’(comps) &
              synsem!loc!cat!val!subj[]>

hd_comp_ph := <hd_comp_ph & @hd_nexus_ph &
              @’VALP’(subj) & @’VALP’(spr)>

@hd_subj_ph & phon!P0-PH &
@hd_dtr!(Head &
           synsem!loc!cat!val!subj[S]) &
@subj_dtr!(Subj & synsem!S)
--- [Head & <phrase & phon!P1-PN],
     Subj & <phrase & phon!P0-P1].

@hd_comp_ph & phon!P0-PH &
@hd_dtr!(Head &
           synsem!loc!cat!val!comps[C]) &
@comp_dtrs![Comp & synsem!C]
--- [Head & <word & phon!P0-P1],
     Comp & <phrase & phon!P1-PH].
```

Figure 6.4: Principles, Phrase Types, Schemata
Again, immediate dominance schemata are implemented as PSG rules, using the schematic categories *word* and *phrase*, rather than traditional PSG categories (NP, VP etc). However, there is an important difference here from the previous implementations which were used with SAX and SGX: for use with van Noord's BUG1 generator, the semantic head must always be first in the list of daughters. Linear precedence is therefore specified by the order of combination of PHON strings, implemented as Prolog difference lists. Some example Head-Subject and Head-Complements Schemata are shown in Figure 6.4.

We use a slightly modified form of the BUG1 algorithm, shown in Figure 6.5. There are two differences from the algorithm of van Noord (1990), which was shown in Chapter 3 (Figure 3.1).

```prolog
bug1(Node) :-
    predict_word(Node, Small),
    connect(Small, Node).

connect(Node, Node).
connect(Small, Big) :-
    predict_rule(Small, Middle, Others, Big),
    connect(Middle, Big),
    gen_ds(Others).

gen_ds([]).
gen_ds([Node|Nodes]) :-
    bug1(Node),
    gen_ds(Nodes).

predict_word(Node, Word) :-
    semantics(Node, Sem),
    semantics(Word, Sem),
    lex(Word).

predict_rule(Head, Mother, Others, _) :-
    ( Mother --- [Head|Others] ).

generate(Sem, Sign, String) :-
    semantics(Sign, Sem),
    saturated(Sign),
    bug1(Sign),
    string(Sign, String).
```

Figure 6.5: The modified BUG1 algorithm

First, in order to use BUG1 with HPSG, we provide an interface between the Prolog algorithm and the HPSG grammar and lexicon which are implemented in ProFIT. We define two interface procedures `string/2` and `semantics/2` which extract the string and the semantics respectively from a given HPSG sign. These interface procedures insulate the generation algorithm from the internal details of the feature structures used in any particular grammar. The definition of `predict_word` avoids reference to the internal structure of lexical entries by using the `semantics` interface procedure.

Second, the clauses in the `connect` predicate are re-ordered, as discussed by van Noord (1993) p. 94. This variant of the algorithm, in which the generation of the daughters of each chain rule is only performed once the head has been connected upward, was proposed by Russell et al. (1990).
6.4.3 Quantifiers and Context

This simple form of head-driven generation works fine, provided the semantics is head-driven. Basically, that means that all necessary information must be contained inside the CONTENT feature, and cannot be distributed in other features (such as QSTORE, BACKGR). Moreover, when an NP is assigned to the semantic role of a verb, the whole of the NP’s CONTENT must be assigned to the role (not only its INDEX).

However, this is significantly different from the semantics of Pollard and Sag (1994). In BUG1, predictword requires unification of the top-level logical form with the logical form of some lexical entry. By equating logical form (as starting point for generation) only with semantic content (as in HPSG’s CONTENT feature), we have made a major simplification.

To start with, there is a complication in the HPSG theory of Pollard and Sag (1994) caused by the use of Cooper storage to handle quantifier scope ambiguities. Although scoped quantifiers are included in the QUANTS list within CONTENT, unscoped quantifiers are stored in the QSTORE set outside CONTENT. This means that logical form cannot be equated only with CONTENT, but needs to include QSTORE information as well.

In this approach, a quantifier may be retrieved from storage at any suitable syntactic node. A quantifier retrieved at a particular node is a member of the QSTORE set (but not the QUANTS list) of a daughter of that node, and due to the retrieval it is a member of the QUANTS list (but not the QSTORE set) of the mother node. Pollard and Sag (1994) define a modified Semantics Principle to cater for this, but the effect of retrieval on QSTORE and QUANTS means that the mother and the semantic head daughter must have different logical forms. Although the daughter is the semantic head by the HPSG definition, it is not a semantic head as defined by the generation algorithm.

In addition to semantic content, natural language generation requires presuppositions and other pragmatic and discourse factors. In HPSG, such factors are part of CONTEXT. To specify these factors for generation, the usual approach is to include them in the logical form. So logical form needs to include CONTEXT as well as CONTENT and QSTORE. This extended logical form can be specified for BUG1 by extending the semantics interface procedure to include CONTEXT and QSTORE as well as CONTENT, as shown in Figure 6.6.

```plaintext
semantics( synsem!loc!(cnt!Ct & conx!Cx & qstore!Q0-QN),
               ct!Ct & cx!Cx & qs!Q0-QN).
```

Figure 6.6: Extending the Logical Form

However, semantic head-driven generation simply does not work with this inclusive logical form, given the theory of Pollard and Sag (1994). Even if we ignore quantifier retrieval and look at a very simple sentence, there is a fundamental difficulty with CONTEXT.

Figure 6.1 showed the standard HPSG analysis of “She saw Kim”. We recall that in HPSG, “she” has a non-empty BACKGR set in CONTEXT, stating a pragmatic requirement that the referent is female. This background condition is passed up from NP to S by the Principle of
Contextual Consistency. Similarly, “Kim” has a background condition that the referent bears this particular name. This condition is also passed from NP to VP, and from VP to S.

It is clear in Figure 6.1 that S, VP and V all share the same CONTENT (shown by tag \[1\]). If logical form is restricted to semantic content, then V is the semantic head of VP and VP is the semantic head of S, not only in terms of the HPSG definition but also in terms of the BUG algorithm. In this case, “saw” can be found immediately by predict_word in BUG1. But if we extend logical form to include context factors represented by BACKGR, as required for adequate realization, it is clear from Figure 6.1 that S does not have the same logical form as VP, and VP does not have the same logical form as V, as their BACKGR sets differ. Therefore, although V is still the semantic head of VP according to the HPSG definition, it is not the semantic head according to the BUG algorithm. Similarly, VP is still the semantic head of S for HPSG, but it is not the semantic head for BUG. In this case, predict_word cannot find any semantic head word in the lexicon, and BUG1 cannot generate the sentence.

### 6.4.4 Lexical Amalgamation in ProFIT

Extending logical form to include unscoped quantifiers and background conditions seems to be necessary for adequate generation, but it makes the notion of semantic head unclear. In fact there are two different definitions of “semantic head”: the HPSG definition based on adjunct daughter or syntactic head daughter, and the generation algorithm definition based on identity of logical forms. Fortunately, the recent proposals for changes in the HPSG account of quantifier scoping and contextual features suggest an approach in which the two notions of semantic head can be brought back together.

We recall that in Pollard and Sag (1994), both QSTORE sets and BACKGR sets are phrasally amalgamated. The Quantifier Inheritance Principle requires a phrase’s QSTORE to be the set union of the QSTOREs of all the daughters, minus any quantifiers in the phrase’s RETRIEVED list. Similarly, the Principle of Contextual Consistency requires a phrase’s BACKGR to be the set union of the BACKGR sets of all the daughters. Instead of this phrasal amalgamation, we will adopt the recent proposals that these sets should be lexically amalgamated.

Lexicalization of quantifier storage, proposed by Pollard and Yoo (1995), takes QSTORE to be a LOCAL feature which can be included in the features subcategorized for by a lexical head, and can therefore be lexically amalgamated in the head. A phrase no longer inherits unscoped quantifiers directly from all daughters, instead they are inherited indirectly via the semantic head daughter.

Lexicalization of context, proposed by Wilcock (1997) and described in Section 6.1.4, takes the same approach. As CONTEXT is a local feature, it can be subcategorized for by a head word and lexically amalgamated in the head by means of a BACKGR amalgamation constraint. Instead of a phrase inheriting BACKGR conditions directly from all daughters by the Principle of Contextual Consistency, they are inherited indirectly via the “contextual head” daughter which is the same as the semantic head daughter.

In the ProFIT implementation, QSTORE sets and BACKGR sets are Prolog difference lists. We therefore implement lexical amalgamation of both sets as shown in Figure 6.7, the lexical entry
for the verb "saw". The subject’s BACKGR set B0-B1 and the object’s BACKGR set B1-BN are amalgamated in the verb’s BACKGR set B0-BN. The subject and object QSTORE sets, Q0-Q1 and Q1-QN, are similarly amalgamated in the verb’s QSTORE Q0-QN.

The basic Semantics Principle, for semantic content only, was implemented by the ProFIT templates 'SemP' and 'SemP'(adjunct) as shown in Figure 6.4. In order to include unscoped quantifiers and background conditions in logical form, and still make it possible for the logical form of a phrase to be identical to the logical form of its semantic head, the Semantics Principle needs to be replaced and extended.

As proposed by Wilcock (1997), we will combine three principles as described in Section 6.2.3: the Semantic Head Inheritance Principle (SHIP), the Quantifier Inheritance Principle (QUIP), and the Contextual Head Inheritance Principle (CHIP). These are implemented by templates as shown in Figure 6.8 (only the non-adjunct forms are shown). To include the three principles in the grammar, the template for \texttt{hd\_nexus\_ph} in Figure 6.4 is extended as shown in Figure 6.8.

\begin{verbatim}
'SHIP' := synsem!loc!cont!Cont &
hd_dtr!synsem!loc!cont!Cont.
'QUIP' := synsem!loc!qstore!QS &
hd_dtr!synsem!loc!qstore!QS.
'CHIP' := synsem!loc!conx!Conx &
hd_dtr!synsem!loc!conx!Conx.

hd\_nexus\_ph := ⟨hd\_nexus\_ph & hd\_ph &
0'SHIP' & 0'QUIP' & 0'CHIP'⟩.
\end{verbatim}

Figure 6.8: Inheritance of Logical Form in ProFIT

With these revisions, it is possible to include unscoped quantifiers and background conditions in the starting logical form, and perform head-driven generation successfully using the BUGI generator. However, there remain various technical difficulties in this implementation. The ProFIT system does not support either dynamic constraint checking or set-valued features. The methods shown (template expansion and difference lists) are only partial substitutes for the required facil-
ities. Unfortunately, a full implementation of the CI-ONE extensions of ProFIT to support set constraints (Manandhar, 1994) was not available.

One problem caused by the use of difference lists to represent unordered sets is that the specific order of elements in QSTORE or BACKGR (which should be unordered) can interfere with generation. This can be seen in the examples in the demonstration of head-driven generation with lexicalization of context in Appendix C.

Care must be taken in implementing adjuncts with lexicalization of context and quantifier storage. Following Pollard and Sag (1994), the adjunct is the semantic head and must therefore include the entire semantics of the modified sign within its own semantics. An example is shown in Figure 6.9.

```
adj ective(Re ln) := @word &
    phon!(Re ln[P]-P &
    synsem!loc!(cat!(head!(<adj &
        mod!(Nbar &
            loc!(cont!(NCont & index!I &
                restr!NbarRestr) &
                conx!backgr!BC-EN &
                qstore![det!Det & restind!NCont|QC]-QN)) &
                val!(subj[] &
                    spr[] &
                    comps[])) &
                cont!(AdjCont & index!I &
                    restr![reln!Reln & inst!I|NbarRestr]) &
                conx!backgr!BC-EN &
                qstore![det!Det & restind!AdjCont|QC]-QN)).
```

```
lex( adjective(big) ).
lex( adjective(little) ).
lex( adjective(good) ).
lex( adjective(bad) ).
```

Figure 6.9: Lexical Template for Modifying Adjectives

This shows the lexical template for modifying adjectives. The SYNSEM value of the modified sign (an Nbar) is accessible through the adjective’s MOD attribute. The adjective’s RESTR is the concatenation [reln!Reln & inst!I|NbarRestr] of its own restrictive soa, [reln!Reln & inst!I] with the RESTR of the Nbar, NbarRestr. The adjective’s BACKGR is simply unified with the Nbar’s BACKGR, as the adjective makes no contribution to the background conditions. However, the adjective’s QSTORE cannot just be unified with the Nbar’s QSTORE, because the RESTIND (restricted index) within the quantifier in the Nbar’s QSTORE does not include the adjective’s contribution to RESTR. The RESTIND in the quantifier in the adjective’s QSTORE needs to be unified with the adjective’s CONTENT, AdjCont, which does include the adjective’s contribution as well as the RESTR of the Nbar.

Some examples of head-driven generation of sentences with modifying adjectives are included in the demonstration listing in Appendix C.

119
6.5 Summary

In this chapter we have proposed the lexicalization of context in HPSG, giving three motivations. On a theoretical level, it naturally follows other recent revisions in HPSG theory concerned with set-valued features, bringing contextual features into line with the lexicalization of nonlocal features and the lexicalization of quantifier scoping. Combining lexicalization of context and lexicalization of quantifier scoping restores and enhances the role of semantic heads in head-driven grammar (which is significant on a computational level for semantic head-driven generation). Finally, the lexicalization of context may be exploited in the development of constraint-based lexicalist approaches in linguistic analysis, as suggested in a brief sketch of lexical constraints on case assignment and register variation in English relative clauses.

We also described a first implementation of the lexicalization of context. We showed how van Noord’s BUG1 generator can easily be adapted for use with an HPSG grammar implemented in ProFIT. Although this works well if the semantics is strictly head-driven, we found that if we implement the full HPSG textbook semantics, with quantifier storage and contextual background conditions, the notion of semantic head becomes unclear. Surprisingly, this natural approach does not work, even for simple examples. In order to use semantic head-driven generation algorithms with HPSG, we must adopt recent theoretical revisions to include quantifier storage and contextual background inside semantic heads. We showed how the HPSG grammar in ProFIT can be extended with these revisions, although there are technical difficulties in implementing set constraints.
Chapter 7

Towards Incremental Generation:
Minimal Recursion Semantics

In Chapter 6 we discussed the important role which semantic heads ought to play in the organization of a head-driven grammar such as HPSG. We noted that the reformulation of the Semantics Principle in Chapter 8 of Pollard and Sag (1994) reduces the significance of semantic heads. We argued that the lexicalization of quantifier scoping and the lexicalization of context allow the original Semantics Principle to be reinstated, and even allow the significance of semantic heads to be enhanced.

There are, however, counterarguments against strengthening the role of semantic heads. One is concerned with the needs of translation, an important application for linguistic theories. Another is the recent trend in HPSG towards the ideas of Construction Grammar. Both factors tend to move away from the direction proposed in Chapter 6, and as we noted in Chapter 5 (Section 5.2) both have been important since 1994 in influencing the development of Minimal Recursion Semantics (Copestake et al., 1997).

In this chapter we therefore turn away from semantic head-driven generation, which has been the principal theme of the thesis, to take a relatively brief look at non-head-driven generation. We discuss approaches to generation intended for use in machine translation. We describe how Minimal Recursion Semantics can be implemented in ProFIT. We show how an approach to bag generation with categorial grammar and indexed logical forms can be adapted for generation with HPSG, by using MRS relations as indexed logical forms.

Another factor in favour of this non-head-driven approach is the need to support incremental generation for highly interactive dialogues. We describe how the bag generation algorithm can be adapted to implement incremental generation with HPSG and MRS. We finally discuss the need for an effective implementation of information structure within HPSG, to support elliptical generation. These issues, however, require future work.
7.1 Machine Translation

Machine translation lies outside the scope of this thesis. Nevertheless the requirements of translation have strongly influenced work on non-head-driven generation. For example, a central issue of machine translation research has been how to handle “head-switching” between languages, when the syntactic or semantic head of a source language structure does not naturally transfer to the head of a translationally equivalent structure in the target language. In these cases, which are frequent, a strongly head-driven semantic representation may not be desirable. A “flat” representation, made up of a number of rather simple individual terms concatenated together, tends to be preferred.

There have been various different ideas about what kinds of things should make up such flat representations. Are they logical formulae, or words of a particular natural language, or something else? We will not discuss this question here, as we are concerned only with generation, and the methods required for generation from all such flat representations seem to be similar.

The basic method for non-head-driven generation is to use a chart. Chart-based generation was proposed by Shieber (1988), and the key ideas are reviewed by Kay (1996). In the case of head-driven generation, the basic algorithm is relatively efficient as it is guided by the notion of semantic heads. Nevertheless, using a chart makes it more efficient (Hanno et al., 1996). In the case of non-head-driven generation there is no similar notion, so the basic algorithm is unguided and inefficient. The use of a chart is therefore essential.

7.1.1 Generation from Logical Forms

A chart generation algorithm specifically designed for use in machine translation was developed at UMIST by Phillips (1991), published as Phillips (1993). The published algorithm is for use with categorial grammar, but we will show in Section 7.2 how it can be used with HPSG. The algorithm generates from a flat list of terms, which Phillips takes to be logical formulae.

A fundamental problem for generation is logical form equivalence, discussed by Shieber (1993). In parsing, a grammar may derive a certain logical form from a surface string, and if the grammar is reversible it will be able to generate the string from the same logical form. But an application may supply a different yet equivalent logical form, from which the grammar cannot necessarily generate the string. The difference may be simply in the order of items in the logical form.

In machine translation, such differences in the order of items in a logical form are very frequent. It is therefore an important property of Phillips’ algorithm that the order of the terms in the list makes no difference to the generation process.

Although Phillips’ algorithm was developed for machine translation, it was adopted for surface generation with categorial grammar in the PLUS dialogue system at UMIST (Black and Cunningham, 1992). Based on the experience of its use in a dialogue framework, Lager and Black (1994) suggested that the approach could be adapted for incremental generation. We take up this suggestion in Section 7.3, where we show how the algorithm can be adapted for incremental generation with HPSG and MRS.
7.1.2 Generation from Lexical Signs

In Shake and Bake machine translation, which was proposed by Whitelock (1991), published as Whitelock (1994), and by Beaven (1992) and Whitelock (1992), translation does not involve logical forms, but is based on direct mappings between lexical items of the languages. Generation starts from a flat list, but the items in the list are lexical items of the target natural language, not logical forms.

Nevertheless, Whitelock's and Phillips' approaches are basically similar, as both approaches depend crucially on the use of indices to obtain the correct result. In Phillips' algorithm, the logical forms are used to find the required target language lexical items in a monolingual lexicon. In Whitelock's method, source language lexical items are used to find the required target language lexical items from a bilingual lexicon. In both cases generation has the task of arranging the unordered bag of target language lexical items into the correct surface order, using the grammar of the target language. The possible results are constrained by the indices, which are instantiated from the source language analysis.

Like Phillips, Whitelock (1991) proposed his method for use with categorial grammar. He sketched a simple shift-reduce bag generator which was not intended to be efficient. Brew (1992) proposed ways to improve the efficiency of generation. Subsequently, influenced by the ACQUILEX project (Sanfilippo et al., 1992; Copestake and Sanfilippo, 1993), the Shake and Bake approach was adapted for use with HPSG. An implementation was developed in ALE (Sanfilippo et al., 1994), and more efficient generation techniques were developed by Poznanski et al. (1995).


7.1.3 Generation from Minimal Recursion Semantics

When HPSG was developed by Pollard and Sag (1987) at Ohio and CSLI Stanford, it was linked with the work on Situation Semantics by Barwise and Perry (1983) which was active at CSLI at the time. Situation Semantics subsequently continued to be associated with HPSG as the underlying semantic theory up to the publication of Pollard and Sag (1994).

Following the work of Kay et al. (1994), CSLI became a participant in the Verb mobil face-to-face dialogue translation project (Wahlster, 1993). Semantic transfer in Verb mobil (Copestake, 1995) was influenced by the Shake and Bake MT work and the ACQUILEX project mentioned in Section 7.1.2. In order to facilitate the use of HPSG in machine translation, and specifically in Verb mobil, Minimal Recursion Semantics (MRS) was developed at CSLI by Copestake et al. (1995) as a new semantic representation for HPSG.

Like the earlier approaches of Phillips (1991) with logical forms and Whitelock (1991) with lexical signs, in MRS the underlying semantics is neo-Davidsonian, with explicit event variables associated with verbs. The specific instantiations of these variables, and the instantiations of the
semantic indices associated with nouns, play a crucial role in constraining generation. We describe MRS representations in Section 7.2.2, where MRS is compared with the indexed logical form used by Phillips (1991).

The recent revisions of MRS by Copestake et al. (1997) show the influence of Construction Grammar (Fillmore and Kay, To appear). In particular, MRS now explicitly allows for the possibility that a construction may make its own contribution towards semantic content, in addition to the normal composition of semantic content from its component parts. This possibility is exploited by Riehemann (1996) to give a new account of idiomatic expressions in HPSG. Note that this non-compositional possibility would not necessarily prevent the use of head-driven generation. It only means that not all rules can be chain rules, so a simple algorithm like BUG cannot be used but the SHD algorithm could in principle be used.

However, the crucial point for generation is that MRS is 'minimally recursive' because it is basically a flat list-based representation without nested structures. Instead of building complex semantic representations by unification of nested structures, the basic operation is concatenation of lists. As far as generation is concerned, this means that non-head-driven generation is more suitable for MRS. The concatenation of lists is a form of phrasal amalgamation, like the set union operation on set-valued features which we investigated in Chapter 6. Head-driven generation with list-valued features would require the lexicalization of the list-valued features: they would need to be lexically amalgamated within the semantic head word.

Given the aim of MRS to be useful for translation, and given the trend towards a new role for constructions in HPSG, it is more appropriate to investigate non-head-driven generation with MRS. From the work on Shake and Bake MT, this approach has become known as 'bag' generation. In Section 7.2 we describe how Phillips' algorithm can be adapted for use with HPSG and MRS, by a simple implementation using ProFIT.

7.2 Bag Generation

7.2.1 Phillips' Algorithm

Phillips (1993) proposed a bottom-up chart generation algorithm for use with indexed logical forms and categorial grammar in machine translation. An important property of the algorithm is that the order of terms in the logical form is not significant. Phillips presented the algorithm in Prolog, as shown in Figure 7.1.

We now show how Phillips' algorithm can be used with HPSG grammar, with a bag of MRS relations instead of a bag of indexed logical terms. Though he presents the algorithm as a generator for categorial grammar, Phillips suggests that it can be adapted for use with phrase structure grammar (PSG), provided an indexed logical form is used and the indices are included in the syntactic categories. HPSG uses indices for agreement, and therefore includes the indices inside syntactic categories. By implementing HPSG as a PSG extended with typed feature structures (as described in Chapter 4), and by implementing MRS as a similarly extended indexed logical form, we can adapt Phillips' algorithm for use with HPSG.

The adapted algorithm is shown in Figure 7.2. We use simple chart processing from the bottom-
generate(Syntax, Semantics, Text) :-
    abolish(edge,1),
    foreach((e(Word,Definition,Sem), accept(Sem,Semantics)),
        addEdge(Definition:[Word|R]:R), !,
        edge(Syntax:Text:[])).
addEdge(Category1) :-
    assert(edge(Category1)),
    foreach((edge(Category2), apply(Category2,Category1,Mother)),
        addEdge(Mother)).
apply( Root\Arg:Mid:End, Arg:Start:Mid, Root:Start:End ).
apply( Root/Arg:Start:Mid, Arg:Mid:End, Root:Start:End ).
accept(Semi1+Sem2, Target) :- !,
    accept(Semi1, Target), accept(Sem2, Target).
accept(Word, Target) :- member(Word, Target).
member(X, [X|_]) . member(X, [_|L]) :- member(X, L).
foreach(X, Y) :- X, call((Y, !)), fail. foreach(_, _).
/* Lexicon */
e(john, np#J, jcn(J)).
e(letter, n#L, letter(L)).
e(a, np#N/n#N, indef(N)).
e(wrote, s#S
p#Act np#Pat, write(S)+past(S)+actor(S,Act)+patient(S,Pat)).
e(penned, s#S np#Act np#Pat, write(S)+past(S)+actor(S,Act)+patient(S,Pat)).
/* Test */
letter :-
    generate(s#,
        [letter(1), indef(1),
         write(e), past(e), actor(e,j), patient(e,1), jcn(j)],
        T),
    display(T), nl, fail.

Figure 7.1: Phillips' Generation Algorithm (for Categorial Grammar)
up chart parser of Gazdar and Mellish (1989). The main work is done by `start_gen/1`, which looks up the next term in the list of semantic terms, adds appropriate edges to the chart, and calls itself recursively on the remaining terms. Generation finishes when all edges have been built, and is successful if an inactive edge “spans” the whole input semantics, i.e., if an inactive edge’s semantics are a permutation of the input semantics. `permutation/2` is a library predicate.

generate(Semantics, Category, String) :-
  abolish(edge, 2),
  start_gen(Semantics),
  clause(edge(Category, []), true),
  semantics(Category, EdgeSem),
  permutation(Semantics, EdgeSem),
  string(Category, String).

start_gen([]).

start_gen([Term | Terms]) :-
  foreach(lookup_term(Term, Category),
    add_edge(Category, [])),
  start_gen(Terms).

add_edge(Cat, Cats) :-
  clause(edge(Cat, Cats), true), !.

add_edge(Cat1, []) :-
  asserta(edge(Cat1, [])),
  foreach(rule(Cat2, [Cat1 | Cats]),
    add_edge(Cat2, [Cat1 | Cats])),
  foreach(edge(Cat2, [Cat1 | Cats]),
    add_edge(Cat2, Cats)).

add_edge(Cat1, [Cat2 | Cats]) :-
  asserta(edge(Cat1, [Cat2 | Cats])),
  foreach(edge(Cat2, []),
    add_edge(Cat1, Cats)).

rule(Mother, Daughters) :-
  (Mother ---> Daughters).

Figure 7.2: A Simple Bag Generator (for PSG)

The algorithm assumes that Categories include surface strings and semantics as well as syntactic information (HPSG has Category `sign`, with string in PHON and semantics in CONTENT). For a particular grammar, the predicates `string/2` and `semantics/2` extract the string and semantics from the Category, giving a clean interface between the algorithm and the grammar. For our HPSG grammar implemented in ProFIT, suitable definitions of `string/2` and `semantics/2` are shown in Figure 7.3.

For a particular lexicon, `lookup_term/2` returns the Category of a lexical entry which includes the given semantic predicate, giving an interface between the algorithm and the lexicon. There is a problem, noted by Phillips (1993), of how to generate words which are semantically vacuous, such as pleonastic `it in it seems`, or lexically-specified prepositions such as `on in depend on`. The problem is how, and when, to introduce them into the chart. One approach would be to insert all semantically vacuous words into the chart at the outset, regardless of whether they are actually required or not. We adopt a more delicate approach, adding semantically vacuous words to the chart only when they
are subcategorized for by semantically contentful words which are being added. This is (partially) implemented by the various clauses of `lookup_term/2` shown in Figure 7.3.

```
mrs(MRS) := synsem!loc!cont!MRS.

semantic(phrase(0mrs(liszt!Lisz-[[]]), Liszt)).

string(phrase(phon!Phon-[]), String) :-
  closed_list(Phon, String).

closed_list([], []) :- !.
closed_list([H|Open], [H|Closed]) :-
  closed_list(Open, Closed).

key(Rel) := synsem!loc!cont!liszt![Rel|X]-X.

lookup_term(Rel, word(It)) :-
  dict(_HeadWord, @key(Rel) & synsem!loc!cat!arg_st![0mp(Case,<it)]),
  dict(it, It & @pronoun(Case,<it)).
lookup_term(Rel, word(Prep & 0mrs(index/Tag))) :-
  dict(_HeadWord, @key(Rel) & synsem!loc!cat!arg_st![-, @pp(Pform, Tag)]),
  dict(Pform, Prep & @prep(Pform, prd:n)).
lookup_term(Rel, word(Sign)) :-
  dict(Word, Sign & @key(Rel)),
  empty_sem(Word, word(Sign)).
empty_sem(Word, word(Sign)) :-
  dict(Word, Sign & 0mrs(liszt!L1-L2)),
  L1 == L2.
```

Figure 7.3: Lexicon Interface Procedures

### 7.2.2 Implementing MRS in ProFIT

We now describe how MRS can be implemented in ProFIT. As MRS was motivated by the needs of machine translation, where flat representations are preferred over strongly head-driven representations, it is very similar to the indexed quasi-logical form (QLF) of Phillips (1993). The QLF was implemented straightforwardly in Prolog, as shown in Figure 7.1. Like the QLF, MRS depends on the use of indices to represent dependencies between the terms in the flat list. HPSG previously used indices only for entities of type nominal object, to assign them to semantic roles as participants in states of affairs and to carry agreement features. In MRS, indices are also used for events, as in the QLF.

A major difference between MRS and the QLF is that MRS uses typed feature structures instead of ordinary logical terms. Each element in the list is an HPSG typed feature structure of type relation. This facilitates the integration of MRS into HPSG. While the QLF logical terms could be represented in Prolog, we need ProFIT to extend the terms with typed feature structures for MRS. We thus implement MRS as an extension of QLF in the same way that we implement HPSG as an extension of PSG.
Another major difference, which makes MRS a significant improvement over the QLF, is that MRS supports the representation of quantifier scope (either fully resolved or underspecified). This is done by including handles which label each term in the list. Scope can be represented by means of the handles, while maintaining the flat list representation, without the nesting required when operators are used to represent scope. As a musical joke about semantic composition, the handle feature is named HANDEL and the list feature is named LISZT.

content > [quant, nom_obj, psca]
  intro [handel, index:individual, liszt].
nom_obj > [npro, prcn].
prcn > [ppro, ana].
ana > [ref1, recp].

individual > [event, entity] intro [ref].
entity > [ref, there, it] intro [agr:agr].
agr fin_dom [m,f,n] * [sg,pl] * [1,2,3].

rel > [quant_rel, ncnm_rel, name_rel, verb_rel]
  intro [handel, reln].
quant_rel intro [bv:ref, restr, scope].
ncnm_rel intro [inst:ref].
nname_rel intro [name, named:ref].
verb_rel > [act_rel, act_und_rel, act_rec_und_rel, exp_und_rel]
  intro [event:event].
act_rel intro [act:ref].
act_und_rel intro [act:ref, und:ref].
act_rec_und_rel intro [act:ref, rec:ref, und:ref].
exp_und_rel intro [exp:ref, und:ref].

Figure 7.4: Semantic Sort Hierarchy for MRS

As we are only changing the semantics within the grammar, to an MRS representation, most of the HPSG implementation is unchanged from the one described in Chapter 5. The syntactic sort hierarchy is the same as shown in Figure 5.4, so we do not repeat it here. However, the semantic sort hierarchy is different, and is shown in Figure 7.4. Note that the CONTENT feature introduces the three attributes HANDEL, INDEX and LISZT.

In MRS, the individual semantic terms which make up a LISZT are relations of different types. In this simple implementation, we include quantifiers (quant_rel), common nominal entities (ncnm_rel), names (name_rel) and verbal relations (eventualities: events or states) (verb_rel). The attributes of quantifiers are now BV (bound variable), RESTR and SCOPE. Verbal relations have an EVENT attribute, and are classified along the lines proposed by Davis (1996), with attributes such as ACT (actor) and UND (undergoer).
Figure 7.5: Example Verb Lexical Entry with MRS

Figure 7.5 shows an example lexical entry for transitive verbs with actor and undergoer roles. This is in fact a lexical inference rule, as described in Chapter 4, and can be compared with the similar rule shown in Figure 4.5. Note that the CONTENT feature has handel, index and liszt attributes, and the verb relation in LISZT has handel and event attributes. Lexical entries for some determiners are shown in Figure 7.6. Here the quantifier relation in LISZT has handel, bv and restr attributes.

Figure 7.6: Some Determiner Lexical Entries with MRS
7.2.3 The Semantics Principle

In the semantics of Pollard and Sag (1994), which was derived from Situation Semantics, semantic composition is performed by recursive unification of semantic feature structures, to produce a single complex semantic feature structure in the semantic head. This semantic structure is structure-shared between a phrase and its semantic head daughter by the Semantics Principle. This form of semantic representation is therefore suitable for semantic head-driven generation. The original, strong form of the Semantics Principle is shown by 'SemP' in Figure 7.7, which repeats the details described in Chapter 6.

\[\text{‘HFP’} := \text{synsem!loc!cat!head!HF} \& \\
\text{hd_dtr!synsem!loc!cat!head!HF}.
\]

\[\text{‘SemP’} := \text{synsem!loc!cont!Cont} \& \\
\text{hd_dtr!synsem!loc!cont!Cont}.
\]

\[\text{hd_ph} := <\text{hd_ph} \& \& \text{‘HFP’} \& \\
\text{synsem!loc!cat!val!comps}![].
\]

\[\text{hd_nexus_ph} := <\text{hd_nexus_ph} \& \& \text{‘HFP’} \& \\
\text{synsem!loc!cat!val!subj}![].
\]

\[\text{hd_comp_ph} := <\text{hd_comp_ph} \& \& \text{‘HFP’} \& \\
\text{synsem!loc!cat!val!comps}![].
\]

--- > [Head & <phrase & phon!P1-PN, \\
Subj & <phrase & phon!P0-P1].

\[\text{hd_subject_ph} \& \& \text{‘HFP’} \& \\
\text{hd_dtr!}(\text{Head} \& \\
\text{synsem!loc!cat!val!subj}[S]) \& \\
\text{subj_dtr!}(\text{Subj} \& \text{synsem!S})
\]

\[\text{hd_comp_ph} \& \& \text{‘HFP’} \& \\
\text{hd_dtr!}(\text{Head} \& \\
\text{synsem!loc!cat!val!comps}[C]) \& \\
\text{comp_dtrs!}(\text{Comp} \& \text{synsem!C})
\]

--- > [Head & <word & phon!P0-P1, \\
Comp & <phrase & phon!P1-PN].

Figure 7.7: Strong Semantics Principle

By contrast, in the flat MRS representation, semantic composition is performed by concatenation of the LISZTs of a phrase’s daughters to give the LISZT of the mother. The LISZT of the semantic head daughter will not be the same as the LISZT of the mother. MRS is therefore suitable for bag generation, but not for head-driven generation. (Head-driven generation with MRS would require lexical amalgamation of LISZT).

With this approach, the Semantics Principle must be scrapped or redefined. We show a simple revision of ‘SemP’ in Figure 7.8, in which INDEX and HANDLE (but not LISZT) are shared between mother and semantic head, in contrast to ‘SemP’ in Figure 7.7 which shares the whole CONTENT structure. The reformulation of the Semantics Principle proposed by Copestake et al. (1997) greatly reduces the significance of semantic heads, which we discussed in Chapter 6.
We use Prolog difference lists to implement the LISZT feature for efficient concatenation, as we did previously with PHON. In the case of LISZT, the predicate semantics hides this representation from the algorithm. LISZT concatenation is added to the PSG rules for HPSG schemata, as shown in Figure 7.8.

A full MRS representation includes a top-level handle and a top-level index, which are specified separately from the flat list of terms. A top-level index is also mentioned by Phillips (1993). We have ignored these in the bag generation algorithm, to simplify adaptation to incremental generation in Section 7.3. However, the top-level index specifies what the semantics is about. For example, a simple QLF representation: `[man(m), walk(e,m)]` could mean either “A man walked” or “A man who walked”. These are only distinguished by the top-level index (e or m respectively).

In this respect, the top-level index specifies the topic, part of information structure, which we will discuss in Section 7.4.1.

The simple implementation of MRS shown here does not deal adequately with the handling of quantifier scope, as proposed by Copestake et al. (1997). Similarly, the implementation of bag generation is very simple and does not attempt to introduce techniques to improve efficiency. Nevertheless, this experimental implementation can parse and generate successfully (though we have not described parsing here) and shows that Phillips’ algorithm can be adapted for use with HPSG.
7.3 Incremental Generation

Perhaps one of the reasons HPSG has not been popular for use in generation (as we discussed in the Introduction, Chapter 1) is that it seems difficult to perform incremental generation with HPSG. We have argued that head-driven generation is the natural approach to generation with HPSG, due to the intrinsically head-driven organization of the grammar. However, in some applications such as interactive dialogues or simultaneous interpretation, the ultimate syntactic or semantic head of a communicative unit may not be known initially, but there is an urgent need to get on with processing whatever is available so far. It is simply unacceptable to delay until the head is known.

Some of the advantages of incremental processing for interactive dialogues, both in terms of incremental interpretation and incremental generation, are discussed by van de Veen (1994). The semantic representation which she uses is basically similar to the indexed QLF of Phillips (1993). Unfortunately, van de Veen does not describe either the grammar or the generation algorithm used in the incremental system which she discusses.

7.3.1 Interactive Dialogues

Interactive dialogue management (like machine translation) lies outside the scope of this thesis. However, in Chapter 3 (Section 3.3.6) we described an experiment in which DCG English Engine was adapted for use as a natural language dialogue interface, and we now return to this topic. The dialogue system to which DCG English Engine was connected was PLUS, a Pragmatics-based Language Understanding System (Black et al., 1993). PLUS was a European ESPRIT project, but we only discuss the components developed at UMIST.

As we noted in Chapter 3, although HPSG was the preferred grammar formalism in PLUS, it was used only for parsing and not for generation. Instead of HPSG, the PLUS surface generator (Black and Cunningham, 1992) used categorial grammar. The main reason was that this enabled the generator, which had to be implemented at UMIST in Prolog, to be based on the existing generator developed at UMIST in Prolog by Phillips (1991) which used categorial grammar. Reusing the existing method outweighed the redundancy of having two different grammars for parsing and generation. However, in Section 7.2 we have shown that Phillips’ generation algorithm can in fact be used with HPSG.

The PLUS surface generator performs bag generation, not incremental generation. That is, a whole bag of semantic terms is given to the generator, they are not given one at a time. (This is a separate issue from whether the response is a syntactically complete sentence or an appropriate elliptical response). Jokinen (1996) gives an overview of the PLUS system, and describes the planning of dialogue responses by the dialogue manager (the ‘strategic’ part of natural language generation). The planned responses are passed from the dialogue manager to the natural language engine in the form of a list of indexed QLF representations. The PLUS surface generator (Black and Cunningham, 1992) then generates the surface form of the responses from the indexed QLF (the ‘tactical’ part of natural language generation) using a variant of Phillips’ algorithm with a categorial grammar.
Lager and Black (1994), discussing the PLUS surface generator, suggest that the algorithm and the QLF are suitable for incremental generation. The QLF consists of a flat list of terms, which the algorithm processes by starting with the first term in the list, using it to look up words from the lexicon to add to the chart, then moving along to the next term in the list and doing the same thing, then the next and so on. It would require only a small modification to Phillips’ algorithm to accept the QLF terms one at a time, and otherwise process them in exactly the same way.

Lager and Black (1994) also discuss the effect of using alternative versions of categorial grammar with such a modified incremental algorithm. Different conceptions of syntactic constituency would give rise to different levels of `talkativeness’ in the generator. We will return to this point in Section 7.3.3. First, we will take up their suggestion for an incremental version of Phillips’ algorithm, but instead of combining it with categorial grammar and indexed QLF, we will use HPSG and MRS.

7.3.2 Incremental Generation with HPSG

We now show how the bag generator for HPSG and MRS described in Section 7.2 can be modified for incremental generation with HPSG and MRS. The basic idea is that from an incomplete bag and a partial utterance, the generator attempts to continue the utterance as further semantic terms are added. We will include a simple form of repair to handle the case when the generator cannot find a way to continue the utterance.

Although real-time processing is an important aspect of incremental generation (Kilger and Finkler, 1995), we will ignore that issue here, and deal only with the order of inputs and outputs. Note in particular that, although it remains an important property of Phillips’ algorithm that the order of terms inside the bag is not significant, in the incremental version the order in which the terms are added to the bag does influence very strongly the utterance which is generated.

```prolog
incremental_generation :-
  abolish(edge,2),
  incremental_gen([],[]).
incremental_gen(Sem0,Phon0) :-
  input_term(Term),
  ( Term = end_of_file
   ; foreach(lookup_term(Term,Category),
          add_edge(Category,[])),
     utter([Term|Sem0],Phon0,Phon1),
     incremental_gen([Term|Sem0],Phon1)
  )
).
```

Figure 7.9: Incremental Generation

The basic incremental algorithm is shown in Figure 7.9. incremental_gen/2 is initialized with an empty bag of semantic terms Sem0 and an empty list of strings uttered so far Phon0. The procedure inputs a new semantic term Term, looks it up in the lexicon, and adds a new edge for each word found, thereby triggering construction of further edges. When all edges have been built, the procedure calls utter/3 and then recursively calls itself with the augmented bag of terms [Term|Sem0] and the augmented list of strings uttered Phon1.
A set of utterance rules is shown in Figure 7.10, numbered from 1 to 4. Each rule is attempted, in the order given, until an utterance is produced, or the default rule 4 is reached, which simply utters nothing and allows incremental gen/2 to input another semantic term.

% 1. Inactive edge: Continue
utter(Sem1,Phon0,Phon1) :-
  span_inactive(Sem1,Cat),
  string(Cat,Phon1),
  utter_continue(Phon0,Phon1),!.

% 2. Inactive edge: Repair
utter(Sem1,Phon0,Phon1) :-
  span_inactive(Sem1,Cat),
  string(Cat,Phon1),
  utter_repair(Phon0,Phon1),!.

% 3. Active edge: Continue
utter([Term|___],Phon0,Phon1) :-
  span_active([Term],Cat),
  string(Cat,Phon1),
  utter_continue(Phon0,Phon1),!.

% 4. Active Edge: Wait
utter(_,Phon,Phon).

span_inactive(Sem,Cat) :-
  clause(edge(Cat,[]),true),
  semantics(Cat,EdgeSem),
  permutation(Sem,EdgeSem).

span_active(Sem,Cat) :-
  clause(edge(Cat,[]),true),
  semantics(Cat,EdgeSem),
  permutation(Sem,EdgeSem).

utter_continue(Phon0,Phon1) :-
  append(Phon0,NewPhon,Phon1),
  write(' ... '), write(NewPhon), nl.

utter_repair(Phon0,Phon1) :-
  repair(Phon0,Phon1,Repair),
  write(' Er, ... '), write(Repair), nl.

repair([Word|Words0],[Word|Words1],Repair) :-
  repair(Words0,Words1,Repair).
repair(_,Words1,Words1).

Figure 7.10: Utterance Rules

Utterance Rule 1 succeeds if it finds an inactive edge which spans the bag of semantic terms, and whose PHON list is a continuation of the list uttered so far. (An inactive edge represents a complete syntactic constituent, and an edge “spans” a bag of terms if its own semantics is a permutation of the terms in the bag). In this case, the new part of PHON, continuing what has been uttered so far, is output by utter_continue/2.

Utterance Rule 2 performs a simple form of repair, when there is a complete syntactic constituent which spans the semantics, but its PHON does not continue the utterance which has been started.
In this case, `utter_repair/2` finds the minimum backtracking change needed to effect the repair, and utters the new part only, preceded by “Er,...”.

If there is no complete syntactic constituent (no inactive edge) which spans the bag of terms, the safest approach is to wait for more terms by Utterance Rule 4. However, if we want to make the generator more “talkative”, we can include Utterance Rule 3 which uses active edges from the chart. Active edges represent potential syntactic constituents which are not yet completed and may never be completed. They may turn out to be mistaken analyses, so uttering them prematurely increases the likelihood that the utterance will have to be repaired. Utterance Rule 3 simply continues the utterance with any active edge whose semantics matches the new semantic term. Improvements to this simple set of utterance rules need to be found by further work.

### 7.3.3 Syntactic Constituency

We now return to the question of syntactic constituency, and note that incremental generation appears to highlight some important differences between categorial grammar and HPSG.

Work on incremental generation has to some extent simply assumed that utterances must correspond to syntactic constituents. For example, discussing the possible adaptation of Phillips’ algorithm to incremental generation, Lager and Black (1994) point out that some versions of Categorial Grammar (CG) would make the generator more talkative, by giving rise to “a more generous notion of constituency”. However, it is possible to challenge the underlying assumption.

Differences between CG and HPSG in the basic approach to combining a head (for example, a verb) with its arguments (for example, a verb’s subject and complements) are significant here. Whereas in CG a head may be combined with its arguments one by one, giving a series of unsaturated intermediate constituents until a saturated one is completed, in HPSG a head is usually combined with all of its complements simultaneously in one constituent. For example, a verb is combined with all of its complements at once to make a VP, and then this VP is combined with the subject afterwards.

In incremental generation of English, after the subject has been generated, further semantic input may enable the verb to be generated next. In this case, CG may allow the subject and verb to be combined into a valid syntactic constituent, but HPSG will not recognise this as a constituent. If an incremental generator only utters valid constituents, an HPSG-based generator must wait, after uttering the subject, until all the complements of the verb have been identified, before uttering the verb as part of the complete verb phrase. This would be a significant problem in using HPSG for incremental generation, and a major advantage of CG over HPSG.

However, we can challenge the idea that it is syntactic constituency which decides what is uttered. In a dynamic, social, communicative activity such as interactive dialogue, syntactic factors are surely less important than semantic factors, and semantic factors in turn less important than discourse and pragmatic factors. Of course, all these factors are important, but which ones are decisive? It seems more likely that *information structure* plays a decisive role in deciding what is uttered than syntactic structure. We will discuss this briefly in Section 7.4.

From this point of view, the apparently significant distinction between Categorial Grammar and HPSG highlighted by incremental generation can be reduced to a computational matter. The
distinction between inactive edges and active edges in the chart can, if required, be overridden by more decisive factors. In our example of uttering the subject to be followed by uttering the verb, if the verb has been put into the chart as an active edge pending identification of its complements, it can (if required) be uttered immediately without waiting for the VP edge to be completed. Whether this will need to be repaired later or not is another question, which is not answered simply by adopting a more generous notion of constituency.

7.4 Elliptical Generation

For a dialogue response to be communicatively appropriate, it may well need to be syntactically elliptical. The need for generation of elliptical responses involves pragmatic factors which have not yet been adequately developed in HPSG theory. As described in Chapter 3, when DCG English Engine was adapted for use as a dialogue interface several ad hoc mechanisms were added to implement elliptical generation because there was no account available in HPSG theory, which was developed primarily as a theory of the syntax of sentences.

One major issue is the representation and treatment of information structure. Another major issue is what constitutes the starting point for the generation process. We will briefly look at these issues, but cannot do justice to them here.

7.4.1 Information Structure

The incremental system described in Section 7.3 does not take information structure into account, but is driven purely by the order in which semantic terms are supplied. There may be no alternative to having to rely on this order in some types of “shallow” system (for example, a simultaneous interpretation system). In this case, it is a matter of hoping that the order of the inputs will to some extent implicitly conform to the appropriate underlying information structure.

However, if the application system includes “deep” processing, for example involving complex logical inferencing, then it is obviously not satisfactory to rely on the chronological order in which an inferencing component happens to produce its results. The chronological order may depend on spurious factors such as the relative speeds of specific computer hardware components. A system with deep processing therefore needs explicit management of information structure.

We will briefly compare three approaches to information structure. These come from categorial grammar (Steedman, 1991), from HPSG theory (Engdahl and Vallduvi, 1996), and from the PLUS dialogue project (Jokinen, 1996).

In a framework of categorial grammar, Steedman (1991) argues that there is a direct correspondence between information structure, intonation and syntactic constituency. He therefore considers it to be a significant strength of categorial grammar that it allows suitable syntactic constituents to be created, which coincide neatly with information units.

By contrast, in a framework of HPSG theory, Engdahl and Vallduvi (1996) argue that there is no such correspondence between information structure and syntactic constituency. In fact, they maintain that it is a strength of HPSG’s multidimensional representation that we are not forced to assume such a correspondence.
Engdahl and Vallduvi (1996) propose that information structure is a distinct dimension, not part of syntactic category or semantic content. They therefore locate INFO-STRUCT within the CONTEXT feature in HPSG. However, the representation they propose is purely syntactic: LINK (topic) and FOCUS are equated with the surface syntactic constituents (the NPs and the VPs) which realize the topic concept and the focus information.

It could be argued that both of these two approaches over-emphasise syntactic factors in an area where semantic and pragmatic factors should more appropriately be given the central roles. In a footnote, Engdahl and Vallduvi (1996) accept that it would be more appropriate for the value of INFO-STRUCT to be structure shared with the value of CONTENT.

In the framework of the PLUS dialogue system, Jokinen (1996) describes how a pragmatics-based Dialogue Manager explicitly manages information structure, using a semantics-based representation. Her CentralConcept (Vallduvi’s Link) and NewInfo (Vallduvi’s Focus) are represented using QLFs with explicit indices for discourse referents. This facilitates distinguishing old and new information in the dialogue context, but the QLF lacks explicit representation of scope. This suggests that it would be interesting to include a representation of focus scope (“narrow focus” and “wide focus”) in an extended form of MRS-based representation with HPSG, in a similar way to the handling of quantifier scope.

### 7.4.2 Generation from New Information

Natural language generation has been applied primarily in the area of text generation (McKeown, 1985; Matthiessen and Bateman, 1991), in which text planning leads to surface generation of a sequence of syntactically complete sentences. For use in interactive dialogue applications, methods for generation of elliptical fragments have had to be developed.

Given the sentence-based starting point, one approach to elliptical generation is first to plan complete sentences as before, and then to decide which parts of their semantic representations can be deleted before surface generation takes place. By applying such principles as Grice’s Maxim of Quantity, unnecessary information can be deleted so as to leave only necessary information which will be realized in elliptical form.

The fundamental inadequacy of this approach is pointed out by Jokinen (1995). In the framework of the PLUS dialogue system, using the notions of CentralConcept and NewInfo mentioned above, Jokinen (1994) develops a new approach to generation which starts from NewInfo, the new information which is to be communicated. Basically, if the expression of NewInfo by itself meets the communicative requirements in the given dialogue context, then NewInfo alone is put on the agenda to be passed to the surface generator to be realized in elliptical form.

If items other than NewInfo need to be added in order to meet the communicative requirements, for example to provide an appropriate link using CentralConcept, then (and only then) they are added to the agenda and the communicative requirements are checked again. Checking the communicative requirements is done by means of a set of pragmatic principles (Jokinen, 1995).

The basic change is that instead of starting with a representation for a complete sentence and deleting unnecessary items, Jokinen’s approach starts with just a representation for NewInfo and then adds other items only if they are necessary. Elliptical utterances are no longer treated as
deviations from a norm (a full sentence), but as perfectly well-formed natural expressions.

We mentioned in Section 7.3.2 that we were ignoring real-time processing, which is an important aspect of incremental generation. Its importance is discussed by Kilger and Finkler (1995), with reference to the Verbmobil project. What is required for real-time processing is an any-time algorithm, which can give some appropriate response even when time runs out. From this point of view, Jokinen's approach seems to be closer to human dialogue behaviour. If there is time available for further processing, items other than NewInfo may be added to the agenda to improve clarity, avoid ambiguity and so on. If the time available runs out, at least the NewInfo is on the agenda ready to be passed to the surface generator. The shorter the time available, the more elliptical the response.

Further developments of this approach are described by Jokinen and Morimoto (1997) and Jokinen et al. (1998). In order to use HPSG for surface generation with such an approach, it is clear that we will need to develop a more adequate treatment of information structure. This is left for future work.
Chapter 8

Conclusion

In this concluding chapter, we briefly review some alternative approaches to natural language generation with HPSG. We then evaluate the work we have presented in the thesis and outline directions for future work.

8.1 Other Approaches

The principal approach to generation with HPSG which we have investigated in the main chapters of the thesis is head-driven generation. We also discussed some non-head-driven approaches in Chapter 7. We will now briefly review some other relevant approaches which are concerned with generation or with HPSG implementation.

8.1.1 Compiling HPSG to TAG

The first approach to mention is the radical one of converting HPSG into something else before generation. As we noted in the Introduction in Chapter 1, using HPSG for generation is almost as unpopular as using SFG for parsing. It certainly seems to give support to the view that HPSG is unsuitable for generation, if it is considered necessary to convert an HPSG grammar into something else in order to do generation. However, the actual outcome of research on this approach is rather positive, as it has made a valuable contribution to techniques for compiling HPSG grammars for efficient processing, both for parsing and for generation.

The specific form of the approach which has been developed is to compile an HPSG grammar into an equivalent Tree Adjoining Grammar (TAG) (Joshi, 1985). The possibility of such a compilation is proposed by Kasper et al. (1995). Its application to generation has been carried out by Becker et al. (1998) for the Verbmobil project (Wahlster, 1993), using the ERGO English HPSG grammar developed at CSLI, Stanford.

After the HPSG grammar has been compiled into a TAG grammar, the actual generation process follows an algorithm which is very similar to the semantic head-driven algorithm. It is described as fully lexicalized head-driven syntactic generation by Becker (1998). The aim of pre-compiling into TAG is to avoid on-line application of the HPSG schemata, using pre-compiled elementary TAG trees instead.
From a computational point of view, this approach to generation is well-motivated in that it aims to improve efficiency by a form of partial evaluation. If the compilation of HPSG to TAG is purely a form of partial evaluation, with no linguistic implications at all, then the approach can be evaluated purely in terms of the efficiency benefits which it provides. However, if the change from HPSG to TAG involves changes with linguistic implications, that is a different issue altogether. Of course, for the purpose of building a working application such as the Verbobil system, such theoretical issues are unlikely to be of much relevance.

8.1.2 Abstract Machines for HPSG

The implementations of HPSG which we have described in this thesis share a common foundation, as they are all built on top of an underlying Prolog system, specifically SICStus Prolog (SICS, 1995). The ALE system of Carpenter and Penn (1994), which we used for the implementation described in Chapter 2, is itself written in Prolog. More significantly, ALE compiles the user’s HPSG grammar into a Prolog representation, and executes the ALE parser as a Prolog program. The ProFIT system of Erbach (1995), which we used for the implementation described in Chapter 4, and for all the implementations in later chapters, is even more closely involved with the underlying Prolog system. ProFIT is offered simply as an extension of Prolog to handle typed feature structures, together with some other useful facilities such as templates. The user is expected to implement whatever framework is required (parser, grammar, lexicon, morphology, generator) in Prolog, using the extensions offered.

The formal specification of a grammar within a theoretical formalism like HPSG can be regarded as basically similar to the formal specification of a program in a high-level programming language. From this point of view, compiling an HPSG grammar into a Prolog representation in order to process it is like compiling a Prolog program into another programming language (such as Pascal) in order to process it. In the history of Prolog such methods were originally used, but much better ways to implement Prolog were then found. Similarly, it has been expected that better ways to implement HPSG grammars can be found, following the example of Prolog.

The most widely accepted method for implementing Prolog is to use a Warren Abstract Machine (WAM), as described by Ait-Kaci (1991). The Prolog programs are compiled into the instruction set of an abstract machine, the WAM. The abstract machine is itself implemented on a particular real machine. (This approach has also been used recently for implementing Java). In the case of SICStus Prolog, the abstract machine instructions are executed by a WAM emulator written in C (SICS, 1995).

In the ALE system of Carpenter and Penn (1994), the HPSG grammar is written as a typed attribute-value logic grammar, which ALE compiles into Prolog clauses. These Prolog clauses are then compiled into WAM instructions by the Prolog compiler. Finally, the WAM is executed. This intermediate Prolog level, between the ALE level and the WAM level, is unnecessary and may cause inefficiency.

Carpenter and Qu (1995) therefore propose that HPSG grammars at the ALE level should be compiled directly into instructions at the WAM level, and describe how this can be done. However, the WAM is specifically designed to execute Prolog programs efficiently. The abstract machine
for HPSG grammars should rather be specifically designed to execute typed attribute-value logic grammars efficiently.

### 8.1.3 Grammar Inversion

The proposals of Carpenter and Qu (1995) are concerned with abstract machine techniques for HPSG implementation, but they do not address how to do generation. The use of abstract machines for generation with HPSG has been investigated by Wintner et al. (1997). Their system, AMALIA, is an abstract machine for typed feature structures, described by Wintner and Francez (1995) and applied to unification grammars by Wintner (1997).

In order to perform generation, Wintner et al. (1997) adopt the technique of grammar inversion described by Samuelsson (1995). The rules of the grammar are reorganized to reflect the semantic predicate-argument structure rather than the surface string. Grammars are in fact restricted, as they are required to represent semantics by predicate-argument structures. The inversion algorithm is presented by Samuelsson (1995) in terms of DCG grammars, but it has been adapted for typed feature structures and applied to HPSG by Gabrilovich (1998).

The use of such an inverted grammar enables the AMALIA system to perform generation as well as parsing, as was demonstrated by Gabrilovich at the recent International Workshop on Natural Language Generation (INLG-98). The demonstration is described by Gabrilovich et al. (1998). However, the grammar which was inverted for this system is restricted, and in particular it does not contain any set-valued features (Gabrilovich, personal communication).

If set-valued features are not used, some of the problems of generation with HPSG are avoided. The system described in Chapter 4 and already published by Wilcock and Matsumoto (1996b) performs such generation efficiently using SGX. A challenge for generation with unrestricted HPSG grammars is how to include set-valued features, as discussed in Chapter 6.

### 8.1.4 Pre-compiling Phrasal Signs

Prof Tsujii has established a new research group at University of Tokyo, which has investigated HPSG implementation techniques in detail. They have used several techniques from the other approaches discussed here in their work. They have also studied the source code of the ProFIT-based HPSG implementation of Wilcock and Matsumoto (1996b) (described in Chapter 4) for comparison with their own system.

A technique for pre-compiling phrasal signs is described by Torisawa and Tsujii (1996), which they apply to parsing with HPSG. The basic idea of this technique is very similar to the method of compiling HPSG to TAG discussed in Section 8.1.1. Most of the techniques which the Tokyo group has developed could be applied equally in generation, although so far they have restricted themselves to parsing.

The Tokyo group has developed LiLFeS (Makino et al., 1997), an efficient parser for HPSG grammars. LiLFeS is based on the proposals of Carpenter and Qu (1995) for an abstract machine for attribute-value logic, discussed in Section 8.1.2.

The efficiency of the LiLFeS system is being continuously improved as new versions are developed. Comparisons of parsing speed by Makino et al. (1997) apparently show that LiLFeS is much
faster than ALE, and almost as fast as ProFIT. More recent comparisons by Makino et al. (1998) apparently show that after making various improvements, in some cases LiFeS is now even faster than ProFIT. However, no new versions of ProFIT have been made since 1995. It should also be pointed out that the new version 3.0 of ALE (Carpenter and Penn, 1998) is faster than the old version 2.0 (Carpenter and Penn, 1994) which was used for comparison, and ALE version 3.1 is faster than version 3.0.

In another activity at Tokyo, an underspecified grammar is used to extend coverage in HPSG parsing with LiFeS, as described by Mitsuishi et al. (1998). The basic idea of this approach is similar to the use of underspecification in parsing with HPSG by Wilcock and Matsumoto (1996b), described in Chapter 4.

8.2 Evaluation of the Research

We now turn to an evaluation of the work. A list of the aims of the thesis and a list of its original contributions are given in Chapter 1. Here we will merely give a brief assessment of whether the aims have been met and of the value of the contributions.

The aims of the thesis were to explore generation with HPSG which is a relatively unknown area, to investigate appropriate methods for HPSG implementation, to find out whether existing algorithms could be used with HPSG, to see whether HPSG grammars are reversible, and to see whether generation requires any theoretical changes in HPSG.

8.2.1 Generation with HPSG

The main aim of the thesis is to explore natural language generation with HPSG. We restricted the aim to surface generation, not sentence planning. As we argue that head-driven generation is the natural approach to generation with a head-driven grammar, we have focussed primarily on head-driven generation, but we have also explored non-head-driven generation in Chapter 7.

Within head-driven generation, we developed a basic method for successful generation with DCG grammars which approximate to HPSG, and then found a new way to combine this method with HPSG grammars using typed feature structures in ProFIT.

By focussing on head-driven generation, we were able to achieve more depth in the exploration, rather than merely widening it. We found that there are fundamental problems in handling quantifier scoping and contextual factors within head-driven generation. We then found a way to work towards a solution of these problems using lexical amalgamation.

8.2.2 HPSG Implementation

The ALE system is perhaps the most popular system for HPSG implementation, so its use in Chapter 2 is not at all original. However, the implementation described here is probably still one of the fullest ALE implementations of the 1994 HPSG textbook theory.

The combination of the SAX parser, the SAX morphological analyzer, and the SGX generator to make a practical English Engine is original, although the general design of the engine is explicitly based on that of the Core Language Engine. As a practical implementation, it is successful in terms of speed of parsing and generation. It was easy to modify the linguistic coverage in order to
adapt the system for use as a dialogue interface. However, as the number of rules increased, the difficulty of maintaining the grammar increased, which is inevitable with that kind of grammar.

The introduction of ProFIT in combination of SAX and SGX was a significant step forward. It made it possible to build a practical parsing and generation engine using an HPSG grammar. ProFIT is also used by Lappin and Shih (1996) for parsing English with HPSG in the ELLIP project, but not for generation. This method of combining SAX and SGX with ProFIT has subsequently been adopted by Miyata and Matsumoto (1998) for parsing and generation of Japanese legal texts.

The implementation of HPSG lexical rules described in Section 4.2.3 can be evaluated either in terms of HPSG theory, or in terms of the practical usefulness of the technique. The basic idea is that lexical rules are used on-line during morphological analysis, instead of being expanded off-line in a pre-compilation stage. This approach is of great practical value as it allows much faster compiling of the lexicon than in ALE. For practical grammar development, the slowness of grammar compilation with ALE has been a major bottleneck, which this approach eliminates completely. As a result, the HPSG English Engine framework can really be used for practical grammar development.

In terms of HPSG theory, it can be argued that the method used here (lexical rules used on-line during a morphological preprocess) is a step forward compared to ALE (lexical rules used during off-line lexicon compilation). However, it can also be argued that it does not go far enough, and that lexical rules should in fact be used on-line during syntactic analysis itself (Meurers and Minnen, 1995).

The ProFIT implementation of the proposals of Sag (1997) for English relative clause constructions is a practical contribution to HPSG implementation. The elimination of empty categories through Sag’s theoretical ideas justifies the use of SAX and SGX as implementation tools. The nested use of ProFIT templates as a practical way to implement Sag’s constraints on phrase types and clause types is also quite a useful technique.

8.2.3 Use of Existing Algorithms

The demonstration that several existing algorithms can be used with HPSG grammars is a useful contribution of the thesis. We have shown in detail how SAX and SGX can be used with HPSG.

We also showed that two relatively clear and simple existing algorithms can be used with HPSG. Van Noord’s BUG algorithm is the simplest form of head-driven generation algorithm, and Phillips’ bag generation algorithm is a simple form of non-head-driven generation algorithm. We showed how they can both be adapted for HPSG, with a fairly simple HPSG grammar in ProFIT. This may be useful for pedagogical purposes.

The described implementation of bag generation is primarily an adaptation of existing ideas and methods, and the modification of this approach to enable incremental generation is derived from the suggestion of Lager and Black (1994). The combination of this approach with MRS semantics to enable incremental generation with HPSG is a significant practical contribution to HPSG implementation.

The idea of specifying utterance rules is original. Further research is needed to develop this idea,
as well as to explore in detail the suggested chart-based solution to the problem which HPSG, in contrast to categorial grammar, has with highly incremental generation.

8.2.4 Reversibility

The implementation described in Chapter 4 demonstrates the reversibility of the HPSG grammar used, given the two distinct ways to compile it for parsing and for generation. The statement of the linguistic descriptions is the same in both cases.

The implementation of delayed lexical choice in generation with HPSG, described in Section 4.4, extends and generalises previous work. The established approach to delayed lexical choice is to replace the morphological lexicon with a lexicon of stems for the syntactic generation stage. Our method is more flexible, specifying a *syntactic-semantic lexicon* in place of a lexicon of stems. One advantage of this approach is that it can be applied to classes of words which do not have any kind of stems, such as pronouns. However, the real advantage is that it can be applied in order to delay choices for any kind of feature whatsoever, not only for morphological forms.

We showed that this whole approach to delayed lexical choice is reversible. The use of reversible delayed lexical choice in parsing with HPSG, described in Section 4.4.4, allows deliberate underspecification of selected features for robust parsing. The idea is promising, as it can be applied to any feature or combination of features. However, the specific applications of the idea which are described here are quite limited, and too closely related to the earlier morphologically-oriented approach. Further work is needed to show the full power of the idea.

8.2.5 Theoretical Changes

We have shown that most areas of HPSG grammar can be used in generation as well as parsing, without requiring any theoretical changes at all. This is not surprising, given the careful declarative way in which HPSG grammar is formulated.

However, we found that there are specific difficulties in handling set-valued features in head-driven generation. Given the use of phrasal amalgamation of quantifier stores and contextual backgrounds in Pollard and Sag (1994), it is difficult to include quantifiers and context successfully in the logical form for generation. We therefore believe that some theoretical changes are in fact necessary in order to use HPSG for generation.

The lexicalization of context is the main theoretical contribution of the thesis and constitutes a significant contribution to HPSG theory. The proposed change strengthens the role of semantic heads in the overall organization of HPSG grammar. It also solves fundamental theoretical problems with the inclusion of contextual information in head-driven generation. On a purely theory-internal level, it brings the treatment of context into line with the treatment of nonlocal features and the treatment of quantifier scoping.

The brief sketch of an analysis of register variation within Sag's new analysis of English relative clauses, presented in Section 6.3, is another theoretical contribution to HPSG. However, it will require more work to develop the analysis further, before its significance to the theory can be evaluated.
8.3 Future Work

Several areas suggest that further work would be useful.

One area of promise is the use of reversible delayed lexical choice in parsing with HPSG, described in Section 4.4.4. The method allows deliberate underspecification of selected features for robust parsing, which can be applied to any feature or combination of features. Further work is needed to show the full power of the idea.

The proposals in Section 6.3 for handling register variation within Sag's new analysis of English relative clauses are promising, but they are only sketched out briefly. More work is needed to develop the analysis further and extend it to other areas.

The idea of specifying utterance rules for incremental generation in an interactive dialogue is interesting, but the whole approach to generation in dialogues needs further work, including the treatment of information structure and dynamic dialogue context.
Bibliography

References


Christopher Manning and Ivan A. Sag. 1995. Dissociations between argument structure and grammatical relations. Tübingen HPSG workshop.


Appendix A: Demonstration for Chapter 4

This appendix shows a demonstration execution trace of the implementation of Chapter 4. Each sentence is parsed by SAX, to give a typed feature structure. As the entire structure is very large, only the logical form (the CONTENT feature of the structure) is displayed. The logical form is then input to SGX as the starting point for generation. SGX may generate more than one output from the logical form.

> sicstus -l load
sicstus -l load

SICStus 3.7.1 (SunOS-5.6-i386): Wed Oct 07 14:04:21 MET DST 1998
Licensed to umist.ac.uk
{compiling /home/bigdisk/graham/dev/PhD4/load.pl...}
{loading /usr/local/lib/sicstus-3.7.1/library/system.ql...}
{loaded /usr/local/lib/sicstus-3.7.1/library/system.ql in module system, 10 msec 31360 bytes}
{loading /usr/local/lib/sicstus-3.7.1/library/lists.ql...}
{loaded /usr/local/lib/sicstus-3.7.1/library/lists.ql in module lists, 10 msec 27120 bytes}
{compiling /home/bigdisk/graham/Tools/ProFIT1.54/profit1.54_sics3.pl...}
{Warning: save_program/2 - not redefined}

ProFIT 1.54 - 03 Dec 1995
(c) G. Erbach - DFKI and Universitaet des Saarlandes

{compiled /home/bigdisk/graham/Tools/ProFIT1.54/profit1.54_sics3.pl in module profit, 1420 msec 195452 bytes}
{compiling /home/bigdisk/graham/dev/PhD4/HPSG_Gram.sig...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Gram.sig in module profit, 630 msec 44016 bytes}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Gram.boo...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Gram.tpl...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.sig...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.boo in module profit, 20 msec 2080 bytes}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.tpl...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.tpl in module profit, 60 msec 4744 bytes}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.tencent...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.tencent in module profit, 0 msec 0 bytes}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.tpln...}
{compiled /home/bigdisk/graham/dev/PhD4/HPSG_Lex.tpln in module dict, 3100 msec 212380 bytes}

{compiled /home/bigdisk/graham/Tools/SAX2.0/sax.pl...}
{compiling /home/bigdisk/graham/Tools/SAX2.0/sax_utils.pl...}
{compiled /home/bigdisk/graham/Tools/SAX2.0/sax_utils.pl in module sax, 10 msec 1184 bytes}
{compiling /home/bigdisk/graham/Demo/FIT/SAX_User/sax_user.pl...}
{compiled /home/bigdisk/graham/Tools/SAX2.0/sax_user/extra_sax.pl...}
{compiled /home/bigdisk/graham/Tools/SAX2.0/sax_user/extra_sax.pl in module sax_user, 40 msec 2680 bytes}
{compiled /home/bigdisk/graham/Tools/SAX2.0/sax_user/input_sax.pl...}
{compiled /home/bigdisk/graham/Tools/SAX2.0/sax_user/input_sax.pl in module sax_user, 20 msec 2832 bytes}
{compiled /home/bigdisk/graham/Tools/SAX2.0/sax_user/morph_sax.pl...}

152
SAX Translator 2.0  #0: Tue Mar 09 1993
Loading /home/bigdisk/gram/Tools/SAX2.0/Utility/morph.pl...
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/morph.pl in module morph, 160 msec 16552 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user/morph_sax.pl in module sax_user, 190 msec 20368 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/treeprint.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/term_width.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/treeprint.pl in module treeprint, 450 msec 67384 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user/tree_sax.pl in module sax_user, 480 msec 69816 bytes}
{compiled /home/bigdisk/gram/Tools/Demo/FTT/SAX_User/sax_user.pl in module sax_user, 940 msec 116680 bytes}
{compiling /home/bigdisk/gram/Tools/SAX2.0/sax_user.pl in module sax, 880 msec 131160 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_trans_sics3.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_trans_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user.pl in module sax, 30 msec 3800 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_trans_sics3.pl in module sax_trans, 570 msec 60660 bytes}

SGX Translator 1.0  #0: Thu Mar 18 1993
Loading /home/bigdisk/gram/dev/PhD4/HPStat_Gram.plp...
{compiling /tmp/sp.GaGnd...}
{compiled /tmp/sp.GaGnd in module sax_user, 880 msec 43672 bytes}

SGX Translator 1.0  #0: Thu Mar 18 1993
Loading /home/bigdisk/gram/dev/PhD4/HPStat_Gram.plp...
{compiling /tmp/sp.GaGnd...}
{compiled /tmp/sp.GaGnd in module sax_user, 1150 msec 56184 bytes}

SGX Analysis:

```
| contribution
| | gram_unit
| | | phrase-----mark
| | | | phrase--phrase
| | | | | word word
| | | | | | She walks
```

153
SGX Generation:
She walks.

SAX Analysis:
contribution
| gram_unit
| phrase----mark
| phrase--phrase |
| word word
| word word
| word phrase
| word
| phrase phrase
| word
| word

SGX Generation:
She can walk.
SAX Analysis:

She can walking.

SGX Generation:
She can walk.

SAX Analysis:

I like apples.

SGX Generation:
I like apples.
Apples I like.
They will look at a book.

SGX Generation:
A book they will look at.
They will look at a book.

SAX Analysis:
contribution
gram_unit
phrase-----------------------------------mark
phrase-----phrase
word word-----phrase
| | | word-----phrase
| | | | | phrase-----phrase
| | | | | | word word-----phrase
| | | | | | | phrase-----phrase
| | | | | | | | word word
He says that she needs a car.
SGX Generation:
A car he says she needs.
A car he says that she needs.
He says she needs a car.
He says that she needs a car.
That she needs a car he says.

SAX Analysis:
contribution
| gram_unit
| phrase----------------mark
| phrase-----phrase
| | | |
| word word-----phrase
| | | | |
| | | word--phrase
| | | | | |
| | | | word
| | | | | | |
She tries to walk.

SGX Generation:
She tries to walk.
SAX Analysis:

```
SAX Analysis:
   contribution
   | gram_unit
   | phrase---------mark
   | phrase---------phrase
   | word word---------phrase
   | word word---------phrase
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   She tends to walk.
```

SGX Generation:

```
She tends to walk.
```

SAX Analysis:

```
SAX Analysis:
   contribution
   | gram_unit
   | phrase---------mark
   | phrase---------phrase
   | word word---------phrase
   | word word---------phrase
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   | word
   She believes him to walk.
```

SGX Generation:

```
She believes him to walk.
```

```
She believes that he walks.
That he walks she believes.
```
She persuades him to walk.

Kim we know Sandy claims Dana hates.
Kim we know Sandy claims Dana hates.
Kim we know Sandy claims that Dana hates.
Kim we know that Sandy claims Dana hates.
That Dana hates Kim we know Sandy claims.
That Dana claims Dana hates Kim we know.
That Sandy claims Dana hates Kim we know.
We know Sandy claims Dana hates Kim.
We know Sandy claims that Dana hates Kim.
We know that Sandy claims Dana hates Kim.
We know that Sandy claims that Dana hates Kim.

SAX Analysis:

A car? No, I need a van.
SGX Generation:
A car. No. I need a van.
A car. No. A van I need.

SAX Analysis:

```
gram_unit--gram_unit--gram_unit

phrase  phrase  phrase

phrase--phrase  word  phrase--phrase

word  word

word  word  phrase--phrase

word  word  phrase--phrase

a  car  no  i  need  a  van
```

```
if([index!B&
    restr![quants![det!exists&
        restrind!index!B&
        restr![nuc!H &
    ]&
    nuc!H&
    reln!car&
    inst!B&
    agr!3&sg\n]
]
)

prag!no1

```

161
SGX Generation:
A car. No. I need a van.
A car. No. A van I need.

SAX Analysis:

SGX Generation:
The car is expensive.

SGX Generation:
The car is expensive.

SAX Analysis:

/ contribution /
gram_unit /
phrase---------mark /
phrase----------phrase /
phrase--phrase word--phrase /
word phrase----phrase word /
word phrase--phrase /
word word word /
The big red car is expensive.

SAX Analysis:

/ contribution /
gram_unit /
phrase---------mark /
phrase----------phrase /
phrase--phrase word--phrase /
word phrase----phrase word /
word phrase--phrase /
word word word /
The big red car is expensive.
I would like to try to persuade her to believe that he tends to be asleep.
I would like to try to persuade her to believe he tends to be asleep.

I would like to try to persuade her to believe that he tends to be asleep.

That he tends to be asleep I would like to try to persuade her to believe.

SGX Generation:
I would like you to try to persuade her to believe that he tends to be asleep.
I would like you to try to persuade her to believe he tends to be asleep.

That he tends to be asleep I would like you to try to persuade her to believe.
SGX Generation:

I would like you to try to persuade her to believe he tends to look at the car.
I would like you to try to persuade her to believe that he tends to look at the car.
That he tends to look at the car I would like you to try to persuade her to believe.
The car I would like you to try to persuade her to believe he tends to look at.
The car I would like you to try to persuade her to believe that he tends to look at.
Appendix B: Demonstration for Chapter 5

This appendix shows a demonstration execution trace of the implementation of Sag (1997), which is described in Chapter 5. The procedure test0 inputs a series of noun phrases, which include relative clauses of different types.

Note that the grammar implemented here follows Sag (1997), and does not include restrictions on case assignment and register variation. Therefore two incorrect forms, the man at who we looked and the man at that we looked, are accepted as well as the correct form the man at whom we looked. Methods for implementing the required restrictions on case assignment and register variation are proposed in Chapter 6.

Note also that there are two distinct analyses for the man we looked at, a relative clause analysis and a topicalization analysis.

> sicstus -l load

SICStus 3.7.1 (SunOS-5.6-1386): Wed Oct 07 14:04:21 MET DST 1998
Licensed to umist.ac.uk

{compiling /home/bigdisk/gra ham/Re lC l/ load.pl...}
{loading /usr/local/lib/sicstus-3.7.1/library/system.pl...}
{loaded /usr/local/lib/sicstus-3.7.1/library/system.pl in module system, 10 msec 31360 bytes}
{loading /usr/local/lib/sicstus-3.7.1/library/lists.pl...}
{loaded /usr/local/lib/sicstus-3.7.1/library/lists.pl in module lists, 0 msec 27120 bytes}
{compiling /home/bigdisk/gra ham/Tools/ProF I T1.54/profit1.54_sics3.pl...}
{Warning: save_program/2 - not redefined}

ProFIT 1.54 - 03 Dec 1995
(c) G. Erbach - DFKI and Universitaet des Saarlandes

{compiled /home/bigdisk/gra ham/Tools/ProF I T1.54/profit1.54_sics3.pl in module profit, 1460 msec 195452 bytes}
{compiled /home/bigdisk/gra ham/Re lC l/Sag-96-Gram.sig...}
{compiled /home/bigdisk/gra ham/Re lC l/Sag-96-Gram.boot...}
{compiled /home/bigdisk/gra ham/Re lC l/Sag-96-Gram.tpl...}
{compiled /home/bigdisk/gra ham/Re lC l/Sag-96-Gram.tup in module profit, 310 msec 19152 bytes}
{compiled /home/bigdisk/gra ham/Re lC l/Sag-96-L ex.sig...}
{compiled /home/bigdisk/gra ham/Re lC l/Sag-96-L ex.sig in module profit, 0 msec 0 bytes}
{compiled /home/bigdisk/gra ham/Re lC l/HPSQ-L ex.pl...}
{compiled /home/bigdisk/gra ham/Re lC l/wlists.pl...}
{compiled /home/bigdisk/gra ham/Re lC l/wlists.pl in module wlists, 20 msec 12564 bytes}

168
{compiled /home/bigdisk/gram/dev/RelCl/HPSG-Trap.plp in module dict. 2280 msec 162744 bytes}
{compiling /home/bigdisk/gram/Tools/SAX2.0/sax.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user.pl in module sax, 10 msec 1184 bytes}
{compiling /home/bigdisk/gram/Demo/FIT/SAX/User/sax_user.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user(extra_sax.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user/input_sax.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user/morph_sax.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/User/sax.pl in module sax_user, 1100 msec 110000 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/User/tree_sax.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/User/treeprint.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/term_width.pl...}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/term_width.pl in module term_width, 170 msec 25160 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/treeprint.pl in module treeprint, 480 msec 67384 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/Utility/treeprint.pl in module sax_user, 520 msec 69816 bytes}
{compiled /home/bigdisk/gram/Demo/FIT/SAX/User/sax_user.pl in module sax_user, 126004 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_pl in module sax, 920 msec 141592 bytes}
{compiled /home/bigdisk/gram/Tools/SAX2.0/sax_user.pl in module sax_user, 3800 bytes}
{Warning: abolish(sax_user:sax_lexical_expansion) - no matching predicate}
{Warning: abolish(sax_user:sax_lexical_expansion) - no matching predicate}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user.pl in module sx, 20 msec 1680 bytes}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user.pl in module sx_user, 1000 bytes}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user(extra_sx.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user/output_sx.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user/tree_sx.pl...}
{Warning: abolish(sx_user:sx_lexical_expansion) - no matching predicate}
{Warning: abolish(sx_user:sx_lexical_expansion) - no matching predicate}
{compiled /home/bigdisk/gram/Demo/FIT/SX/User/sx_user.pl in module sx_user, 150 msec 26608 bytes}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx.pl in module sx, 200 msec 42672 bytes}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_trans_sics3.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_trans_sics3.pl in module sx_trans, 690 msec 89832 bytes}
{Warning: abolish(sx_user:sx_lexical_expansion) - no matching predicate}
{Warning: abolish(sx_user:sx_lexical_expansion) - no matching predicate}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user.pl in module sx_user, 150 msec 26608 bytes}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_user.pl in module sx_user, 200 msec 42672 bytes}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_trans_sics3.pl...}
{Warning: abolish(sx_user:sx_lexical_expansion) - no matching predicate}
{Warning: abolish(sx_user:sx_lexical_expansion) - no matching predicate}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_trans_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SX1.0/sx_trans_sics3.pl in module sx_trans, 690 msec 89832 bytes}
SXG Translator 2.0 #0: Tue Mar 09 1993
Loading /home/bigdisk/gram/dev/RelCl/HPSG-Scr.plp...
{compiled /tmp/spqa07c...}
{compiled /tmp/spqa07c in module sax_user, 4170 msec 137408 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_user.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_user(extra_sxg.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_user/morph_sxg.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_user/output_sxg.pl...}
{Warning: abolish(sxg_user:sxg_lexical_expansion) - no matching predicate}
{Warning: abolish(sxg_user:sxg_query_expansion) - no matching predicate}
{compiled /home/bigdisk/gram/Demo/FIT/SXG/User/sxg_user.pl in module sxg_user, 150 msec 26608 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg.pl in module sxg, 200 msec 42672 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_utils.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_trans_sics3.pl...}
{Warning: abolish(sxg_user:sxg_lexical_expansion) - no matching predicate}
{Warning: abolish(sxg_user:sxg_query_expansion) - no matching predicate}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_user.pl in module sxg_user, 150 msec 26608 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_user.pl in module sxg_user, 200 msec 42672 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_trans_sics3.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_trans_utils.pl...}
{loaded /usr/local/lib/sxicstus-3.7/1/library/charsio.pl...}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_trans_sics3.pl in module charsio, 10 msec 17928 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_trans_utils.pl in module sxg, 50 msec 22152 bytes}
{compiled /home/bigdisk/gram/Tools/SXG1.0/sxg_trans_sics3.pl in module sxg, 690 msec 89832 bytes}
SGX Translator 1.0 #0: Thu Mar 18 1993
Loading /home/bigdisk/gram/dev/RelCl/HPSG-Scr.plp...
{compiled /tmp/spra07c...}
{compiled /tmp/spra07c in module sxg_user, 4830 msec 143600 bytes}
{compiled /home/bigdisk/gram/dev/RelCl/HPSG-Echo.pl...}
{compiled /home/bigdisk/gram/dev/RelCl/HPSG-Echo.pl compiled, 60 msec 6320 bytes}
{compiled /home/bigdisk/gram/dev/RelCl/load.pl compiled, 23400 msec 1199088 bytes}
| ?- sax_lexical_expansion(on).
yes
| ?- test.
SAX Analysis: [The\_man\_who\_looked\_at\_us]

SAX Analysis: [The\_man\_that\_looked\_at\_us]
The man who we looked at

The man that we looked at
SAX Analysis: [The, man, we, looked, at]

phrase

phrase-----phrase

word phrase-----phrase

word word--phrase

word

The man we looked at

1f(index!C

restr![quants![]&
nuc!reln!man!

inst!C

quants![]&
nuc!reln!look$

agent!agr!1&pl&
theme!C&

agr!3&sg&m

]

)

phrase

phrase--------phrase

phrase--phrase phrase----phrase

word word word word--phrase

word

The man we looked at

1f(quants![]&

nuc!reln!look$

agent!agr!1&pl&
theme!agr!3&sg&m

)

SAX Analysis: [The, man, who, we, looked]
The man at whom we looked
The man looking at us

SAX Analysis: [The, man, looking, at, us]
SAX Analysis: [The, man, that, looking, at, us]

SAX Analysis: [The, man, at, whom, to, look]

phrase

phrase--phrase phrase--phrase

word word word--phrase word--phrase

The man at whom to look

SAX Analysis: [The, man, who, to, look, at]

SAX Analysis: [The, man, to, look, at]

phrase

phrase--phrase

word phrase--phrase

word word--phrase

The man to look at
Appendix C: Demonstration for Chapter 6

This appendix shows a demonstration execution trace of the experimental implementation of the lexicalization of context, which is described in Chapter 6 (Section 6.4).

The procedure `test/0` inputs a series of logical forms to the BUG semantic head-driven generation algorithm. The logical forms include contextual BACKGR features required for generation of pronouns and names and QSTORE features with quantifiers required for generation of quantified noun phrases. The listing shows each logical form in turn, followed by the generated sentence.

Note that BACKGR and QSTORE, which should be set-valued features, are implemented as Prolog difference lists rather than sets. The trace shows examples in which variation in the order of the list members causes generation to fail completely.

```plaintext
> sicstus -l load
SICStus 3.7.1 (SunOS-5.6-i386): Wed Oct 07 14:04:21 MET DST 1998
Licensed to umist.ac.uk

{compiling /home/bigdisk/graham/dev/HA5/load.pl...}
{compiling /home/bigdisk/graham/Tools/ProFIT1.54/profit1.54_sics3.pl...}
{loading /usr/local/lib/sicstus-3.7.1/library/lists.qi...}
{loaded /usr/local/lib/sicstus-3.7.1/library/lists.qi in module lists, 10 msec 25624 bytes}
{loading /usr/local/lib/sicstus-3.7.1/library/system.qi...}
{loaded /usr/local/lib/sicstus-3.7.1/library/system.qi in module system, 0 msec 28544 bytes}
{Warning: save_program/2 - not redefined}

ProFIT 1.54 - 03 Dec 1995
(c) G. Erbach - DPRI and Universitaet des Saarlandes

{compiled /home/bigdisk/graham/Tools/ProFIT1.54/profit1.54_sics3.pl in module profit, 1530 msec 249900 bytes}
{compiled /home/bigdisk/graham/dev/HA5/Gram13.sig...}
{compiled /home/bigdisk/graham/dev/HA5/gram13.sig in module profit, 1030 msec 65968 bytes}
{compiled /home/bigdisk/graham/dev/HA5/gram13.boo...}
{compiled /home/bigdisk/graham/dev/HA5/gram13.boo in module profit, 20 msec 2080 bytes}
{compiled /home/bigdisk/graham/dev/HA5/gram13.tpl...}
{compiled /home/bigdisk/graham/dev/HA5/gram13.tpl in module profit, 370 msec 21136 bytes}
{compiled /home/bigdisk/graham/dev/HA5/gram13.plp...}
{home/bigdisk/graham/dev/HA5/gram13_plp compiled, 570 msec 34880 bytes}
{compiled /home/bigdisk/graham/dev/HA5/BBU22.ppl...}
{/home/bigdisk/graham/dev/HA5/BBU22.ppl compiled, 20 msec 3896 bytes}
{/home/bigdisk/graham/dev/HA5/load.pl compiled, 5320 msec 456772 bytes}
|
```
John

Mary

[he, saw, the, girl]

[the, girl, saw, him]
[every, big, bad, boy, saw, a, good, little, girl]
inst!A1
]

[]

[]
&
ct!muc!rel!see&
act!A1&
<ref&
und!A1&
<ref

ct!backgr!-(D, D)&
qe!-(]det!exists&
restind!restr![reln!good&
inst!A1
.
reln!little&
inst!A1
.
reln!girl&
inst!A1
]

.
det!every&
restind!restr![reln!big&
inst!A1
.
reln!bad&
inst!A1
.
reln!boy&
inst!A1
]

]

[]

[]
&
ct!muc!rel!see&
act!A1&
<ref&
und!A1&
<ref

[a. good. little. girl. saw. every. big. bad. boy]

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

yes
| ?- halt.
>

182