Natural-Language Interpretation in Prolog

3 authors:

Björn Gambäck
Norwegian University of Science and Technology
114 PUBLICATIONS 1,806 CITATIONS
SEE PROFILE

Jussi Karlgren
Spotify
232 PUBLICATIONS 2,743 CITATIONS
SEE PROFILE

Christer Samuelsson
Deutsches Forschungszentrum für Künstliche Intelligenz
41 PUBLICATIONS 555 CITATIONS
SEE PROFILE

Natural-Language Interpretation in Prolog

Björn Gambäck Jussi Karlgren Christer Samuelsson

nlp@sics.se
Natural Language Processing Group
SICS — Swedish Institute of Computer Science,
Box 1263, S–164 28 KISTA, Sweden

Abstract

This booklet introduces natural-language processing in general and the way it is presently carried out at SICS. The overall goal of any system for natural-language processing system is to translate an input utterance stated in a natural language (such as English or Swedish) to some type of computer internal representation. Doing this requires theories for how to formalize the language and techniques for actually processing it on a machine. How this is done within the framework of the Prolog programming langauge is described in detail.

The SICS perspective on natural-language processing is that any theories regarding it is rather uninteresting unless put to practical use. Thus we are currently testing all our work within a particular large-scale NLP system called the Core Language Engine. In order to exemplify how natural language can be formalized in practice, the system is also fully described together with the ways languages are formalized in it.

Contents

Abstract iii
Table of contents v
List of figures ix

1 Introduction 1

2 Natural-Language Syntax 3
2.1 Formal syntax 4
2.1.1 Axiomatic systems and formal grammars 5
Types of grammars 6
Parse trees 8
2.1.2 Abstract machines 9
Finite-state automata 12
Pushdown automata 13
Turing machines and linear bounded automata 14
2.2 Natural language grammars 16
2.2.1 Context-free grammars 16
2.2.2 Unification grammars 20
2.3 Grammar formalisms 24
2.3.1 Definite Clause Grammar 24
2.3.2 Government and Binding 25
The $\overline{X}$-Convention 26
Transformations 28
Move-α — "Move anything anywhere" 30
2.3.3 Generalized Phrase Structure Grammar 30
Categories and features 32
Metarules 32
2.3.4 Lexical-Functional Grammar 33
2.4 A feature-value based grammar 35
2.4.1 Empty productions 35
WH-questions 39
2.4.2 Lexicalized compliments 42

3 Natural-Language Parsing 43
3.1 Top-down parsing . . . 43
3.2 Well-formed-substring tables . . . 44
3.3 Bottom-up parsing . . . 46
3.4 Link tables . . . 47
3.5 Shift-reduce parsers . . . 49
3.6 Chart parsers . . . 51
3.7 Head parsers . . . 56
3.8 LR parsers . . . 60
3.8.1 Parsing . . . 61
3.8.2 Parsed example . . . 62
3.8.3 Compilation . . . 65
3.8.4 Relation to decision-tree indexing . . . 65
3.8.5 Extensions . . . 66

4 The Core Language Engine 67
4.1 Overview of the CLE . . . 68
4.2 The CLE grammar formalism . . . 70
4.2.1 A toy grammar for Swedish . . . 70
The lexicon formalism . . . 71
4.2.2 The “real” grammar . . . 74
Feature defaults and value spaces . . . 74
Classification of sentence types . . . 75
A grammar rule . . . 75
4.3 Morphology . . . 77
4.3.1 Segmentation . . . 78
4.3.2 Inflectional morphology . . . 78
4.4 Compositional semantics . . . 79
4.4.1 The logical formalism . . . 80
Logical form requirements . . . 81
The predicate logic part . . . 81
4.4.2 The semantic rule formalism . . . 82
4.4.3 Semantic analysis . . . 84
4.4.4 Semantic rules . . . 85
4.4.5 Sense entries . . . 88
4.4.6 Higher order extensions . . . 89
4.4.7 Abstraction and application . . . 89
4.4.8 Generalized quantifiers . . . 90
4.4.9 Statements and questions . . . 91
4.4.10 “Quasi-logical” constructs . . . 91
4.4.11 Quantified terms and descriptions . . . 93
4.4.12 Anaphoric terms . . . 94
4.4.13 Event variables . . . 94
4.5 Later-stage processing . . . 95
4.5.1 Reference resolution . . . 96

4.5.2 Scoping 97

5 The Basics of Information Retrieval 99

5.1 Manual methods 99

5.2 Automatization 99

5.2.1 Words as indicators of document topic 99

5.2.2 Word frequencies - tf 100

5.2.3 Normalizing inflected forms 102

5.2.4 Uncommonly frequent words - idf 103

5.2.5 Methodological problems with idf 103

5.2.6 Combining tf and idf 105

5.2.7 Document length effects 105

5.3 Words as indicators of information need 106

5.4 Query expansion 106

5.4.1 Using retrieved documents 106

5.4.2 Relevance Feedback 106

5.4.3 Extracting terms from retrieved documents 107

5.5 Beyond single words 107

5.5.1 Collocations and Multi-word technical terms 107

5.5.2 Concept Spotting 108

5.5.3 Information Extraction 108

5.5.4 Text is not just a bag of words 108

5.5.5 Text is more than topic 108

5.5.6 Merging several information streams 109

5.6 Evaluating information retrieval 109

5.6.1 How exhaustive is the search? - Recall 109

5.6.2 How much garbage? - Precision 109

5.6.3 Combining precision and recall 109

5.6.4 What is wrong with the evaluation measures? 110

5.7 References 110

References 114

List of Figures

2.1 Grammar G_2 6
2.2 Grammar G_3 8
2.3 Grammar G_1 8
2.4 A sample parse tree 10
2.5 FSA for L_3 12
2.6 PDA for L_2 14
2.7 Grammar 1 16
2.8 Parse tree for "He (sees (a book about Paris))" 18
2.9 Parse tree for "He ((sees a book) about Paris)" 19
2.10 Inflection of the Icelandic adjective "langur". 21
2.11 Grammar G'_1 22
2.12 A possible noun-phrase branching structure 26
2.13 The branching in $\overline{X}$-theory 27
2.14 Swedish sentence structure 27
2.15 Swedish GB example 28
2.16 An example of government 29
2.17 The parse tree for a WH-question 31
2.18 The f-structure for "Kalle gillar Lena" 33
2.19 Grammar 2 36
2.20 Lexicon 2 37
2.21 An implicit parse tree 38
2.22 The parse tree for a WH-question 41

3.1 A simple top-down parser 44
3.2 A parser employing a well-formed-substring table 45
3.3 A simple bottom-up parser 47
3.4 A parser employing a link table 48
3.5 The chart-parser version of Grammar 2 53
3.6 The chart parser version of Lexicon 2 54
3.7 Head Grammar Version of Grammar 2 58
3.8 A Head Driven Parser/Generator 59
3.9 Generalized connect_others/3 60
3.10 Grammar 1 62

3.11 The internal states of Grammar 1 63
3.12 The LR parsing tables for Grammar 1 64

4.1 Broad overview of the CLE architecture 68
4.2 The analysis steps of the CLE 69
4.3 A Swedish grammar in the CLE formalism 72
4.4 A lexicon in the CLE formalism 73
4.5 Sentence classification by feature values 75
4.6 BNF definition of the predicate logic part 82
4.7 Generalized quantifiers 90
4.8 BNF definition of the "quasi-logical" part 92
4.9 The resolution steps of the CLE 96

Chapter 1

Introduction

The purpose of this booklet is to introduce natural-language processing in general and the way it is presently carried out at SICS. The term “natural language” marks in itself a distinction between the programming and formal languages normally used by computers (“artificial languages”) and the languages used by people. Here, we will of course not discuss all the possible aspects of all possible human languages, but rather restrict ourselves to discussing some techniques and theories which can be used while aiming at processing a language on a computer. Most of these will be rather universal in the sense that they could apply to any language; however, we will restrict ourselves even further and only illustrate the techniques by examples mainly from the two languages English and Swedish.

The overall goal of any natural-language processing (NLP) system is to translate an input utterance stated in a natural language to some type of representation internal to a computer, i.e., to *interpret* the utterance. The interpretation thus constructed will normally depend on the task for which the system is to be used. If the system for example is a front-end to a database query system, the internal representation could be the query language actually used by the underlying database (e.g., SQL). If the NLP system is used as a front-end to a operating system, the internal representation could be the operating-system commands, etc.

Commonly, however, the underlying system is abstracted away from the internal representation used and the interpretation will be an expression in some kind of logical formalism. Such a formalism will be described at the end of the booklet (in Section 4.4), but before reaching that level, we will go through several preliminaries in the first chapters.

Chapter 2 will start out by describing how languages can be formalized and processed in the first place, first discussing abstract machines and formal languages and then moving on to natural languages. Most contemporary theories of natural-language description are based on the notion of *unification.* The relevance of this (and how this is naturally implemented in the programming language Prolog) goes as a main theme through the entire booklet, but the discussion of it is initiated in Section 2.3 which describes some common theories used for natural-language grammar formalization.

Giving a language a formal description (i.e., a grammar) is not enough. In order to make anything useful out of the it, we need a way to process an input utterance given the grammar, or to *parse* the utterance. Doing this efficiently is a main research area within the NLP field and is thus discussed at length in Chapter 3. A number of different parsing strategies will be exemplified, but in general they have the common task of either rejecting an input word-string as being ungrammatical, or accepting it as grammatical and producing some useful output structure (such as a parse tree).

The SICS perspective on natural-language processing is that any theories regarding it is rather uninteresting unless put to practical use. Thus we are currently testing all our work within a particular large-scale NLP system called the Core Language Engine (CLE, for short), a system originally developed for English by a group at SRI International in Cambridge, England, and further extended and adapted to Swedish by the NLP group at SICS. The CLE is described in Chapter 4, which also goes into detail in describing how some particular natural-language are handled within the CLE. The main part of the chapter is, however, devoted to the logical formalism used and how utterances actually are interpreted by the system.

Chapter 2

Natural-Language Syntax

Natural-language systems generally translate an input utterance stated in natural language to some type of internal representation, i.e., they construct a semantic interpretation. This can be an expression in a logical formalism or SQL for a database query system; an interlingua or a transfer language in a machine-translation task; or a switch-board or operating-systems command in a command-language interface.

Normally the interpretation process is carried out in several steps. The utterance is usually analyzed syntactically in one of the first processing steps. In order to do this, one needs a formal way of describing the syntax of the natural language and a computing machine for analyzing the input utterance using this formal description. The former is usually a formal grammar and the latter a syntactic parser. Abstract machines take an intermediate position by fully specifying a language and offering a computational procedure for language analysis as well.

In Section 2.1 we will discuss two types of formal language descriptions, namely formal grammars and abstract machines. The reader interested in a more thorough treatment of this topic is referred to [Partee 1987]. In Section 2.2 we will discuss grammars that describe natural languages. We strongly recommend the book [Pereira & Shieber 1987] for the craft of natural-language description in Prolog, [Shieber 1986] for an introduction to unification grammars, and [Sells 1985] for an overview of contemporary syntactic theories of natural language.

2.1 Formal syntax

We will here give some necessary definitions for the further formal treatment of languages. The underlying definition is that a language is a set of strings. We will discuss the relevance of this assumption for the study and processing of natural languages in a section below.

Definition 1 (Alphabet)
An alphabet A is a finite set of symbols.

The set $\{a, b\}$, which consists of the two symbols a and b, is a fairly typical example of a formal alphabet. Slightly more practical alphabets are the sets $\{1, 0\}$, $\{a, b, c, ..., z\}$.

Definition 2 (String)
A string is a sequence of elements selected from an alphabet.

By sequence we mean a set where the elements are ordered. *aabb* is a string. *bbaa* is another. The strings are generally assumed to be finite but unbounded in length.

Definition 3 (String concatenation)
Concatenating two strings is simply sticking two strings after each other.

It is usually denoted by juxtaposition of the symbols representing the string. If $X = aa$ and $Y = bb$ then $XY = aabb$. The concatenation of a string with itself is usually denoted by raising it to the appropriate power. Thus $X^2 = aaaa$, $X^3 = aaaaaa$, $X^1 = aa$, and $X^0 = \epsilon$, where ϵ denotes the empty string.

Definition 4 (String length)
The length of a string A is written $\mid A \mid$.

Definition 5 (Language)
A formal language is defined as a set of strings.

Definition 6 (Operations on languages)
Languages can be operated on with any of the standard set theoretical operations: union, intersection, complement, etc. In addition, languages can be concatenated.

The language $L_A = \{a, aaa\}$ can be concatenated with the language $L_B = \{b, bb\}$ to form the language $L_A L_B = \{ab, aaab, abb, aaabb\}$. The same conventions as for string concatenation apply here, for ${L_A}^2 = \{aa, aaaa, aaaaaa\}$, ${L_A}^0$ = the empty set, and so forth.

The closure of the concatenation operation is denoted with a star, the so called Kleene star, named after S. C. Kleene of metamathematical fame.

Definition 7 (Kleene star)
X^* is the set of all strings formed with concatenations (including zero) from the string X, or in other words, the set of all X^i, where i is a natural number.

In an effort to cause confusion among mathematical linguists, the Kleene star is used for the closure of language concatenation as well as string concatenation. Thus ${L_A}^*$ is the set of all ${L_A}^i$, where i is a natural number. The Kleene star is used with an alphabet as an operand as well: A^* denotes the set[1] of all strings that can be formed from the alphabet. In practice, the different but related uses of the Kleene star do not usually cause the confusion they might be expected to.

Thus, using the above definitions, a language over an alphabet A is a subset of the set of all strings A^* that can be formed from the alphabet. As an example, let the language L_2 denote the set of strings in $\{a, b\}^*$ that consist of a number of a:s followed by an equal number of b:s. This language is often denoted $a^n b^n$ since each string in it consists of n a:s followed by n b:s for some number n.

2.1.1 Axiomatic systems and formal grammars

Formal grammars are one type of popular device for describing languages. A grammar is a meta-language for the language it describes, and is usually defined as an axiomatic system.

Definition 8 (Grammar)
A grammar is a quadruple $\langle \sigma, V_T, V_N, P \rangle$, where σ is the axiom or the top symbol, V_T and V_N are the alphabets and P is the actual content of the grammar.

V_T is the terminal alphabet that the resulting language is written in, and V_N is the nonterminal alphabet used in the formulæ of the meta-language. σ is a member of V_N. Often V_T and V_N are required to be disjunct.

The content of the grammar, P is a set of string-rewriting rules of the form $\psi \rightarrow \omega$. ψ is the left-hand side (LHS) of the rule and ω is the right-hand side (RHS), and both are strings over $V_T \cup V_N$.

The left-hand side should contain at least one nonterminal symbol. The rules are interpreted as follows: in any string where ψ occurs as a substring, ψ may be replaced by ω to form a new string.

Definition 9 (Derivation)
A derivation of a string ω is a sequence of strings starting with the top symbol $\sigma = \psi_1 \Rightarrow \psi_2 \Rightarrow ... \Rightarrow \psi_n = \omega$ where ψ_k can be rewritten as ψ_{k+1} using some grammar rule from P.

Take as an example the following derivation of the string *aaabbb* from the top symbol σ using grammar G_2:

$$\sigma \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

[1] Or in other words, language.

Definition 10 ($L(G)$)
The language $L(G)$ is the set of strings that a grammar G describes, or in other words, all strings that have derivations in L.

Take as a simple example the grammar G_2 in Figure 2.1 which generates the language $L_2 = a^n b^n$.

$$\begin{array}{rcl} V_T & = & \{a, b\} \\ V_N & = & \{S\} \\ \sigma & = & S \\ P & = & \left\{ \begin{array}{lcl} S & \rightarrow & aSb \\ S & \rightarrow & \epsilon \end{array} \right\} \end{array}$$

Figure 2.1: Grammar G_2

Types of grammars

Grammars according to the definition we have given so far are general string rewriting systems. We can, by restricting the formal definition of grammar rules, constrain the expressiveness of the grammar. The expressiveness of a grammar formalism is measured in terms of the complexity of classes of languages that the formalism can generate. Thus an extremely simple grammar formalism can only generate simple classes of languages, whereas a more complex formalism can generate complex classes.

As an example, if we were to define a class of grammars C_{11} where we restrict the size of the rule set P to one single rule with $\mid LHS \mid = \mid RHS \mid = 1$, we can see that it only can produce languages with single short elements. If we define another class of grammars C_{1n} where we remove the restriction on the length of the RHS; i.e., grammars where the rule set P has one single rule with $\mid LHS \mid = 1$, we see that C_{1n} includes C_{11}.

We will now define a series of grammars in order of decreasing complexity.

Unrestricted grammars The grammars as we have defined them so far are called unrestricted grammars.

Context-sensitive grammars If we restrict the grammar rules to be of the form $\alpha A \beta \rightarrow \alpha \psi \beta$ for some nonterminal symbol A, where ψ is not allowed to be the empty string ϵ, we get a context-sensitive grammar. Another way of expressing this restriction is that we disallow "shrinking rules": in context sensitive grammars $\mid LHS \mid$ is never greater than $\mid RHS \mid$.

A rule in a context-sensitive grammar is understood as stating that the nonterminal A may be rewritten as the string ψ in the context of α and

β. An alternative notation for context- sensitive grammar rules is $A \rightarrow \psi / \alpha \_ \beta$.

Context-free grammars If we limit the number of LHS symbols to one single, by necessity nonterminal symbol we get a context-free grammar. This means that each grammar rule is of the form $A \rightarrow \psi$ for some nonterminal symbol A. The name context free is natural in view of the fact that A may always be rewritten as ψ regardless of context.

Regular grammars If we in addition to this last requirement restrict the form of the RHS to be a single terminal symbol optionally followed by a single nonterminal symbol, we get a so-called regular grammar.[2] This means that grammar rules in regular grammars are of the form $A \rightarrow xB$ or $A \rightarrow x$ where the x is in V_T and B is a element in V_N.

These four types of grammars are by no means the only possible grammar classification. In fact, for natural language there is a large interest in grammars that are only slightly more expressive than context-free ones. However, these four types are well studied and have well understood properties, and are usually taken as a starting point for further study of the mathematical properties of formal grammars. Somewhat less imaginatively, Chomsky, who originally defined the hierarchy, chose to call the unrestricted grammars type 0 grammars, the context-sensitive grammars type 1 grammars, the context-free type 2 grammars, and the regular grammars type 3 grammars. We will not use this terminology here.

The classes of grammars defined above are proper subsets of each other.[3] The language generated by a context-free grammar, for example the example grammar G_2 above, is called a context-free language. Since a regular grammar is a special case of a context-free grammar, regular languages are also context-free languages. The set of regular languages is a subset of the set of context-free languages, which in turn is a subset of the set of context-sensitive languages,[4] which in turn is a subset of the set of unrestricted languages.

If we bound the string length, there can only be a finite number of strings in the language since the alphabet is finite. For this reason, this language class is called finite languages and it is a subset of the set of regular languages. Any of the grammar types above will only generate a finite language if the string length is bounded.

$L_3 = a^n b^m$, denoting the set of strings in $\{a, b\}^*$ that consist of some (unspecified) number of a:s followed by some (possibly different) number of b:s, is an example of a regular language described by the regular grammar G_3 of Figure 2.2, while L_2 above denoting $a^n b^n$ is a context-free language described by

[2] There is a trivial variation of regular grammars where the nonterminal *precedes* the terminal symbol.

[3] Apart for some technicalities involving context-sensitive grammars, the empty string ϵ, and the non-shrinking rule requirement.

[4] Again modulo the empty string ϵ.

the context-free grammar G_2 (see Figure 2.1). Note that L_2 is a sublanguage of L_3.

$$\begin{array}{rcl}
V_T & = & \{a, b\} \\
V_N & = & \{S, B\} \\
\sigma & = & S \\
P & = & \left\{ \begin{array}{lcl}
S & \rightarrow & aS \\
S & \rightarrow & bB \\
S & \rightarrow & \epsilon \\
B & \rightarrow & bB \\
B & \rightarrow & \epsilon
\end{array} \right\}
\end{array}$$

Figure 2.2: Grammar G_3

Let L_1 denote the language $a^n b^n c^n$ over $\{a, b, c\}$. This is an example of a context-sensitive language that is not context free. It is generated by the context-sensitive grammar G_1 shown in Figure 2.3.

$$\begin{array}{rcl}
V_T & = & \{a, b, c\} \\
V_N & = & \{S, T, X, Y\} \\
\sigma & = & S \\
P & = & \left\{ \begin{array}{lcl}
S & \rightarrow & aSX \\
S & \rightarrow & aX \\
aX & \rightarrow & abY \\
bX & \rightarrow & bbY \\
bY & \rightarrow & bc \\
cY & \rightarrow & cc \\
YX & \rightarrow & TX \\
TX & \rightarrow & TY \\
TY & \rightarrow & XY
\end{array} \right\}
\end{array}$$

Figure 2.3: Grammar G_1

Examples of languages that are not context sensitive are quite complicated and we will return to these when we discuss Turing machines in Section 2.1.2.

Parse trees

There are two different derivations of the string *aaabbbccc* from the top symbol σ using grammar G_1. One is the following:

$$S \quad \Rightarrow \quad aSX \Rightarrow aaSXX \Rightarrow aaaXXX \Rightarrow aaabYXX \Rightarrow$$

$$\begin{aligned}
&aaabTXX \Rightarrow aaabTYX \Rightarrow aaabXYX \Rightarrow\\
&aaabXTX \Rightarrow aaabXTY \Rightarrow aaabXXY \Rightarrow\\
&aaabbYXY \Rightarrow aaabbTXY \Rightarrow aaabbTYY \Rightarrow\\
&aaabbXYY \Rightarrow aaabbbYYY \Rightarrow aaabbbcYY \Rightarrow\\
&aaabbbccY \Rightarrow aaabbbccc
\end{aligned}$$

This is called the right-most derivation since the right-most substring is always expanded. The left-most derivation is:

$$\begin{aligned}
S \quad \Rightarrow \quad &aSX \Rightarrow aaSXX \Rightarrow aaaXXX \Rightarrow aaabYXX \Rightarrow\\
&aaabTXX \Rightarrow aaabTYX \Rightarrow aaabXYX \Rightarrow\\
&aaabbYYX \Rightarrow aaabbYTX \Rightarrow aaabbYTY \Rightarrow\\
&aaabbYXY \Rightarrow aaabbTXY \Rightarrow aaabbTYY \Rightarrow\\
&aaabbXYY \Rightarrow aaabbbYYY \Rightarrow aaabbbcYY \Rightarrow\\
&aaabbbccY \Rightarrow aaabbbccc
\end{aligned}$$

The only difference is that the rules were applied in a different order, but not in a structurally different way.[5] To eliminate this ambiguity, one often employs a *parse tree* to describe the derivation and the order in which the rule applications are carried out is left unspecified. In the example, both derivations are represented by the parse tree of Figure 2.4.

A parse tree is a connected directed acyclic graph. The arcs in the tree denote the dominance relation D. This relation is asymmetric (and thus irreflexive), and transitive. In the example parse-tree above the "deepest" node labelled X dominates a node labelled Y and a node labelled c. The parse tree has a single root, i.e., there is a unique minimal element w.r.t. D. This is the top-most S node in the example parse-tree.

There is also a partial order indicated by the horizontal position of each node — the precedence relation P. In the example the nodes labelled a precede the nodes labelled b. If one node dominates another, neither one of them precedes the other. Thus a pair of nodes $\langle x, y\rangle$ can be a member of P only if neither $\langle x, y\rangle$ nor $\langle y, x\rangle$ is a member of D.

Arcs in the tree are not allowed to cross. This means that if some node precedes another node, all nodes that the former dominates precede the latter, and that the former node precedes all nodes the latter node dominates. This can be state formally as if $\langle x, y\rangle$ is in P, then $\forall z : \langle x, z\rangle \in D\langle z, y\rangle \in P$ and $\forall z : \langle y, z\rangle \in D\langle x, z\rangle \in P$. There is also a function that maps the nodes of the tree into the set of symbols $V_T \cup V_N$, the so-called labelling function.

2.1.2 Abstract machines

Abstract machines are another type of popular device for describing languages. An abstract machine consists of a finite set of internal states, of which one is the

[5] One says that the two derivations are weakly equivalent.

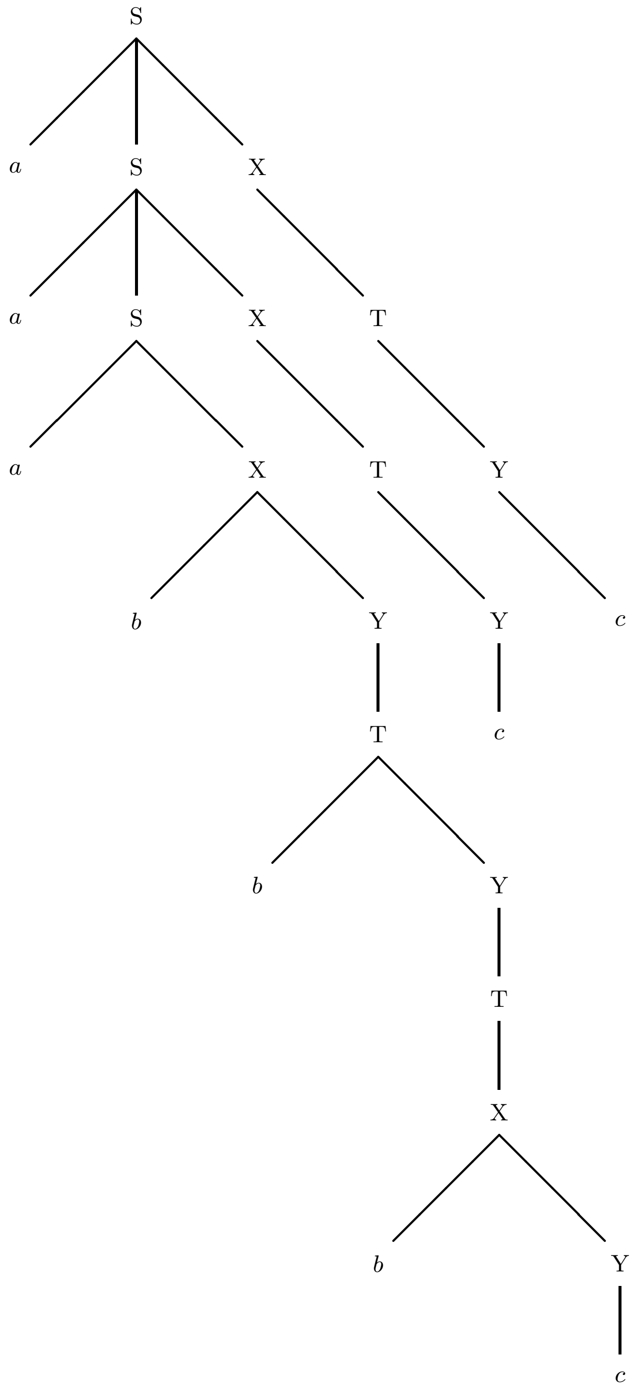

Figure 2.4: A sample parse tree

distinguished initial state and where a set of states are designated final states. In addition to this, the machine may have some other internal memory as well, depending on what type of machine it is.

The abstract machine is given an input string and its task is to determine whether or not the input string is a member of the language that the machine describes. There is a reader head to scan the input string and a transition relation between the internal states. This transition relation may take into account not only the current state and the current input symbol, but also the information present in the internal memory. The transitions may also change the content of the internal memory.

A machine *decides* a language if it will halt in finite time on any input string and the state in which it halts determines whether or not the string is a member of the language. A machine *accepts* a language if it halts in a final state for any input string that is a member of the language, but not for any string that is not.

We will discuss four types of abstract machines namely:

Finite state automata (FSA) decide regular languages, i.e., languages generated by regular grammars. Thus for each regular language L there exists some finite-state automaton that accepts all strings in L and that rejects all strings not in L. Further more, if some finite-state automaton decides a language L, then L is by necessity regular.

Pushdown automata (PDA) decide context-free languages, i.e., languages generated by context-free grammars. Thus for each context-free language L there exists some pushdown automaton that accepts all strings in L and that rejects all strings not in L. Also, if some pushdown automaton decides a language L, then L is by necessity context-free. Of course, it *may* be the case that L is in fact regular.

Linear bounded automata (LBA) accept context-sensitive languages, i.e., languages generated by context-sensitive grammars. Linear bounded automata and context-sensitive languages are not totally equivalent since ϵ is not a member of any context-sensitive language. For each context-sensitive language L, though, there exists a linear bounded automaton that accepts $L \cup \{\epsilon\}$. Likewise if there exists a linear bounded automaton that accepts a language L, then $L \backslash \{\epsilon\}$ is a context-sensitive language.

Turing machines (TM) decide recursive sets and accept recursively enumerable sets, i.e., languages generated by unrestricted grammars. For each recursive set there is a Turing machine that halts[6] on every input string and that halts in a final state only for input strings that are members of that set. Also, for each recursively enumerable set there is a Turing machine that halts on every input string that is a member of that set.

[6] A Turing machine may compute forever. If it does not for some particular input string, it is said to "halt" on that input string.

These abstract machines differ in how much internal memory they have at their disposal, and in what ways they may manipulate it. In the following, we will give definitions of these abstract machines. Note that there are other ways of defining them that give them the same expressive power, i.e., allows them to decide and accept the same classes of languages.

Finite-state automata

A finite-state automaton has no internal memory apart from the set of internal states. The reader head scans the input string from left to right, one symbol at a time. Thus the transition relation is determined solely on the basis of the current state and the current input symbol.

Definition 11 (Finite-state automaton)
A finite-state automaton is a tuple $\langle K, \Sigma, \Delta, \sigma, F\rangle$ where K is the set of states, Σ is the alphabet, Δ is a relation from $K \times \Sigma$ to K, σ is the initial state and $F \subseteq K$ is the set of final states.

The transitions can be denoted

$$(q_i, a, q_j)$$

This means that if the current state is q_i, and the current input symbol is a, then transit to state q_i and advance the reader head to the next input symbol. If the FSA is in a final state when the entire input string has been read, the input string is accepted by the FSA, otherwise the string is rejected.

Take as an example the simple automaton of Figure 2.5 deciding the language $L_3 = a^n b^m$:

$$\begin{array}{rcl}
K & = & \{q_0, q_1\} \\
\Sigma & = & \{a, b\} \\
\Delta & = & \left\{ \begin{array}{c} (q_0, a, q_0) \\ (q_0, b, q_1) \\ (q_1, b, q_1) \end{array} \right\} \\
\sigma & = & q_0 \\
F & = & \{q_1\}
\end{array}$$

Figure 2.5: FSA for L_3

Finite-state automata come in non-deterministic and deterministic varieties. For a deterministic FSA, the transition relation Δ is a function. This means that in each state there is exactly one transition for each possible input symbol.

For a non-deterministic FSA, there may be several different, one unique, or none possible transitions for each combination of state and input symbol.

A deterministic FSA is trivially non-deterministic. It can be shown that for each non-deterministic FSA there exists an equivalent deterministic FSA in the sense that both automata decide the same language.

Pushdown automata

A pushdown automaton is essentially an FSA where the internal memory has been extended with a pushdown stack, i.e., a “last-in first-out” queue. The top elements of the stack may be read or removed, or other symbols may be added above them, or the stack may be left unchanged. The stack symbols are not restricted to the terminal alphabet, i.e., the alphabet of the input string, but may contain symbols from the nonterminal alphabet as well.

The transitions can be represented as

$$(q_i, a, \alpha) \rightarrow (q_j, \beta)$$

This is to be interpreted as follows: In state q_i, with input symbol a and with the symbols α on top of the stack, transit to state q_j and replace α with the string β.

We admit both α and β to be the empty string ϵ. In the former case the action is independent of the stack context, in the latter the symbols α are popped from the stack. Formally:

Definition 12 (Non-deterministic pushdown automaton)
A non-deterministic pushdown automaton is described by a tuple $\langle K, \Sigma, \Gamma, \Delta, \sigma, F\rangle$ where K is the set of states, Σ is the input (i.e., terminal) alphabet, $\Gamma \subseteq V_T \cup V_N$ is the stack alphabet, Δ is a relation from $K \times \Sigma \times \Gamma^*$ to $K \times \Gamma^*$, σ is the initial state and $F \subseteq K$ is the set of final states.

Apart from the transitions in general being dependent on the symbols of the top portion of the stack, and the stack being manipulated by the transitions, a PDA works like an FSA. An input string is accepted if the PDA halts in a final state with an empty stack after reading the entire input string. Otherwise it is rejected.

Take as an example the simple pushdown automaton shown in Figure 2.6 which is deciding the language $L_2 = a^n b^n$.

There are deterministic and non-deterministic pushdown automata as well. The non-deterministic and the deterministic PDA are not equivalent, though. Intuitively this can be appreciated since there is no way for a deterministic PDA to know when the middle of a string is reached when attempting to recognize the language XX^R, i.e., the set of strings where the second half of the string is the mirror image of the first. This is a simple task for a non-deterministic PDA.

$$\begin{array}{rcl}
K & = & \{q_0, q_1\} \\
\Sigma & = & \{a, b\} \\
\Gamma & = & \{A\} \\
\Delta & = & \left\{ \begin{array}{l} (q_0, a, \epsilon) \rightarrow (q_0, \{A\}) \\ (q_0, b, \{A\}) \rightarrow (q_1, \epsilon) \\ (q_1, b, \{A\}) \rightarrow (q_1, \epsilon) \end{array} \right\} \\
\sigma & = & q_0 \\
F & = & \{q_0, q_1\}
\end{array}$$

Figure 2.6: PDA for L_2

Turing machines and linear bounded automata

A Turing machine has an internal memory that consists of an infinite tape on which it may read or write. In fact, we will assume that the input string is already written on the internal tape and that the reader head actually reads from that tape.

A TM can write a symbol on the tape or move the reader head one step to the right or to the left. The actions a TM can perform are determined by a finite set of quadruples

$$(q_i, a, q_k, X)$$

These should be interpreted as follows: In state q_i reading the symbol a, transit to state q_k and perform action X. If X is a symbol of the alphabet, write X on the tape (over-writing a). If X is the special symbol L then move the reader head one step to the left on the tape; if X is the symbol R move the reader head one step to the right. Formally,

Definition 13 (Turing machine)
A Turing machine is a tuple $\langle K, \Sigma, \Delta, s, F\rangle$ where K is a finite set of states, Σ is a finite set (the alphabet), Δ is a partial function from $K \times \Sigma$ to $K \times (\Sigma \cup \{L, R\})$, $\sigma \in K$ is the initial state and F is the set of final states.

Here we have assumed that the Turing machine is deterministic. Deterministic and non-deterministic Turing machines are equivalent in the sense that they determine the same classes of languages. In fact, no known “extension” to deterministic Turing machines extends their descriptive power. Thus a Turing machine is the most powerful abstract computing machine devised as of yet.

Since the reader head can move both left and right, an option open to a Turing machine, but not to an FSA or a PDA as they have been formulated above, is to compute forever. Thus for Turing machines there is a substantial difference between accepting languages and deciding them: If we know that the

Turing machine halts on any input and the state in which it halts determines whether or not the input string is in the language, it decides the language. If it is guaranteed to halt only on strings in the language, but may compute forever given strings that are not in the language, it accepts the language.

A linear bounded automaton is simply a Turing machine where the size of the available tape is limited and in fact bounded by some linear function of the length of the input string. Thereof the term “linear bounded”. Since a *pushdown automaton* must in each step consume an input symbol and can in each step only add a bounded number of items to the pushdown stack, the amount of memory available to a PDA is limited by a linear function of the length of the input string. Thus a pushdown automaton is a type of linear bounded automaton.

Whether deterministic and non-deterministic linear bounded automata are equivalent is still an open research question.

2.2 Natural language grammars

We will use the apparatus of formal language descriptions presented in the previous section for natural languages, but it is at this point important to discuss the appropriateness of that model.

Firstly, the assumption is that a language consists of strings over a finite alphabet. Since the alphabet will consist of the vocabulary of a natural language, and since this vocabulary is most likely infinite, this assumption is invalid.

Secondly, the strings are assumed to be unbounded in length, while the utterances made in a natural language most probably are not. This is an important observation, since any limit of the string length will eliminate the difference between the various language types, all of the languages will then be finite.

Thirdly, it is assumed that language membership is well-defined. This is not true for natural languages, since some sentences may be dubious, but neither obviously ”grammatical” nor ”ungrammatical” (in a more intuitive sense of the word).

2.2.1 Context-free grammars

We will take context-free grammars as a starting point for describing natural languages. Consider for example the small context-free grammar of Figure 2.7 describing a tiny fragment of English.

$$
\begin{array}{rcl}
V_T & = & \{a, about, book, gives, Paris, he, John, sees, sleeps\} \\
V_N & = & \{Det, N, NP, PP, P, Pron, S, V, VP\} \\
\sigma & = & S \\
P & = & \left\{
\begin{array}{lllllll}
 & \text{Grammar} & & & & \text{Lexicon} & \\
S & \rightarrow & NP\ VP & (1) & Det & \rightarrow & a \\
VP & \rightarrow & V & (2) & P & \rightarrow & about \\
VP & \rightarrow & V\ NP & (3) & NP & \rightarrow & Paris \\
VP & \rightarrow & V\ NP\ NP & (4) & NP & \rightarrow & John \\
VP & \rightarrow & VP\ PP & (5) & Pron & \rightarrow & he \\
NP & \rightarrow & Det\ N & (6) & N & \rightarrow & book \\
NP & \rightarrow & Pron & (7) & V & \rightarrow & sleeps \\
NP & \rightarrow & NP\ PP & (8) & V & \rightarrow & gives \\
PP & \rightarrow & P\ NP & (9) & V & \rightarrow & sees
\end{array}
\right\}
\end{array}
$$

Figure 2.7: Grammar 1

The first rule $S \rightarrow NP\ VP$ states that a sentence S (like *He sleeps*) can be constructed from a noun phrase NP (like “He” or “The man with the golden gun”) and a verb phrase VP (like “sleeps” or “is one of Roger Moore’s first

Bond movies"). Note that the structure of the noun phrase and the verb phrase is not specified by this rule.

The division of the rules into a grammar and lexicon part is for convenience. The LHS symbols of the rules of the lexicon are often referred to as *preterminals*. One way of coping with the potentially infinite number of words in natural languages is to consider the set of preterminals as the effective terminal alphabet V_T. Note however that for example "John" is a lexicalized *NP* and would in such a case violate the requirement that V_T and V_N be disjoint. Using lexicalized nonterminals is common practise in natural-language grammar design.

Figures 2.8 and 2.9 show two parse trees for the sentence *He sees a book about Paris* representing two structurally different analyses of the sentence. These two analyses are distinguished by how the prepositional phrase (*PP*) "about Paris" is attached; to the noun phrase "a book" in the former case (the natural reading) or to the verb phrase (*VP*) "sees a book" in the latter (a reading similar to *Pedro beats his donkey with a stick*). This and other types of ambiguity are very common in natural languages.

The grammar in Figure 2.7 is a so-called *phrase structure grammar (PSG)* where each rule specifies a possible structure of a phrase. This is by no means the only possible way of describing a natural language. In fact, another very successful way of doing this is to instead specify well-formedness constraints on a sentence. For example, there might be constraints such as that any well-formed sentence contains at least one finite verb. This is the approach taken in the constraint-grammar scheme, see [Voutilainen *et al* 1992].

Let us examine the nature of the rules of a phrase-structure grammar. Firstly, each rule has a LHS and a RHS which specifies the dominance relation locally. For example, the S of the first rule dominates the *NP* and the *VP*. Secondly the order of the phrases of the RHS specifies the precedence relation locally. In the example rule the *NP* precedes the *VP*. In languages with a large degree of freedom regarding word order, such as Finnish or Japanese, it might be more sensible to separate these two constraints. We could re-formulate the phrase-structure rule $S \rightarrow NP\ \ VP$ as

$$\begin{array}{rcl} S & \rightarrow & NP,\ VP \\ NP & \prec & VP \end{array}$$

The first part specifies the *immediate-dominance (ID)* relation and the second part specifies the *linear-precedence (LP)* relation, and such rules are often referred to as ID/LP rules. Note the somewhat subtle way of distinguishing a phrase-structure rule from an immediate-dominance rule by the use of a comma, rather than a blank character, to separate the phrases of the RHS of the rule.

For example, in Japanese the two constructions *Taro wa Hanako wo aishite iru* and *Hanako wo Tara wa aishite iru*, both meaning *John loves Mary*, are captured by the ID/LP rule

$$\begin{array}{rcl} S & \rightarrow & NP[+wa],\ NP[+wo],\ V \\ NP & \prec & V \end{array}$$

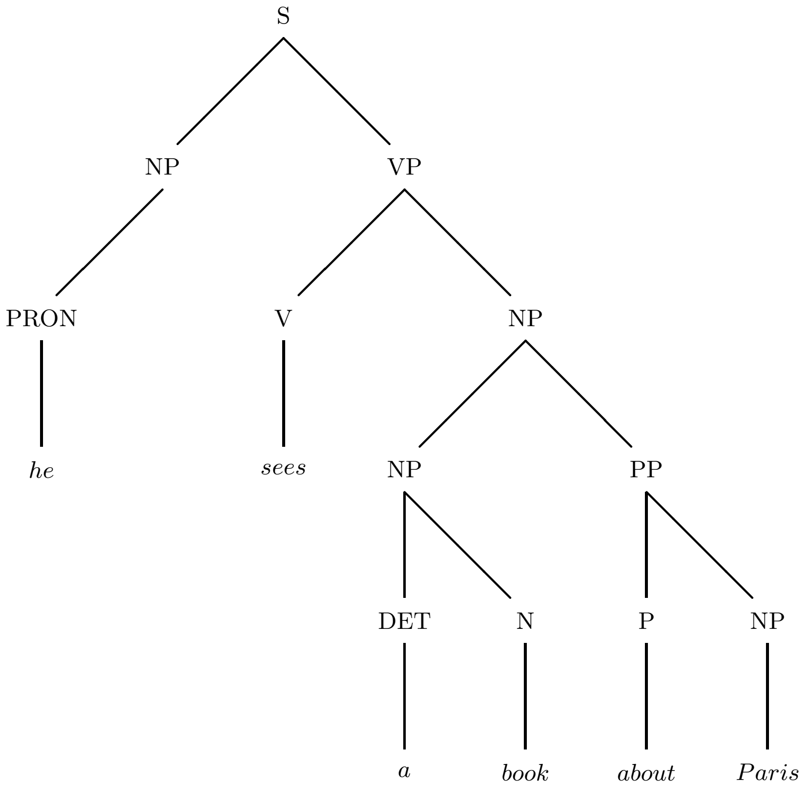

Figure 2.8: Parse tree for “He (sees (a book about Paris))”

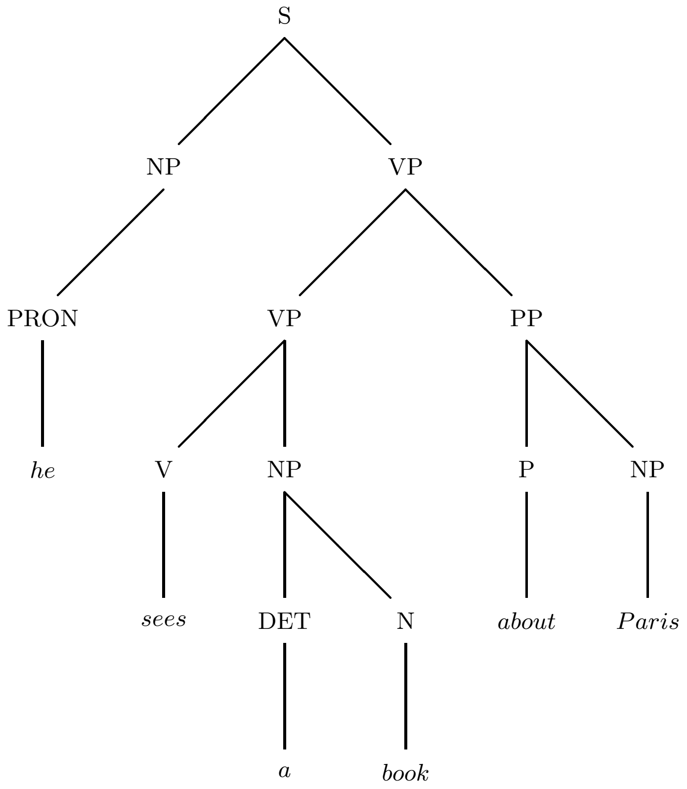

Figure 2.9: Parse tree for “He ((sees a book) about Paris)”

NP[+wa] means that the noun phrase is marked by the subject particle ”wa”. The above rule would for example reject the incorrect sentence *Tara wo Hanako wo aishite iru*. The separation of immediate dominance and linear precedence is an important ingredient in the generalized phrase-structure grammar scheme discussed in the next section. We will now turn to unification grammars.

2.2.2 Unification grammars

Experience has it that context-free grammars almost suffice for describing natural languages, and it has not been easy to find counter-examples. The most famous one is due to Stuart Shieber, see [Shieber 1985], and involves cross-serial dependencies in Swiss German.

Studies by Chytil and Karlgren [Chytil & Karlgren 1988], however, indicate that the extra expressive power needed is very small, and in particular very much less than that of a context-sensitive grammar. In view of this, and the observed discrepancy between the basic assumptions of the underlying models and the nature of natural languages, we will refrain from pursuing this rather academic discussion any further.

What is important, though, is the convenience of using various grammar types and the resulting parsing efficiency. The following example should convince the reader that natural languages are indeed context sensitive in the more intuitive sense that the form of a word is dependent on its context.

Figure 2.10 tabulates the 48 forms of the adjective ”tall” (literally ”long”) used in Icelandic in the positive sense for ”tall man”, ”tall woman” and ”tall child” respectively. The comparative and superlative forms have been omitted out of space considerations. Here “Nom.” stands for nominative, “Gen.” for genitive, “Dat.” for dative and “Acc.” for accusative case respectively.

It is obviously more convenient to write a single grammar rule for this than 48 different:

$$\begin{array}{lll} NP & \rightarrow & Adj : [case{=}Case, gen{=}Gen, num{=}Num, spec{=}Spec] \\ & & Noun : [case{=}Case, gen{=}Gen, num{=}Num, spec{=}Spec] \end{array}$$

This rule states that a noun phrase consisting of an adjective and a noun is well-formed if the latter two phrases agree w.r.t. case (*case*), gender (*gen*), number (*num*) and species (*spec*).

Here, the syntactic category has been augmented with a Prolog-type feature list and these are separated by a colon. The feature list consists of feature-value pairs of the type *Name=Value* where *Name* specifies the feature name and *Value* the feature value.[7]

Actually, the syntactic category could itself have been a feature-value pair such as *cat=Cat*, but by convention it generally is not. Normally only a finite number of different feature names are allowed.

[7] “=” is simply an operator joining the feature name and feature value.

Indefinite species				
Num.	Case	Masculine	Feminine	Neuter
Sing.	Nom.	langur ma∂ur	löng kona	langt barn
	Gen.	langs manns	langrar konu	langs barns
	Dat.	löngum manni	langri konu	löngu barni
	Acc.	langan mann	langa konu	langt barn
Plur.	Nom.	langir menn	langar konur	löng börn
	Gen.	langra manna	langra kvenna	langra barna
	Dat.	löngum mönnum	löngum konum	löngum börnum
	Acc.	langa menn	langar konur	löng börn
Definite species				
Sing.	Nom.	langi ma∂urinn	langa konan	langa barni∂
	Gen.	langa mannsins	löngu konunnar	langa barnsins
	Dat.	langa manninum	löngu konunni	langa barninu
	Acc.	langa manninn	löngu konuna	langa barni∂
Plur.	Nom.	löngu mennirnir	löngu konurnar	löngu börnin
	Gen.	löngu mannanna	löngu kvennanna	löngu barnanna
	Dat.	löngu mönnunum	löngu konunum	löngu börnunum
	Acc.	löngu mennina	löngu konurnar	löngu börnin

Figure 2.10: Inflection of the Icelandic adjective “langur”.

Two phrases can be unified if they are of the same syntactic category and if their feature lists can be merged without conflicting specifications on any feature value. This can be formalized as follows:

Definition 14 (Unification)
Two terms T_1 and T_2 can be unified iff there is a substitution θ such that $T_1\theta \equiv T_2\theta$. The result is that of applying the most general unifier (mgu), i.e., the minimal substitution θ' for which $T_1\theta' \equiv T_2\theta'$ holds, to either T_1 or T_2.

A substitution is an assignment of variables to terms or to other variables. A substitution may add extra feature-value pairs to feature lists if the feature name is not already present in the feature list. Equivalence ($\equiv$) is defined as follows:

Definition 15 (Equivalence)

1. *Two variables V_1 and V_2 are equivalent iff they are identical, e.g. $X \equiv X$.*

2. *Two atoms a_1 and a_2 are equivalent iff they are identical, e.g. $a \equiv a$.*

3. *Two terms $F_1(A_{11}, ..., A_{1N})$ and $F_2(A_{21}, ..., A_{2M})$ are equivalent iff they have the same functor, the same arity, and if the corresponding arguments are equivalent, i.e., iff F_1 and F_2 are identical, $N = M$ and $\forall_i A_{1i} \equiv A_{2i}$.*

4. *Two feature lists L_1 and L_2 are equivalent iff every feature name N_{1i} that occurs in L_1 also occurs in L_2, and vice versa, and if the corresponding feature values V_{1i} and V_{2i} are equivalent, i.e., $V_{1i} \equiv V_{2i}$.*

A term T_1 is said to subsume another term T_2 if the former can be made equivalent to the latter by instantiating it further, i.e., if and only if $T_1\theta \equiv T_2$ for some substitution θ.

Due to the use of feature lists, the described unification is not quite Prolog unification, but the latter can be used if all feature-value pairs are specified irrespectively of whether they are relevant in the context (and if they aren't, the value can be assigned the "don't care" variable "_") and if the feature-value pairs are arranged in some specific order w.r.t. the feature name.

This type of grammar is usually referred to as a *unification grammar (UG)*. Note that this deviates from the definition of formal grammars above, where the grammar symbols are atomic, and not allowed any internal structure. So how does this influence the expressive power of the grammar? This depends on the values that the features are allowed to take. If each feature is only allowed a finite value range, the expressive power is no more than that of a context-free grammar. In this case the grammar (formalism) is said to be weakly context free, since it can only generates context-free languages.

If the features are allowed a discrete range, we arrive at so-called indexed grammars, see [Aho 1968]. Such grammars are not as expressive as general context-sensitive grammars, but suffice for generating for example the language $L_1 = a^n b^n c^n$, which is not context free. Consider grammar G'_1 shown in Figure 2.11.

$$\begin{array}{rcl}
V_T & = & \{a, b, c\} \\
V_N & = & \{S, A:[f=\_], B:[f=\_], C:[f=\_]\} \\
\sigma & = & S \\
P & = & \left\{ \begin{array}{lcl}
S & \rightarrow & A:[f=N]\ B:[f=N]\ C:[f=N] \\
A:[f=0] & \rightarrow & \epsilon \\
A:[f=s(N)] & \rightarrow & a\ A:[f=N] \\
B:[f=0] & \rightarrow & \epsilon \\
B:[f=s(N)] & \rightarrow & b\ B:[f=N] \\
C:[f=0] & \rightarrow & \epsilon \\
C:[f=s(N)] & \rightarrow & c\ C:[f=N]
\end{array} \right\}
\end{array}$$

Figure 2.11: Grammar G'_1

The range of the single feature f is the set of natural numbers, here represented using Peano arithmetic. This grammar is arguably more comprehensible than the grammar G_1.

However, if the range of some feature value is unrestricted, the resulting grammar is Turing complete, i.e. it has the same descriptive power as unrestricted grammars. Thus it can describe languages that are extremely more complicated than any observed natural language. For this reason, many theorectical systems have been proposed to limit the number, range and co-occurrence of the features. Some of these theories are the topic of the next section.

2.3 Grammar formalisms

In this section we introduce some common grammatical theories. This brief overview of the state of the art in *generative grammar* is certainly not to be considered a complete record. Nor is it strictly necessary for the reader not interested in the particularities of grammatical analysis to dwell into this section — it is rather just a background to the grammatical theory that will be introduced in Section 4.2, the framework actually used in the Core Language Engine.

The formalisms we will look at are: "Definite Clause Grammar" (DCG), "Government and Binding" (GB), "Lexical-Functional Grammar" (LFG) and "Generalized Phrase Structure Grammar" (GPSG). For the reader with specific interest in any of these theories, each subsection will contain the most central references to the theory in question. General overviews and comparisons of these theories can also be found in for example [Sells 1985, Horrocks 1987].

All these theories build on "phrase structure rules" which define grammatical correctness. Such a rule can be viewed as a function from phrase-types to phrase-types, i.e. from a phrase to its daughter or daughters. For example `s -> np vp` is a rule stating that a sentence consists of a noun-phrase followed by a verb-phrase. `np` and `vp` would be defined by other (possibly recursive) phrase structure rules.

2.3.1 Definite Clause Grammar

Definite clause grammars [Colmerauer 1978, Pereira & Warren 1980] were "invented" by Alain Colmerauer as an extension of the well-known context-free grammars. DCGs are of interest here mainly because they can be easily expressed in Prolog. In general, a DCG grammar rule in Prolog takes the form

```
HEAD --> BODY.
```

meaning "a possible form for `HEAD` is `BODY`". Both `BODY` and `HEAD` are sequences of one or more items linked by the standard Prolog conjunction operator ','.

DCG grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyses some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. The following example shows how DCG grammar rules are translated into ordinary Prolog clauses by making explicit the extra arguments.

A rule such as

```
s --> np , vp.
```

translates into

```
s(P0, P2) :- np(P0, P1) , vp(P1, P2).
```

which is to interpreted as "there exists a constituent `s` from the point `P0` to the point `P2` in the input string if there exists an `np` between point `P0` and `P1` and a `vp` between `P1` and `P2`".

The way a Definite Clause Grammar extend context-free grammars is somewhat dependent on the Prolog dialect in which it is implemented. For example the DCG extensions used in SICStus Prolog are defined as:

1. A non-terminal symbol may be any Prolog term (other than a variable or number).

2. A terminal symbol may be any Prolog term. To distinguish terminals from non-terminals, a sequence of one or more terminal symbols is written within a grammar rule as a Prolog list. An empty sequence is written as the empty list `[]`. If the terminal symbols are character codes, such lists can be written (as elsewhere) as strings. An empty sequence is written as the empty list, `[]` or `""`.

3. Extra conditions, in the form of Prolog procedure calls, may be included in the right-hand side of a grammar rule. Such procedure calls are written enclosed in `{}` brackets.

4. The left-hand side of a grammar rule consists of a non-terminal, optionally followed by a sequence of terminals (again written as a Prolog list).

5. Alternatives may be stated explicitly in the right-hand side of a grammar rule, using the disjunction operator `;` or `|` as in Prolog.

6. The cut symbol may be included in the right-hand side of a grammar rule, as in a Prolog clause. The cut symbol does not need to be enclosed in `{}` brackets.

2.3.2 Government and Binding

The grammatical formalism that nowadays is known as "GB-theory" was introduced by Noam Chomsky in the fifties and has since been elaborated on both by Chomsky himself and by hundreds of his disciples [Chomsky 1957, Chomsky 1981, Chomsky 1986, van Riemsdijk & Williams 1986].

The formalisms GB, LFG and GPSG all have in common the notion of *Universal Grammar*, that is that the same type of grammar should be possible to use for all natural languages, and that indeed, all languages are rooted in the same grammar. In GB, cross-linguistic variation is specified by *parameters*. Other important concepts of that and the other theories will be described briefly. Starting off with GB, the central building-blocks of the theory are the $\overline{X}$-convention, transformations, c-command, and as the name indicates, the notions of government and binding.

The $\overline{X}$-Convention

Usually the phrase of the right-hand side of a phrase-structure rule that is most similar to the left-hand side is called the *head* of the rule. For example, the noun is the head of the rule $NP \rightarrow DET\ N$, the preposition is the head of the rule $P \rightarrow P\ NP$ and the two *NP*s of the RHS of a noun-phrase conjunction rule like $NP \rightarrow NP\ Conj\ NP$ are both heads. Although slightly controversial, the finite verb is often considered to be the head of a sentence. For each phrase we can follow the chain of heads down to some lexical item. The phrase is said to be a *projection* of this lexical item.

The head of any phrase is termed X, the phrasal category containing X is termed $\overline{X}$ (which is read as "X-bar"), and the phrasal category containing $\overline{X}$ is termed $\overline{\overline{X}}$ ("X double-bar"). The head is sometimes called X^0 and the node of completed phrases such as *NP*, *VP*, *PP* etc., that is the X with the maximal number of bars, normally phrases with a bar value of 2, is referred to as X^{max} (for "maximal projection") or XP. The semi-phrasal level ($\overline{X}$) is in practice often left out if it does not branch.

$$\begin{array}{lcl} \overline{\overline{N}} & \rightarrow & DET\ \overline{N} \\ \overline{N} & \rightarrow & N\ \overline{\overline{P}} \end{array}$$

Thus e.g. *NP* corresponds to $\overline{\overline{N}}$ and if built by the above rules would give a branching structure as the one shown in Figure 2.12.

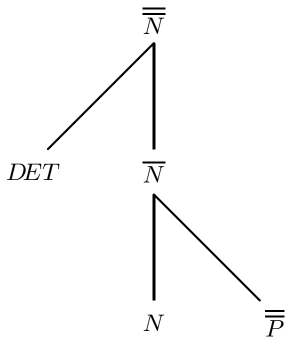

Figure 2.12: A possible noun-phrase branching structure

In general, the branching structure is as in Figure 2.13. For English (and Swedish) A is "specifiers" (determiner, degree, etc.), B is "modifiers" and D is "arguments" (C hardly exists).

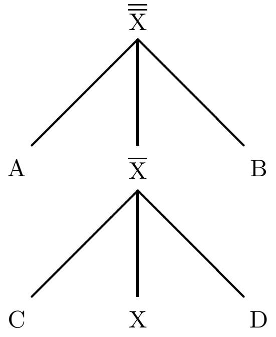

Figure 2.13: The branching in $\overline{X}$-theory

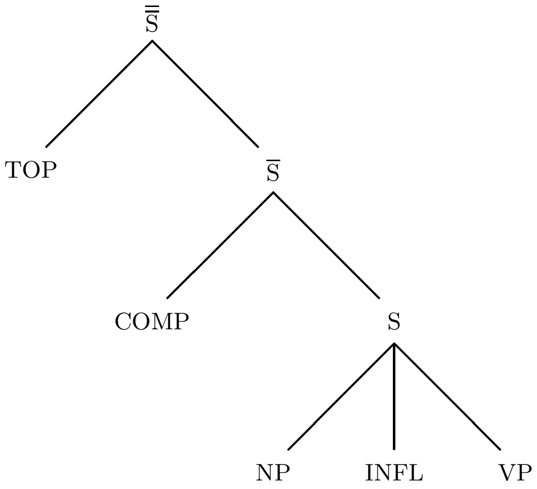

Figure 2.14: Swedish sentence structure

The sentence structure for Swedish is given in Figure 2.14. The *TOP* and *COMP* nodes are used for moved constituents, *INFL* ("inflection") for e.g., tense, aspect, and agreement. The sentence "Kalle gillar Lena" could be interpreted as the tree of Figure 2.15.

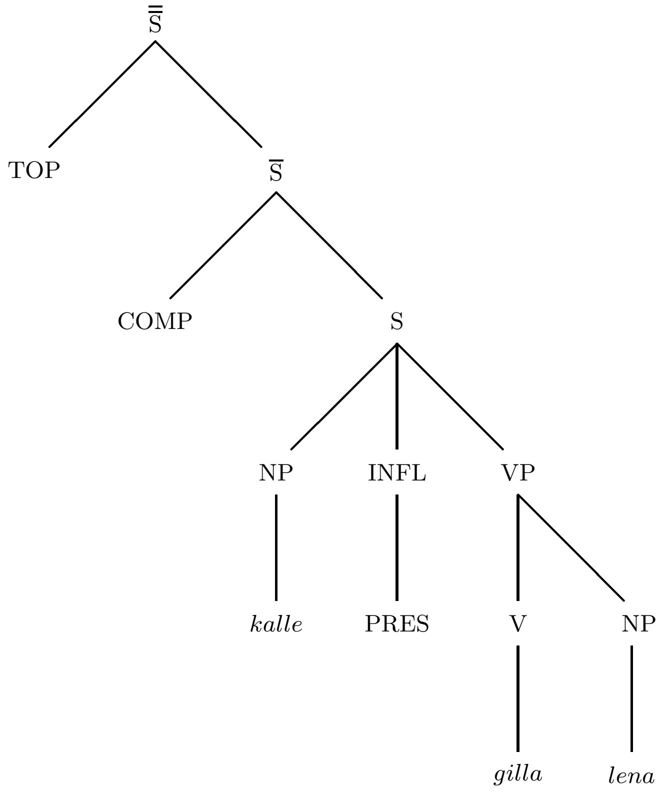

Figure 2.15: Swedish GB example

Transformations

The notion of transformations is based on a claim by Chomsky which basically says that "Phrase structures are not sufficient to describe human language". According to this view the description of natural languages also must rely on functions that destructively rearrange the phrase-structure trees. Going from some kind of deeper-level representation of the sentence ("D-structure"), the transformations are used to produce the output format (somtimes referred to as the "surface structure", even though most GB-theoreticians nowadays refrain from using the terms "deeper" and "surface"):

$$\text{D-structure} - {\scriptstyle\text{transformation(s)}} \rightarrow \text{S-structure}$$

Thus the S- and D-structures for normal (straight) sentences are the same, while the S-structures for e.g. questions are derived from the corresponding D-structure through the application of certain transformations. Most transformations are of the nature "move a constituent from one place in the sentence structure to another". The moved constituent will then leave a hole at its original position; this "hole" is referred to as a *trace* and its relation to the moved constituent is normally indicated by coindexing (a trace will be written as ϵ in the text following).

Classical transformation grammar theory contains a multitude of such transformations, while modern GB-theory seeks to minimize the number of transformations needed. The applicability of a transformation is expressed in terms of constraints defined by the following notions:

Definition 16 (C-command)
X c-commands Y iff the first branching node dominating X also dominates Y, X itself does not dominate Y, and Y does not dominate X.

Movement must always be to a c-commanding position and an anaphor must always be c-commanded by its antecedent (*anaphors* are reflexives, reciprocals, and obligatory control pronomina).

Definition 17 (Binding)
X binds Y if X and Y are coindexed and X c-commands Y.

Definition 18 (Government)
X governs Y iff Y is contained in the maximal $\overline{X}$-projection of X, X^{max}, and X^{max} is the smallest maximal projection containing Y and X c-commands Y.

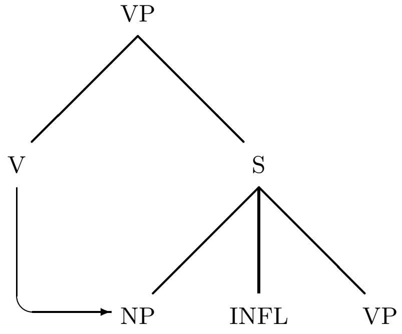

Figure 2.16: An example of government

The *governors* are normally X^0 (i.e., N, V, A, P). An example of government is given by the tree in Figure 2.16 where the V governs the NP. Note that this would *not* be the case if the top VP subcategorized for an $\overline{\overline{S}}$ (instead of an S), since this would introduce an X^{max} between the V and the NP. This is sometimes expressed as "$\overline{\overline{S}}$ is a barrier to government", or that $\overline{\overline{S}}$ (and the subtree it dominates) is an island isolated from the rest of the tree.

In general the governors (X in the definition) can be:

- X^0 (i.e., N, V, A, P)
- *INFL([+tns],AGR)*
- NP_i where the *governee* (the Y in the definition) is (another, but coindexed) NP_i

Move-α — "Move anything anywhere"

The ultimate goal of modern GB-theory is to have just *one* transformation, "Move-α", which takes care of all necessary movements of sentence constituents.

An S-structure is assumed to have been reached through (possibly several) applications of the Move-α transformation. In Swedish the *TOP* ("topic") node normally holds a topicalized *NP* or a WH-word,[8] while the *COMP* holds a V after question-transformation. Thus the sentence "Vem gillar Kalle?" would be regarded as having the same D-structure as "Kalle gillar Lena" above, and would be derived after two applications of Move-α. The first application turns the straight sentence into a Yes-No question with a verb-trace ($Gillar_i$ *Kalle* ϵ_i *Lena?*), while the second creates the S-structure shown in Figure 2.17, leaving a WH-trace in the place where the proper name "Lena" was before the transformation.

2.3.3 Generalized Phrase Structure Grammar

In the late seventies and early eighties Gerald Gazdar and others worked on the linguistic theory now known as GPSG [Gazdar *et al* 1985]. Their aim was to create a unification-based grammar formalism building on the idea that phrase-structure rules of PSG *without* transformations really are enough to describe natural languages. The key motivation behind the theory was actually simplicity, or rather, the use of a *simplified framework*:

- *one* level of syntactic representation: surface structure.
- *one* kind of syntactic object: the phrase structure rule.
- *one* empty category: *NULL* ($\approx$ GB's WH-trace).

[8] *WH-words* are the ones introducing *WH-questions*, so named simply because many of the question-words in English (i.e., who, which, where, why, how, etc.) begin with the letters "WH".

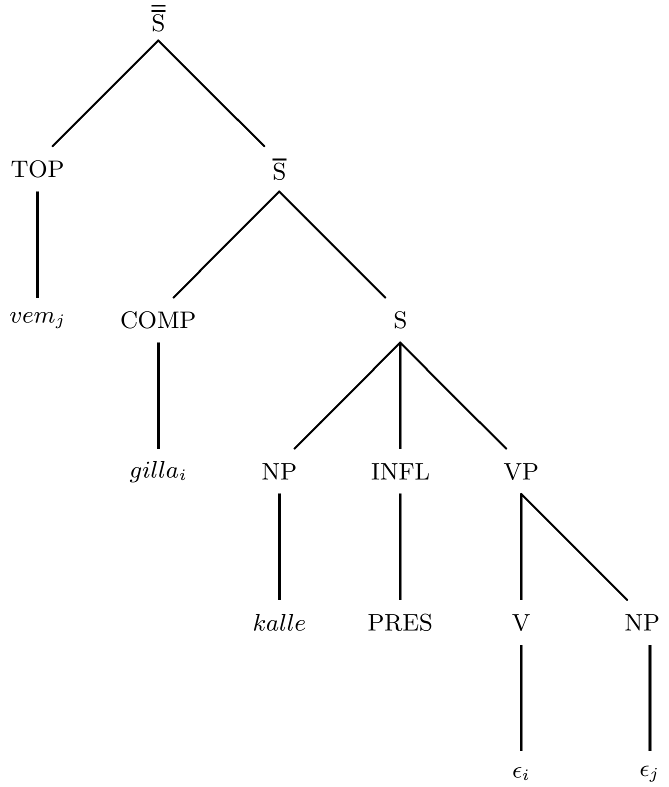

Figure 2.17: The parse tree for a WH-question

This simplification of the grammar rules is mainly achieved by using an elaborate category-feature system and by introducing so called “meta-rules”, that is phrase-structure rules that act on other phrase-structure rules. The phrase-structure rules are separated into ID and LP-rules as described above (see Section 2.2.1).

Categories and features

The GPSG theory does (like most unification-based theories) distinguish between categories and features, allowing two types of categories:

- **major**: N V A P
- **minor**: DET COMP ...

The major categories are the ones that participate in the $\overline{X}$ scheme, while the minor do not. The features are distinguished by two properties, namely their

- kind of value (e.g. atom-valued)
- distributional regularities (shared with other features)

To keep the grammar rules as simple as possible, the propagation of features is governed by several general principles and conventions, so that most feature instantiations in a rule need not be explicitly stated. One example of such a principle is the “head-feature convention” which will be described later on (see Page 42).

Metarules

GPSG has been a very influential theory in that it has introduced several concepts which now are central to most unification-based grammar theories. Thus we will not go into too much details of the theory as such here, but will introduce many GPSG concepts in other places in the text. One central notion, however, is that of metarules, which:

- perform some of the duties of GB’s transformations.
- derive (classes of) rules from (classes of) rules.

An example is the following “Passive Metarule” for Swedish:

$$\begin{array}{c} VP \;\;\rightarrow\;\; W\;\; NP \\ \Downarrow \\ VP[\mathrm{PAS}] \;\;\rightarrow\;\; W\;\; (PP[\mathrm{av}]) \end{array}$$

where W is a variable over categories. The rule says that a passive verb-phrase may be obtained by deleting a noun-phrase, possibly adding a prepositional phrase starting with the preposition “av” (of). This can be done regardless of whatever categories introduce the verb-phrase.

2.3.4 Lexical-Functional Grammar

LFG, like GPSG, is a unification-based theory based on categories and feature structures. The framework is described in [Kaplan & Bresnan 1982]. Of the grammar formalisms introduced in this section, LFG is the only one using the "school-book" grammatical functions: subject, object, etc.; however, this in hardly central to LFG as such, instead the main idea in the formalism is to assign two levels of syntactic description to a sentence: c- and f-structures.

The c-structure expresses word order and phrasal structure (i.e., "ordinary" trees obtained by phrase structure rules), while the f-structure encodes grammatical relations as functions from names to values:

$$SUBJ \quad \rightarrow \quad kalle$$

Information in the f-structure is represented as a set of ordered pairs each of which consists of an *attribute* and a specification of that attribute's *value*. An attribute is a name of a grammatical function or feature (*SUBJ, PRED, OBJ, NUM, CASE*, etc.).

Values are:

- Simple symbols (constants, below in *italics*).
- Semantic forms (constants, below in **bold face**).
- Subsidiary f-structures.
- Sets of symbols, semantic forms, or f-structures.

Definition 19 (Uniqueness Condition)
In a particular f-structure a particular attribute may have at the most one value.

A functional structure is a set of ordered pairs satisfying the Uniqueness Condition, i.e., a standard mathematical function.

$$\left[\begin{array}{ll} SUBJ & \left[\begin{array}{ll} SPEC & [\,] \\ NUM & sing \\ PRED & \textbf{kalle} \end{array}\right] \\ TENSE & PRES \\ PRED & \textbf{gilla}\langle(\uparrow \textbf{SUBJ})(\uparrow \textbf{OBJ})\rangle \\ OBJ & \left[\begin{array}{ll} SPEC & [\,] \\ NUM & sing \\ PRED & \textbf{lena} \end{array}\right] \end{array}\right]$$

Figure 2.18: The f-structure for "Kalle gillar Lena"

The f-structure for “Kalle gillar Lena” would be written as in Figure 2.18; showing a sentence in present tense (*TENSE=PRES*), where neither the subject (*SUBJ*) nor the object (*OBJ*) contain any specifiers (*SPEC* is the empty list, here depicted as [ ]), i.e., no determiners, etc., and both have singular number agreement (*NUM=sing*).

Natural-language semantics will be addressed further on in the text, but since we are not going to discuss LFG in more detail than done in this section, we note right away that the semantic interpretation of the sentence in LFG is obtained from the value of its predicate (*PRED*) attribute: **gilla**⟨(↑ **SUBJ**)(↑ **OBJ**)⟩, which is a predicate-argument expression containing the semantic predicate name (**gilla**) followed by an argument list specification enclosed in angle brackets.

The argument list specification defines a mapping between the logical arguments of the two-place predicate **gilla** and the grammatical functions of the f-structure: the first argument position is to be filled by the formula that results from interpreting the *SUBJ* function of the sentence, while the second position is to be the interpretation of the *OBJ*. The formula from the embedded *SUBJ* f-structure is determined by *its PRED* value, the semantic form **kalle**, and so on.

Arrows (↑ and ↓) are to be read as *immediate domination*. In the above example, the first argument slot of **gilla** is filled by the subject of the structure immediately dominating **gilla**, i.e., the sentence itself.

2.4 A feature-value based grammar

A key idea in the grammar formalisms of the previous section was to (in different ways) limit the number, range and co-occurrence of the features. We are now ready to use the mechanism of feature-value pairs to improve the grammar of Figure 2.7.

Consider the unification grammar of Figure 2.19. The anatomy of a rule `LHS => Id-RHS` is the following: `LHS` is simply the left-hand side of the grammar rule, `Id` is a mnemonic rule identifier and `RHS` is the right-hand side of the grammar rule (again in Prolog list notation).

For example, the first rule, named `s_np_vp`, corresponds to the context-free grammar rule $S \rightarrow NP\ VP$ of Grammar 1. In the new rule, the feature `agr` is an agreement feature ensuring that the sentences *He sleeps* is grammatical while sentences like *He sleep* or *They sleeps* are not. The feature `subcat` is a subcategorization feature for verbs specifying their complement type, thus ensuring that *He gives* is not grammatical, but that *He sleeps* and *He gives Mary a book* are. The feature `tree` is the parse tree of the phrase, reflecting its internal structure.

A unification grammar for a real natural-language system would employ a much larger set of features.

A term used in the following is *context-free backbone grammar*. This is a grammar that consists entirely of atomic (or at least ground) symbols, but that has the same structure as the underlying unification grammar. Such a grammar could be constructed for Grammar 2 above by for example omitting the feature lists and only taking the atomic syntactic categories into account. This would give us Grammar 1.

Another useful term is *implicit parse tree*, a tree where the nodes are the grammar rules resolved on at each point, rather than the syntactic categories. Implicit parse trees are convenient for describing UG derivations. Figure 2.21 shows such a tree for the sentence *He sees a book about Paris.* (The lexicon look-ups are commonly not shown as explicitly as in this tree.)

2.4.1 Empty productions

Another creative use of features is associated with empty productions, or as they are often called in UG theory, "gaps". The rule `np_gap` introduces an empty *NP* construction, i.e., an NP-trace:[9]

```
np:[agr=Agr,gaps=([np:[agr=Agr,wh=y]|G],G),wh=n] =>  np_gap-
     [].
```

Gap rules like this one are used to model *movement* as in the sentence *What*$_i$ *does John seek* ϵ_i*?*, which is viewed as being derived from its declarative counterpart *John seeks what?*. The trace "ϵ_i" marks the position from which the

[9] For the sake of clarity, the `tree` feature has been omitted in the following.

```
top_symbol(s:[tree=_]).

s:[tree=s(NP,VP)] =>                             s_np_vp-
     [np:[agr=Agr,tree=NP],
      vp:[agr=Agr,tree=VP]].

vp:[agr=Agr,tree=vp(V)] =>                       vp_v-
     [v:[agr=Agr,subcat=intran,tree=V]].

vp:[agr=Agr,tree=vp(V,NP)] =>                    vp_v_np-
     [v:[agr=Agr,subcat=tran,tree=V],
      np:[agr=_,tree=NP]].

vp:[agr=Agr,tree=vp(V,NP1,NP2)] =>               vp_v_np_np-
     [v:[agr=Agr,subcat=ditran,tree=V],
      np:[agr=_,tree=NP1],
      np:[agr=_,tree=NP2]].

vp:[agr=Agr,tree=vp(VP,PP)] =>                   vp_vp_pp-
     [vp:[agr=Agr,tree=VP],
      pp:[tree=PP]].

np:[agr=Agr,tree=np(DET,N)] =>                   np_det_n-
     [det:[agr=Agr,tree=DET],
      n:[agr=Agr,tree=N]].

np:[agr=Agr,tree=np(PRON)] =>                    np_pron-
     [pron:[agr=Agr,tree=PRON]].

np:[agr=Agr,tree=np(NP,PP)] =>                   np_np_pp-
     [np:[agr=Agr,tree=NP],
      pp:[tree=PP]].

pp:[tree=pp(P,NP)] =>                            pp_p_np-
     [p:[tree=P],
      np:[agr=_,tree=NP]].
```

Figure 2.19: Grammar 2

```
lexicon(a,det:[agr=sg,tree=det(a)]).
lexicon(the,det:[agr=_,tree=det(the)]).
lexicon(several,det:[agr=pl,tree=det(several)]).

lexicon(about,p:[tree=p(about)]).
lexicon(in,p:[tree=p(in)]).

lexicon(paris,np:[agr=sg,tree=np(paris)]).
lexicon(john,np:[agr=sg,tree=np(john)]).
lexicon(mary,np:[agr=sg,tree=np(mary)]).

lexicon(he,pron:[agr=sg,tree=pron(he)]).
lexicon(she,pron:[agr=sg,tree=pron(she)]).
lexicon(they,pron:[agr=pl,tree=pron(they)]).

lexicon(book,n:[agr=sg,tree=n(book)]).
lexicon(books,n:[agr=pl,tree=n(books)]).

lexicon(house,n:[agr=sg,tree=n(house)]).
lexicon(houses,n:[agr=pl,tree=n(houses)]).

lexicon(sleeps,v:[agr=sg,subcat=intran,tree=v(sleeps)]).
lexicon(sleep,v:[agr=pl,subcat=intran,tree=v(sleep)]).

lexicon(gives,v:[agr=sg,subcat=ditran,tree=v(gives)]).
lexicon(give,v:[agr=pl,subcat=ditran,tree=v(give)]).

lexicon(sees,v:[agr=sg,subcat=tran,tree=v(sees)]).
lexicon(see,v:[agr=pl,subcat=tran,tree=v(see)]).
```

Figure 2.20: Lexicon 2

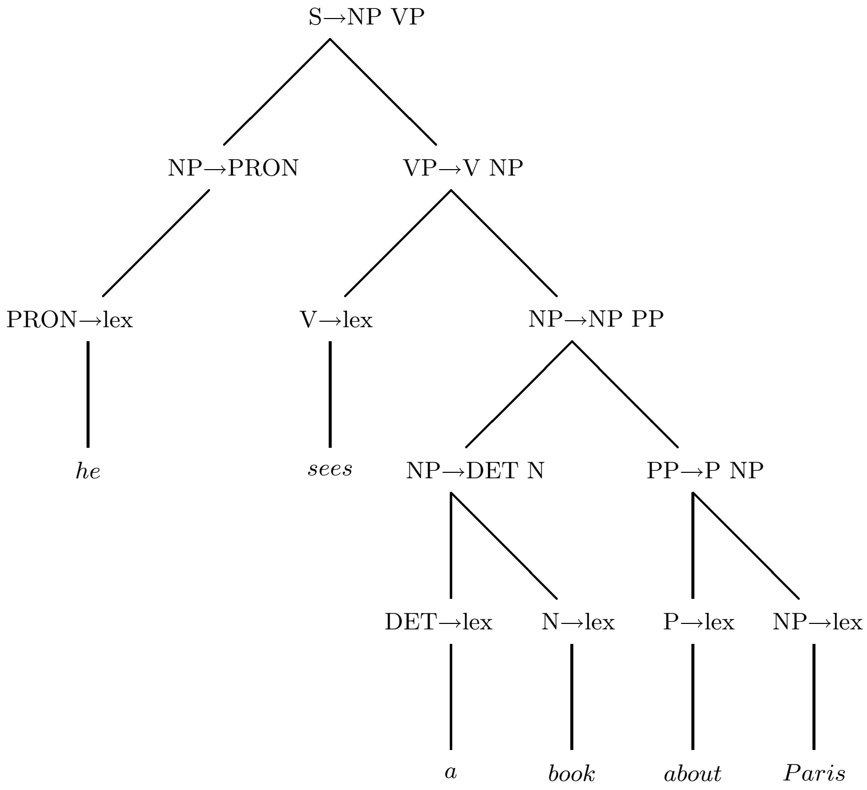

Figure 2.21: An implicit parse tree

word "what$_i$" has been moved (and co-indexing, here with subscript index i is used as before to associate the moved word with the trace).

This is an example of *left movement*, since the word "what" has been moved to the left. Examples of right movement are rare in English, but frequent in other languages, the prime example being German subordinate clauses.

The feature `gaps` is used for gap threading, i.e., passing around a list of moved phrases to ensure that an empty production is only applicable if there is a moved phrase elsewhere in the sentence to license its use.

WH-questions

We will now exemplify several aspects of the feature-value concept and the treatment of empty productions in unification grammars by trying to extend Grammar 2 to handle WH-questions. In order to do this, we will need gap-rules like the one above and might also introduce the binary-valued features `fin` and `wh` indicating whether the main verb is finite or not and whether a word (or, to be exact, an *NP*) is a question-word or not, respectively.

With a feature setting `fin=y` on the top-symbol we can impose a restriction on the sentences which will be accepted by the grammar, namely that they must contain a finite verb. Non-finite verbs would in contrast be the ones which could form the lower *VP* of a verb-phrase formation rule $VP \rightarrow V\ VP$ used for auxiliary verbs, i.e., verbs like "do" which can be viewed as subcategorizing for another (non-finite) verb-phrase.

Whether a verb is finite or not can in this simplified framework be captured a setting the feature in the lexicon, but will in a real system be obtained from *morphology rules* (morphology will be discussed briefly later on).

Whether an *NP* consists of a question-word or not would also be indicated in the lexicon. Words with `wh=y` would be the only ones for which a specific question-rule $S \rightarrow NP\ V\ S$ would apply. This rule would state that a sentence can consist of a (moved) WH-word followed by a (moved) verb and another sentence. The lower *S* should thus have gaps for both the verb and the WH-word.

In English the moved verb must be an auxiliary. In many other languages (e.g., Swedish), this restriction does not apply; however, we will impose it in the following example by giving the feature `subcat` the special value `aux` on auxiliaries.

In addition to the `np_gap` rule above we add the rules

```
s:[fin=Fin,gaps=([],[])] =>                      s_np_v_s-
     [np:[agr=Agr1,gaps=([],[]),wh=y],
      v:[agr=Agr2,fin=Fin,gaps=([],[]),subcat=aux],
      s:[fin=n,gaps=([v:[agr=Agr2,fin=Fin,subcat=aux],
                      np:[agr=Agr1,wh=y]|G]),G]].

vp:[agr=Agr,fin=Fin,gaps=(G0,G)] =>              vp_v_vp-
```

```
     [v:[agr=Agr,fin=Fin,gaps=(G0,G1),subcat=aux],
      vp:[agr=_,fin=n,gaps=(G1,G)]].

v:[agr=Agr,fin=n,
   gaps=([v:[agr=Agr,fin=_,subcat=SubCat]|G],G),
   subcat=SubCat] =>                           v_gap-
    [].
```

and modify the rest of the grammar accordingly, e.g:

```
top_symbol(s:[fin=y,gaps=([],[])]).

s:[fin=Fin,gaps=(G0,G)] =>                     s_np_vp-
     [np:[agr=Agr,gaps=([],[]),wh=n],
      vp:[agr=Agr,fin=Fin,gaps=(G0,G)]].

vp:[agr=Agr,fin=Fin,gaps=(G0,G)] =>            vp_v_np_np-
     [v:[agr=Agr,fin=Fin,gaps=(G0,G1),subcat=ditran],
      np:[agr=_,gaps=(G1,G2),wh=n],
      np:[agr=_,gaps=(G2,G),wh=n]].

vp:[agr=Agr,fin=Fin,gaps=(G0,G)] =>            vp_vp_pp-
     [vp:[agr=Agr,fin=Fin,gaps=(G0,G1)],
      pp:[gaps=(G1,G),lexform=_]].
```

The point is that the missing *NP* can be consumed by either the first or second *NP* of the `vp_v_np_np` rule, or by the *PP* of the `vp_vp_pp` rule, but only by one of them. This will cover sentences like the following

Whom$_i$ *did*$_j$ *John* ϵ_j *give* ϵ_i *a book in Paris?*
What$_i$ *did*$_j$ *John* ϵ_j *give Mary* ϵ_i *in Paris?*
(What city)$_i$ *did*$_j$ *John* ϵ_j *give Mary a book in* ϵ_i*?*

The implicit parse tree for the first example sentence is shown in Figure 2.22.

The feature value of the `gaps` feature is interesting from two points of view: Firstly, it is represented using a so called *difference list* `(L1,L2)` where the list `L2` is the tail of the list `L1` and the list represented is the list of elements preceding `L2` in `L1`. To give a simple example, the list `[a,b]` can be represented as the difference between the two lists `[a,b,c,d]` and `[c,d]`. In the example above `L1` is `[np:[agr=Agr,gaps=_]|G]` and `L2` is the Prolog variable `G`, so the represented list is `[np:[agr=Agr,gaps=_]]`.

Secondly, the feature value is complex in the sense that it may contain grammar phrases, the phrase `np:[agr=Agr,gaps=_]` in the example above.

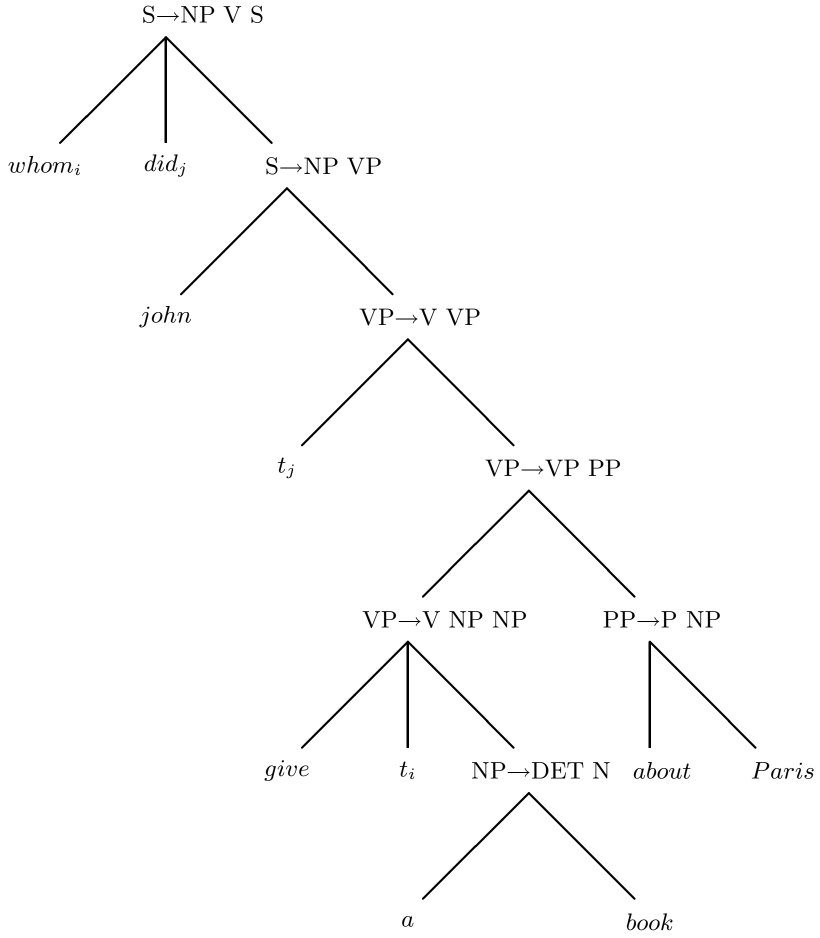

Figure 2.22: The parse tree for a WH-question

2.4.2 Lexicalized compliments

Another case where grammar phrases figure in the feature values is that of lexicalized complements, which is commonly used to keep the grammar as simple as possible by incorporating into the lexicon large quantities of information that has traditionally resided in the grammar rules. In Grammar 2 we could replace the three rules `vp_v`, `vp_v_n` and `vp_v_np_np` with a single rule (or rather, *rule schema*)

```
vp:[agr=Agr] =>                         vp_v_compl-
     [v:[agr=Agr,subcat=Complements]
     | Complements].
```

and instead specify the verb's complements in the lexicon, e.g:

```
lexicon(sees,v:[agr=sg,subcat=[np:[agr=_,wh=_]]]).
```

In this example, the *VP* inherits its structure from a favoured RHS phrase, namely the verb, that is the *head* of the verb-phrase. Quite a lot of features values tend to be shared between the head and LHS of a rule, and in GPSG there is a convention that says that unless otherwise specified, the values of the features that have been classified as *head features* should be passed on between the LHS and the head of the grammar rule without this necessarily being stated explicitly in the grammar rule.

For example, let the `lexform` feature, specifying the lexical item that the phrase is a projection of, be a head feature. The rule `pp_p_np` will then ensure that the value of `lexform` is the same for the *PP* and the preposition. This allows us to specify that a particular *PP* must start with some specific preposition, for example that `pp:[lexform=to]` must be realized as "*to NP*" for some *NP*.

Chapter 3

Natural-Language Parsing

The task of a syntactic parser is to use the grammar rules to either accept an input sentence as grammatical and produce some useful output structure (such as a parse tree), or to reject the input word-string as ungrammatical. It is important that this task is performed efficiently and in particular that the parsing process terminates.

The strategies a parser can implement differ along several lines. The parsing strategy can be *top-down* — goal driven, or it can be *bottom-up* — data driven. The order in which the RHS phrases of a rule are processed differ, parsing left-to-right, so-called *left-corner parsing*, being the most common.[1] Parsers also differ in the way hypothesized or previously parsed phrases are used to filter out spurious search branches, and what use they make of the internal memory, e.g., employing *well-formed-substring tables* and *charts*. Also the form of the transition relation, usually referred to as the *parsing tables*, varies.

3.1 Top-down parsing

A top-down parser tries to construct a parse tree starting from the root (i.e., the top symbol of the grammar) and expand it using the rules of the grammar. Figure 3.1 shows a simple parser implementing the parsing strategy of Prolog — top-down and left-to-right.

The predicate `parse/4` parses a phrase either by the phrase being a preterminal matching the next word in the sentence, or by applying a grammar rule whose left-hand side matches the phrase. The latter is called *top-down (rule) prediction* and is the essence of top-down parsing. The predicate `parse_rest/4` is a simple list recursion. The first argument is a phrase in `parse/4` and a list of phrases in `parse_rest/4`, the second two arguments constitute a difference

[1] Alternative strategies have been suggested by Kay [Kay 1989] and van Noord [van Noord 1991] (Head-corner parsing) and by Milne [Milne 1928] (Pooh-corner parsing). The former strategy is discussed in Section 3.7.

```
parse(Words,Tree) :-
   top_symbol(S),
   parse(S,Words,[],Tree).

parse(Phrase,[Word|Words],Words,lex-Word) :-
   lexicon(Word,Phrase).
parse(Phrase,Words0,Words,Id-Trees) :-
   (Phrase => Id-Body),
   parse_rest(Body,Words0,Words,Trees).

parse_rest([],Words,Words,[]).
parse_rest([Phrase|Rest],Words0,Words,[Tree|Trees]) :-
   parse(Phrase,Words0,Words1,Tree),
   parse_rest(Rest,Words1,Words,Trees).
```

Figure 3.1: A simple top-down parser

list denoting the portion of the input word string that the phrase spans, and the fourth argument is the output implicit parse tree(s) reflecting the structure of the analysis.

3.2 Well-formed-substring tables

If we for example want the grammar on Page 36 to cover the sentence *John gives a book to Mary*, we might add the following rule[2]

```
vp:[agr=Agr,tree=vp(V,NP1,pp(p(to),NP2))] =>    vp_v_np_pp-
    [v:[agr=Agr,subcat=dative,tree=V],
     np:[agr=_,tree=NP1],
     pp:[tree=pp(p(to),NP2)]].
```

and the following lexicon entries

```
lexicon(gives,v:[agr=sg,subcat=dative,tree=v(gives)]).
lexicon(give,v:[agr=pl,subcat=dative,tree=v(give)]).

lexicon(to,p:[tree=p(to)]).
```

The problem now is that the parser first will analyze “gives” as a ditransitive verb using the rule

[2] Here the parse tree of the PP is used in a way similar to the `lexform` feature to ensure that the preposition is realized as “to”.

```
vp:[agr=Agr,tree=vp(V,NP1,NP2)] =>     vp_v_np_np-
     [v:[agr=Agr,subcat=ditran,tree=V],
      np:[agr=_,tree=NP1],
      np:[agr=_,tree=NP2]].
```

When it fails to analyze “to Mary” as a second noun phrase it will resort to the other lexicon entry for “gives” and the new grammar rule. Despite already having analyzed “a book”as a noun phrase it will re-do this analysis, since that information was lost when back-tracking. To avoid this, all intermediate results are stored in a special table, a *well-formed-substring table*, which is consulted before parsing. If the ways of parsing a certain phrase starting in this position are already exhausted, the stored results are retrieved and no further parsing is attempted. This strategy is implemented in the parser of Figure 3.2.

```
parse(Words,Tree) :-
   reset_wfst,
   top_symbol(S),
   parse(S,Words,[],Tree).

parse(Phrase,Words0,Words,Tree) :-
   complete(Phrase,Words0),!,
   wfst(Phrase,Words0,Words,Tree).
parse(Phrase,Words0,Words,Tree) :-
   parse1(Phrase,Words0,Words,Tree),
   assert(wfst(Phrase,Words0,Words,Tree)).
parse(Phrase,Words0,_Words,_Tree) :-
   assert(complete(Phrase,Words0)),
   fail.

parse1(Phrase,[Word|Words],Words,lex-Word) :-
   lexicon(Word,Phrase).
parse1(Phrase,Words0,Words,Id-Trees) :-
   (Phrase => Id-Body),
   parse_rest(Body,Words0,Words,Trees).

parse_rest([],Words,Words,[]).
parse_rest([Phrase|Rest],Words0,Words,[Tree|Trees]) :-
   parse(Phrase,Words0,Words1,Tree),
   parse_rest(Rest,Words1,Words,Trees).
```

Figure 3.2: A parser employing a well-formed-substring table

Here the predicate `parse1/4` does the real parsing. As we can see, `parse/4` asserts the derived results into the Prolog database and their re-use is regulated by the predicate `complete/2`. The first clause is only applicable when an exhaustive search has been carried out. In this case the well-formed-substring table is consulted. Otherwise search is carried out by the second clause through the call to `parse1` and the result stored in the well-formed-substring table. When no more solutions can be found by clause two, Prolog fails into clause three and the flag `complete` is set.

If both `wfst/4` and `complete/2` were defined to be dynamic predicates, the predicate `reset_wfst/0` could be defined simply as

```
reset_wfst :-
   retractall(wfst(_,_,_,_)),
   retractall(complete(_,_)).
```

Now the parser will not re-do analyses unnecessarily. Whether or not using a well-formed-substring table pays off from an efficiency point-of-view is in general an empirical question.

3.3 Bottom-up parsing

The `vp_vp_pp` and `np_np_pp` rules causing the ambiguity in the sentence *He sees a book about Paris* above are interesting from another point of view: They are recursive (since the LHS unifies with a phrase of the RHS), and are the simplest form of cyclic derivations. Cyclic derivations are a constant source to non-termination problems. The problem with rules like `np_np_pp` in conjunction with top-down parsing is that if they are applicable once, they are applicable an infinite number of times, building a left-branching parse tree where the right-hand side NP of one incarnation of the rule is unified with the left-hand side of another. For this and other reasons, parsing is often performed bottom-up instead.

A bottom-up parser tries to construct a parse tree starting from the leaves (i.e., from the words of the input sentence) and combine these using the rules of the grammar. The bottom-up parser described here is due to the one in [Pereira & Shieber 1987] which in turn is based on Matsumoto's original parser, "BUP" presented in [Matsumoto *et al* 1983].

In Figure 3.3 we have modified the `parse/4` predicate to instead parse bottom-up. This predicate now parses a phrase by first looking up a preterminal corresponding to the next word in the input string using `leaf/4`. It then tries to connect the preterminal with the predicted phrase using `connect/6`. The first clause of this predicate is the base case stating that each phrase is connected to itself. The second clause tries to connect the `SubPhrase` with the `SuperPhrase` by invoking a grammar rule where the first RHS phrase unifies with the `SubPhrase`. This is called *bottom-up (rule) prediction* and is the essence of bottom-up parsing. It then parses the remaining RHS phrases with `parse_rest/4`, which again is a simple list recursion, before attempting to connect the LHS of the rule with the `SuperPhrase`.

```
parse(Words,Tree) :-
   top_symbol(S),
   parse(S,Words,[],Tree).

parse(Phrase,Words0,Words,Tree) :-
   leaf(SubPhrase,Words0,Words1,SubTree),
   connect(SubPhrase,Phrase,Words1,Words,SubTree,Tree).

leaf(Phrase,[Word|Words],Words,lex-Word) :-
   lexicon(Word,Phrase).

connect(Phrase,Phrase,Words,Words,Tree,Tree).
connect(SubPhrase,SuperPhrase,Words0,Words,SubTree,Root) :-
   (Phrase => Id-[SubPhrase|Rest]),
   parse_rest(Rest,Words0,Words1,Trees),
   Tree = Id-[SubTree|Trees],
   connect(Phrase,SuperPhrase,Words1,Words,Tree,Root).

parse_rest([],Words,Words,[]).
parse_rest([Phrase|Phrases],Words0,Words,[Tree|Trees]) :-
   parse(Phrase,Words0,Words1,Tree),
   parse_rest(Phrases,Words1,Words,Trees).
```

Figure 3.3: A simple bottom-up parser

3.4 Link tables

It is possible to save some search time by in advance working out what can potentially be a left-corner of what and maintain this in a table, which for historical reasons is called either a *link table*, *viable prefix table* or *reachability table*. The parser in Figure 3.4 uses the table to avoid attempting to construct phrases that cannot start the (super)phrase being sought for. This is called *top-down filtering*.

Empty production rules constitute a notorious problem for parser developers since the LHSs of these grammar rules have no realization in the input word-string, their RHSs being empty. This means that when parsing strictly bottom-up, they are always applicable, and a bottom-up parser can easily get stuck hallucinating an infinite number of empty phrases at some point in the input word-string. This problem does not occur when parsing top-down.

Using gap-threading to limit the applicability of empty productions, as described in Section 2.2.2, in conjunction with a link table is a type of top-down filtering. Top-down filtering is typically used in conjunction with bottom-up parsing strategies to prune the search space and avoid non-termination.

```
parse(Words,Tree) :-
   top_symbol(S),
   parse(S,Words,[],Tree).

parse(Phrase,Words0,Words,Tree) :-
   leaf(SubPhrase,Words0,Words1,SubTree),
   link(SubPhrase,Phrase),
   connect(SubPhrase,Phrase,Words1,Words,SubTree,Tree).

leaf(Phrase,[Word|Words],Words,lex-Word) :-
   lexicon(Word,Phrase).

connect(Phrase,Phrase,Words,Words,Tree,Tree).
connect(SubPhrase,SuperPhrase,Words0,Words,SubTree,Root) :-
   (Phrase => Id-[SubPhrase|Rest]),
   link(Phrase,SuperPhrase),
   parse_rest(Rest,Words0,Words1,Trees),
   Tree = Id-[SubTree|Trees],
   connect(Phrase,SuperPhrase,Words1,Words,Tree,Root).

parse_rest([],Words,Words,[]).
parse_rest([Phrase|Phrases],Words0,Words,[Tree|Trees]) :-
   parse(Phrase,Words0,Words1,Tree),
   parse_rest(Phrases,Words1,Words,Trees).

link(s:[tree=s(A,B)], s:[tree=s(A,B)]).
link(np:[agr=_,tree=_], s:[tree=s(_,_)]).
link(det:[agr=_,tree=_], s:[tree=s(_,_)]).
link(pron:[agr=_,tree=_], s:[tree=s(_,_)]).
link(v:[agr=B,subcat=C,tree=A], v:[agr=B,subcat=C,tree=A]).
link(vp:[agr=A,tree=_], vp:[agr=A,tree=_]).
link(v:[agr=A,subcat=_,tree=_], vp:[agr=A,tree=_]).
link(det:[agr=B,tree=A], det:[agr=B,tree=A]).
link(n:[agr=B,tree=A], n:[agr=B,tree=A]).
link(pron:[agr=B,tree=A], pron:[agr=B,tree=A]).
link(pp:[tree=A], pp:[tree=A]).
link(p:[tree=A], pp:[tree=pp(A,_)]).
link(p:[tree=A], p:[tree=A]).
link(np:[agr=A,tree=_], np:[agr=A,tree=_]).
link(det:[agr=A,tree=_], np:[agr=A,tree=_]).
link(pron:[agr=A,tree=_], np:[agr=A,tree=_]).
```

Figure 3.4: A parser employing a link table

3.5 Shift-reduce parsers

A *shift-reduce parser* is a parser that in each cycle performs one of the two actions *shift* and *reduce*. The shift actions consume a word from the input sentence and the reduce actions apply a grammar rule. LR parsers (discussed in Section 3.8) are a kind of shift-reduce parser.

We will here describe a simple shift-reduce parser. The parser implements a left-to-right, bottom-up parsing strategy and employs a *rule invocation* as its current working hypothesis. This consists of a *goal* corresponding to the LHS of the grammar rule and a *body* corresponding to those phrases of the RHS that remain to be found. The rule invocations are represented as

```
edge(Goal,Body,Tree)
```

The reasons behind the choice of functor will become apparent later.

Since the parser can temporarily switch working hypothesis, we will use a pushdown stack to store rule invocations. The current rule invocation will simply be the one on top of the stack. Thus the parser is a type of pushdown automaton, even though strictly speaking it is not one according to the formal definition of Section 2.1.2, since the reduce actions do not consume any input symbols. This is not important from a theoretical point of view, since it can be proved that this does not change the expressive power of the machine. Likewise the use of complex stack objects, namely the rule invocations, does not in this case affect the expressiveness of the automaton, but improves readability. A theoretically more important objection is that the symbols of the alphabets may be feature-based. The automaton only has a single internal state corresponding to a call to the predicate `shift_or_reduce/3`.

```
parse(Sentence,Tree) :-
   empty_stack(Stack),
   top_symbol(TopSymbol),
   (TopSymbol => Id-Body),
   Edge = edge(TopSymbol,Body,Id-[]),
   push(Stack,Edge,Stack1),
   shift_or_reduce(Sentence,Stack1,Tree).

shift_or_reduce(Words,Stack,Tree) :-
   shift(Words,Stack,Tree).
shift_or_reduce(Words,Stack,Tree) :-
   reduce(Words,Stack,Tree).
```

The shift action consumes an input word and produces a corresponding preterminal as a new phrase. The reduce action pops off the current rule invocation from the stack, producing the LHS of the rule as a new phrase in the process. Reductions are only applicable to rule invocations where the body is empty.

```
shift([Word|Words],Stack,Tree) :-
   lexicon(Word,NewPhrase),
   top_down_filter(NewPhrase,Stack),
   predict_or_match(NewPhrase,Words,Stack,lex-Word,Tree).

reduce([],Stack,Tree) :-
   pop(Stack,Edge,Stack1),
   Edge = edge(_,[],Tree0),
   empty_stack(Stack1),!,
   Tree0 = Tree.
reduce(Words,Stack,Tree) :-
   pop(Stack,Edge,Stack1),
   Edge = edge(Goal,[],Tree0),
   predict_or_match(Goal,Words,Stack1,Tree0,Tree).
```

The new phrase is either matched against the first phrase in the body or used to predict a new rule invocation. In the latter case the first phrase of the body is used for top-down filtering and the new rule invocation is pushed onto the stack.

```
predict_or_match(NewPhrase,Words,Stack,NewTree,Tree) :-
   predict(NewPhrase,Words,Stack,NewTree,Tree).
predict_or_match(NewPhrase,Words,Stack,NewTree,Tree) :-
   match(NewPhrase,Words,Stack,NewTree,Tree).

predict(NewPhrase,Words,Stack,NewTree,Tree) :-
   (NewGoal => Id-[NewPhrase|Rest]),
   top_down_filter(NewGoal,Stack),
   push(Stack,edge(NewGoal,Rest,Id-[NewTree]),Stack1),
   shift_or_reduce(Words,Stack1,Tree).

match(NewPhrase,Words,Stack,NewTree,Tree) :-
   pop(Stack,Edge,Stack1),
   Edge = edge(Goal,[NewPhrase|Body],Tree0),
   fix_trees(Tree0,NewTree,Tree1),
   Edge1 = edge(Goal,Body,Tree1),
   push(Stack1,Edge1,Stack2),
   shift_or_reduce(Words,Stack2,Tree).
```

Some auxiliary predicates:

```
top_down_filter(NewPhrase,Stack) :-
   top(Stack,Edge),
   Edge = edge(_,Body,_),
   Body = [Phrase|_],
   link(NewPhrase,Phrase).
```

```
fix_trees(Tree,SubTree,Tree1) :-
   Tree = Id-Trees,
   append(Trees,[SubTree],Trees1),
   Tree1 = Id-Trees1.

%%% Stack Predicates

empty_stack([]).
top([Item|_Stack],Item).
pop([Item|Stack],Item,Stack).
push(Stack,Item,[Item|Stack]).
```

By extending the `shift_or_reduce/3` predicate with a clause for handling empty productions we would arrive at a parser implementing the basic parsing strategy of the Core Language Engine [Alshawi (ed.) 1992].

3.6 Chart parsers

The chart-parsing paradigm is due to Martin Kay [Kay 1980] and based on Earley deduction [Earley 1969]. A very nice overview of various chart-parsing strategies is given in Mats Wirén's PhD dissertation [Wirén 1992].

A chart parser stores partial results in a table called the chart. This is an extension of the idea of storing completed intermediate results in a well-formed-substring table, as discussed in Section 3.2. The chart entries are edges between various positions in the sentence marked with phrases. There are "passive" and "active" edges. Let us call a phrase spanning two positions in the input word string a goal:

1. Passive edges correspond to proven goals or facts; one knows that there is a phrase of the type indicated by the edge between its starting and end points.

2. Active edges correspond to unproven goals; one can find the type of phrase indicated by the edge if one can prove the remaining subgoals. These subgoals are specified by the edge.

The parse is completed when one has constructed a passive edge between the starting point and the end point of the sentence marked with the top symbol of the grammar.

The similarity with the rule invocations of the shift-reduce parser of the previous section is not coincidental, nor is the choice of "edge" as the name of the stack elements for that parser. The goal of an edge corresponds to the goal of a rule invocation and the remaining subgoals of an edge to the body of a rule invocation.

Now we describe a chart parser based on that of [Pereira & Shieber 1987]. We will represent the positions in the sentence explicitly. Apart from this, the grammar and lexicon of Figures 3.5 and 3.6 are the same as Grammar 2 and Lexicon 2 in Figures 2.19 and 2.20 of Section 2.2.2 respectively.

We will use the same notation for edges here as for the shift-reduce parser. `edge(Goal,[],Tree)` are passive edges corresponding to proven goals. The active edges are `edge(Goal,Body,Tree)`, where `Body` is a non-empty list of subgoals that remain to be proven. We have succeeded when we have added the edge `edge(s(0,End):Feats,[],Tree)` to the chart.

```
parse(Sentence,Tree) :-
   clear_chart,
   lexical_analysis(Sentence,End),
   top_symbol(S,End),
   prove(S,Tree).

prove(Goal,Tree) :-
   predict(Goal,Agenda),
   process(Agenda),
   edge(Goal,[],Tree).
```

The parser first clears the chart (i.e., removes all occurences of the dynamic predicate `edge/3`), looks up the words of the input sentence in the lexicon, and adds the corresponding preterminals as passive edges to the chart. The sentence *John sees a house*, for example, would be added to the chart as the passive edges

```
edge(np(0,1):[agr=sg,tree=np(john)], [], lex-john).
edge(v(1,2):[agr=sg,subcat=tran,tree=v(sees)], [], lex-sees).
edge(det(2,3):[agr=sg,tree=det(a)], [], lex-a).
edge(n(3,4):[agr=sg,tree=n(house)], [], lex-house).
```

The parsing process is driven by an agenda that keeps track of new edges added to the chart, which remain to be processed. When processing an edge in the agenda, different measures are taken depending on whether the edge is active or passive.

```
process([]).
process([edge(Goal,Body,Tree)|OldAgenda]) :-
   process_one(Goal,Body,Tree,SubAgenda),
   append(SubAgenda,OldAgenda,Agenda),
   process(Agenda).

process_one(Goal,[],Tree,Agenda) :-
   resolve_passive(Goal,Tree,Agenda).
process_one(Goal,[First|Body],Tree,Agenda) :-
   predict(First,Front),
   resolve_active(edge(Goal,[First|Body],Tree),Back),
   append(Front,Back,Agenda).
```

```
top_symbol(s(0,End):[tree=_],End).

s(P0,P):[tree=s(NP,VP)] =>                  s_np_vp-
     [np(P0,P1):[agr=Agr,tree=NP],
      vp(P1,P):[agr=Agr,tree=VP]].

vp(P0,P):[agr=Agr,tree=vp(V)] =>            vp_v-
     [v(P0,P):[agr=Agr,subcat=intran,tree=V]].

vp(P0,P):[agr=Agr,tree=vp(V,NP)] =>         vp_v_np-
     [v(P0,P1):[agr=Agr,subcat=tran,tree=V],
      np(P1,P):[agr=_,tree=NP]].

vp(P0,P):[agr=Agr,tree=vp(V,NP1,NP2)] =>    vp_v_np_np-
     [v(P0,P1):[agr=Agr,subcat=ditran,tree=V],
      np(P1,P2):[agr=_,tree=NP1],
      np(P2,P):[agr=_,tree=NP2]].

vp(P0,P):[agr=Agr,tree=vp(VP,PP)] =>        vp_vp_pp-
     [vp(P0,P1):[agr=Agr,tree=VP],
      pp(P1,P):[tree=PP]].

np(P0,P):[agr=Agr,tree=np(DET,N)] =>        np_det_n-
     [det(P0,P1):[agr=Agr,tree=DET],
      n(P1,P):[agr=Agr,tree=N]].

np(P0,P):[agr=Agr,tree=np(PRON)] =>         np_pron-
     [pron(P0,P):[agr=Agr,tree=PRON]].

np(P0,P):[agr=Agr,tree=np(NP,PP)] =>        np_np_pp-
     [np(P0,P1):[agr=Agr,tree=NP],
      pp(P1,P):[tree=PP]].

pp(P0,P):[tree=pp(P,NP)] =>                 pp_p_np-
     [p(P0,P1):[tree=P],
      np(P1,P):[agr=_,tree=NP]].
```

Figure 3.5: The chart-parser version of Grammar 2

```
lexicon(a,det(_,_):[agr=sg,tree=det(a)]).
lexicon(the,det(_,_):[agr=_,tree=det(the)]).
lexicon(several,det(_,_):[agr=pl,tree=det(several)]).

lexicon(about,p(_,_):[tree=p(about)]).
lexicon(in,p(_,_):[tree=p(in)]).

lexicon(paris,np(_,_):[agr=sg,tree=np(paris)]).
lexicon(john,np(_,_):[agr=sg,tree=np(john)]).
lexicon(mary,np(_,_):[agr=sg,tree=np(mary)]).

lexicon(he,pron(_,_):[agr=sg,tree=pron(he)]).
lexicon(she,pron(_,_):[agr=sg,tree=pron(she)]).
lexicon(they,pron(_,_):[agr=pl,tree=pron(they)]).

lexicon(book,n(_,_):[agr=sg,tree=n(book)]).
lexicon(books,n(_,_):[agr=pl,tree=n(books)]).

lexicon(house,n(_,_):[agr=sg,tree=n(house)]).
lexicon(houses,n(_,_):[agr=pl,tree=n(houses)]).

lexicon(sleeps,v(_,_):[agr=sg,subcat=intran,tree=v(sleeps)]).
lexicon(sleep,v(_,_):[agr=pl,subcat=intran,tree=v(sleep)]).

lexicon(gives,v(_,_):[agr=sg,subcat=ditran,tree=v(gives)]).
lexicon(give,v(_,_):[agr=pl,subcat=ditran,tree=v(give)]).

lexicon(sees,v(_,_):[agr=sg,subcat=tran,tree=v(sees)]).
lexicon(see,v(_,_):[agr=pl,subcat=tran,tree=v(see)]).
```

Figure 3.6: The chart parser version of Lexicon 2

A passive edge in the agenda corresponds to a new fact, and thus the active edges in the chart are compared with this fact to see if it matches one of their remaining goals.

```
resolve_passive(Fact,Tree,Agenda) :-
   findall(Edge,
           Fact^p_resolution(Fact,Tree,Edge),
           Agenda).

p_resolution(Fact,SubTree,Edge) :-
   edge(Goal,[Fact|Body],Tree),
   fix_trees(Tree,SubTree,Tree1),
   Edge = edge(Goal,Body,Tree1),
   store(Edge).
```

An active edge has goals that remain to be proven. One way of doing this is to invoke grammar rules that are potentially useful for proving them.

```
predict(Goal,Agenda) :-
   findall(Edge,
           Goal^prediction(Goal,Edge),
           Agenda).

prediction(Goal,Edge) :-
   (Goal => Id-Body),
   Edge = edge(Goal,Body,Id-[]),
   store(Edge).
```

Here rules are chosen where the goal matches the left-hand side of the rule — a top-down strategy.

Another way of proving the remaining goals of an active edge is to search the chart for passive edges, i. e. facts, that match them.

```
resolve_active(Edge,Agenda) :-
   findall(NewEdge,
           Edge^a_resolution(Edge,NewEdge),
           Agenda).

a_resolution(edge(Goal,[First|Body],Tree),Edge) :-
   edge(First,[],SubTree),
   fix_trees(Tree,SubTree,Tree1),
   Edge = edge(Goal,Body,Tree1),
   store(Edge).
```

We only want to add truly new edges to the chart. For this reason we must check if the edge, or a more general one, has already been added to the chart. This is what `subsumed/1` does.

```
store(Edge) :-
   \+ subsumed(Edge),
   assert(Edge).

subsumed(Edge) :-
   edge(GenGoal,GenBody,_),
   subsumes(edge(GenGoal,GenBody,_),Edge).

subsumes(General,Specific) :-
   \+ \+ (make_ground(Specific),
          General = Specific).

make_ground(Term) :-
   numbervars(Term,0,_).
```

Actually, the subsumes check is a built-in predicate in SICStus Prolog called `subsumes_chk/2` that could be used instead.

3.7 Head parsers

Most grammar formalisms, and thus most parsers, assume that string production is a question of concatenating elements. This assumption may be a consequence of the fact that most of the work in computational linguistics has focused on a small family of languages with a relatively constrained word order.

There are excellent arguments for questioning this assumption. “Our Latin teachers were apparently right”, as Martin Kay puts it [Kay 1989]. Latin is a language with relatively free word order, and as most Latin students know, it is good practice when parsing a Latin sentence to search for the main verb first. It carries inflectional information which will aid in determining the other constituents of the sentence. In the general case, languages are not concatenative, constituents are not necessarily continuous, and dependencies in a clause are not necessarily neatly stacked.

If we wish to generalize the representation we have been working with so far, we can start by examining a normal context free grammar, augmented with feature structures, such as the ones we have been making use of throughout this text. For each rule, a certain element of the right hand side is designated the *head* of the phrase. The parser should look for the head first, and then try to extend the left and right contexts of the head. The features of the head constituent percolate up to the root node, and are then a powerful top-down prediction tool when extending the context.

A central question when writing a head grammar is how the head is chosen from the right-hand constituents available. It is of course appropriate to pick a head which distinguishes the right-hand side well. The problem of designating the head is usually not addressed in the theory — the grammar writer is assumed to have good intuitions.

An avenue of approach which has not been explored is constructing an algorithm whereby a head is picked automatically. This may be difficult: the element on the right-hand side with the highest informational content is the natural candidate, but this may be distorted through lop-sided application statistics on the grammar rules and can thus not be determined by looking at the grammar alone.

In a feature structure augmented grammar it may even be a complex matter to determine which element has the highest informational value, depending on the feasibility of factoring out all the features and their value space.

A head grammar version of Grammar 2 (from Page 36) is shown in Figure 3.7, with the heads of different types of phrases stated explicitly.

Given a head grammar the generalization from previous parsing algorithms can be stated in terms of link tables: instead of a link table which indicates which root a certain leftmost element in a string can *start*, a head relation indicates which root a certain element in a string can *motivate*. This element is called the *seed* of a phrase.

The parser, as most bottom-up parsers, proceeds from head to head, starting from the seed, until it has proven that a seed can indeed be the seed of a phrase.

In the code for the parser below (see Figure 3.8), which is a simple reversible parser/generator by Martin Kay [Kay 1989], the basic procedure is to, given a string, assume a goal and a head for the goal. This is done in the `syntax/3` predicate.

Determining that the head is in the string is done using the `range/3` predicate, which steps through the string until it finds an element in it which unifies with the description of a head. If no such element is found, another head is tried. If the head is found, its range will be in HeadRange. After finding the head, the goal is built around it.

A goal is built using the `connect/5` predicate. A rule which connects the head with the goal is identified, and the left and right contexts of the rule are built, so that the LHS of the rule extends from LL to RR in the string. `connect/5` is invoked recursively until the goal is complete.

This parser and grammar formalism does not allow for other than concatenative rules. The difference between this parser and the bottom-up parser discussed in Section 3.3 is only the more flexible linking. To generalize to nonconcatenative grammars we need to change the formalism somewhat.

If we change the rule formalism from

```
LHS => LeftContext,Head,RightContext
```

to

```
LHS => Head,OtherDaughters
```

and generalize the two predicates `connect_left/3` and `connect_right/3` to one `connect_others/3`, as per below, we get a more general framework.

The other daughters in the `OtherDaughters` list have information on in which direction they are to be sought, or on which side they are to be concatenated to the head.

```
top_symbol(s:[tree=_]).

s:[tree=s(NP,VP)] =>
     [np:[agr=Agr,tree=NP]],
     vp:[agr=Agr,tree=VP],
     [].

vp:[agr=Agr,tree=vp(V)] =>
     [],v:[agr=Agr,subcat=intran,tree=V],
     [].
vp:[agr=Agr,tree=vp(V,NP)] =>
     [],v:[agr=Agr,subcat=tran,tree=V],
     [np:[agr=_,tree=NP]].
vp:[agr=Agr,tree=vp(V,NP1,NP2)] =>
     [],v:[agr=Agr,subcat=ditran,tree=V],
     [np:[agr=_,tree=NP1],np:[agr=_,tree=NP2]].
vp:[agr=Agr,tree=vp(VP,PP)] =>
     [],vp:[agr=Agr,tree=VP],
     [pp:[tree=PP]].

np:[agr=Agr,tree=np(DET,N)] =>
     [det:[agr=Agr,tree=DET]],
     n:[agr=Agr,tree=N],
     [].
np:[agr=Agr,tree=np(PRON)] =>
     [],pron:[agr=Agr,tree=PRON],
     [].
np:[agr=Agr,tree=np(NP,PP)] =>
     [],np:[agr=Agr,tree=NP],
     [pp:[tree=PP]].

pp:[tree=pp(P,NP)] =>
     [],p:[tree=P],
     [np:[agr=_,tree=NP]].

head(s:_,vp:_).
head(s:_,v:_).
head(np:_,n:_).
head(pp:_,p:_).
head(K,K).
```

Figure 3.7: Head Grammar Version of Grammar 2

```
parse(String,Struct) :-
   syntax(String/[],Struct,String/[]).

syntax(GoalRange,Goal,MaxRange) :-
   head(Goal,Head),
   range(HeadRange,Head,MaxRange),
   connect(GoalRange,Goal,HeadRange,Head,MaxRange).

range(_,_,X/Y) :-
   nonvar(X),
   X=Y,
   fail.
range(L/R,Head,L1/_) :-
   (    var(L1), !
   ;    L = L1
   ),
   dict(L/R, Head).
range(HRange,Head,MaxL/MaxR) :-
   nonvar(MaxL),
   MaxL = [_|T],
   range(HRange,Head,T/MaxR).

connect(R,G,R,G,_).
connect(GL/GR,Goal,HL/HR,Head,MaxL/MaxR) :-
   (LHS => Left,Head,Right),
   head(Goal,LHS),
   connect_left(LL/HL,Left,MaxL/HL),
   connect_right(HR/RR,Right,HR/MaxR),
   connect(GL/GR,Goal,LL/RR,LHS,MaxL/MaxR).

connect_left(X/X,[],_).
connect_left(L/R,[Head|Heads],MaxL/MaxR) :-
   syntax(HL/R,Head,MaxL/MaxR),
   connect_left(L/HL,Heads,MaxL/HL).

connect_right(X/X,[],_).
connect_right(L/R,[Head|Heads],MaxL/MaxR) :-
   syntax(L/HR,Head,MaxL/MaxR),
   connect_right(HR/R,Heads,HR/MaxR).

dict([K|X]/X,L) :- lexicon(K,L).
```

Figure 3.8: A Head Driven Parser/Generator

The example in Figure 3.9 still only handles concatenative grammars, and is equivalent to the example above, but if we add new `connect_others/3` clauses we will be able to build more complex constituents, and allow for interesting ways of coping with movement and discontinuous constituents. This is reminiscent of the ID/LP grammar rule paradigm referred to in Section 2.2.1. For examples of how this is done and how it can be put to use, we refer to [van Noord 1991].

```
connect(R,G,R,G,_).
connect(GL/GR,Goal,HL/HR,Head,MaxL/MaxR) :-
   (LHS => Head,Others),
   head(Goal,LHS),
   connect_others(Others,LL/HL,HR/RR),
   connect(GL/GR,Goal,LL/RR,LHS,MaxL/MaxR).

connect_others([],L/L,R/R).
connect_others([LeftDaughter:direction=l:Features|Others],
                MaxL/MaxR,RightContext) :-
   syntax(NewL/MaxR,LeftDaughter:Features,MaxL/MaxR),
   connect_others(Others,MaxL/NewL,RightContext).
connect_others([RightDaughter:direction=r:Features|Others],
               LeftContext,MaxL/MaxR) :-
   syntax(MaxL/NewR,RightDaughter:Features,MaxL/MaxR),
   connect_others(Others,LeftContext,NewR/MaxR).
```

Figure 3.9: Generalized connect_others/3

The bottom line question in practical natural-language processing naturally will be: Are head grammars a good idea or not in terms of actually analyzing natural languages? There is no answer to this question as of yet. The jury is still out. It will probably depend heavily on the language that is being analyzed.

3.8 LR parsers

An LR parser is a type of shift-reduce parser that was originally devised by Donald Knuth [Knuth 1965] for programming languages and is well described in e.g., [Aho *et al* 1986]. Since Tomita's article [Tomita 1986] it has become increasingly popular for natural languages.

The success of LR parsing lies in handling a number of grammar rules simultaneously by the use of prefix merging, rather than attempting one rule at a time as the shift-reduce parser of Section 3.5 does.

3.8.1 Parsing

An LR parser is basically a pushdown automaton, i.e., it has a pushdown stack in addition to a finite set of internal states and a reader head for scanning the input string from left to right one symbol at a time. Thus an LR parser employs a left-corner parsing strategy. In fact, the "L" in "LR" stands for left-to-right scanning of the input. The "R" stands for constructing the rightmost derivation in reverse.

The stack is used in a characteristic way: Every other item on the stack is a grammar symbol and every other is a state. The current state is simply the state on top of the stack.

The most distinguishing feature of an LR parser is however the form of the transition relation — the action and goto tables: a (non-deterministic) LR parser can in each step perform one of four basic actions. In state `S` with lookahead symbol[3] `Sym` it can:

1. `accept(S,Sym)`: Halt and signal success.

2. `error(S,Sym)`: Fail and backtrack.

3. `shift(S,Sym,S2)`: Consume the input symbol `Sym`, place it on top of the stack, and transit to state `S2` by placing it on top of the stack.

4. `reduce(S,Sym,R)`: Pop off a number of items from the stack corresponding to the RHS of grammar rule `R`, inspect the stack for the old state `S1`, place the `LHS` of rule `R` on the stack, and transit to state `S2` determined by `goto(S1,LHS,S2)` by placing `S2` on the stack.

Like the shift-reduce parser of Section 3.5, this is not a pushdown automaton according to our definition above, since the reduce actions do not consume any input symbols, and the same remarks apply here. This gives the parser the possibility of not halting by reducing empty productions and transiting back to the same state, and care must be taken to avoid this.

Prefix merging is accomplished by each internal state corresponding to a set of partially processed grammar rules, so-called "dotted items" containing a dot $(\cdot)$ to mark the current position.

For example, if the grammar contains the following two rules,

$$\begin{array}{lcl} NP & \rightarrow & Det\ Noun \\ NP & \rightarrow & Det\ Adj\ Noun \end{array}$$

[3] The lookahead symbol is the next symbol in the input string i.e., the symbol under the reader head.

there will be a state containing the dotted items

$$\begin{array}{lcl} NP & \rightarrow & Det \cdot Noun \\ NP & \rightarrow & Det \cdot Adj\ Noun \end{array}$$

This state corresponds to just having found a determiner (*Det*). Which of the two rules to apply in the end will be determined by the rest of the input string; at this point no commitment has been made to either of the two rules.

The concept of dotted items is closely related to that of rule invocations in Section 3.5 and to that of edges in Section 3.6. The LHS of a dotted item corresponds to the goal of an edge and the phrases following the dot correspond to the body.

3.8.2 Parsed example

Grammar 1, reproduced in Figure 3.10 for reference, will generate the internal states of Figure 3.11.

$$\begin{array}{lcll} S & \rightarrow & NP\ VP & (1) \\ VP & \rightarrow & V & (2) \\ VP & \rightarrow & V\ NP & (3) \\ VP & \rightarrow & V\ NP\ NP & (4) \\ VP & \rightarrow & VP\ PP & (5) \\ NP & \rightarrow & Det\ N & (6) \\ NP & \rightarrow & Pron & (7) \\ NP & \rightarrow & NP\ PP & (8) \\ PP & \rightarrow & P\ NP & (9) \end{array}$$

Figure 3.10: Grammar 1

These in turn give rise to the parsing tables of Figure 3.12. The entry "s2" in the action table, for example, should be interpreted as "shift the lookahead symbol onto the stack and transit to State 2". The action entry "r7" should be interpreted as "reduce by Rule 7". The goto entries simply indicate what state to transit to once a phrase of that type has been constructed.

Note the two possibilities in States 11, 12 and 13 for lookahead symbol "P". We can either shift it onto the stack or perform a reduction. This is called a *shift-reduce conflict* and is the source to the ambiguity previously observed.

State 0

$$\begin{array}{lll} S' & \rightarrow & \cdot\ S \\ S & \rightarrow & \cdot\ NP\ VP \\ NP & \rightarrow & \cdot\ Det\ N \\ NP & \rightarrow & \cdot\ Pron \\ NP & \rightarrow & \cdot\ NP\ PP \end{array}$$

State 1

$$\begin{array}{lll} S & \rightarrow & NP\ \cdot\ VP \\ NP & \rightarrow & NP\ \cdot\ PP \\ VP & \rightarrow & \cdot\ V \\ VP & \rightarrow & \cdot\ V\ NP \\ VP & \rightarrow & \cdot\ V\ NP\ NP \\ VP & \rightarrow & \cdot\ VP\ PP \\ PP & \rightarrow & \cdot\ P\ NP \end{array}$$

State 2

$$\begin{array}{lll} NP & \rightarrow & Det\ \cdot\ N \end{array}$$

State 3

$$\begin{array}{lll} NP & \rightarrow & Pron\ \cdot \end{array}$$

State 4

$$\begin{array}{lll} S' & \rightarrow & S\ \cdot \end{array}$$

State 5

$$\begin{array}{lll} S & \rightarrow & NP\ VP\ \cdot \\ VP & \rightarrow & VP\ \cdot\ PP \\ PP & \rightarrow & \cdot\ P\ NP \end{array}$$

State 6

$$\begin{array}{lll} VP & \rightarrow & V\ \cdot \\ VP & \rightarrow & V\ \cdot\ NP \\ VP & \rightarrow & V\ \cdot\ NP\ NP \\ NP & \rightarrow & \cdot\ Det\ N \\ NP & \rightarrow & \cdot\ Pron \\ NP & \rightarrow & \cdot\ NP\ PP \end{array}$$

State 7

$$\begin{array}{lll} NP & \rightarrow & NP\ PP\ \cdot \end{array}$$

State 8

$$\begin{array}{lll} PP & \rightarrow & P\ \cdot\ NP \\ NP & \rightarrow & \cdot\ Det\ N \\ NP & \rightarrow & \cdot\ Pron \\ NP & \rightarrow & \cdot\ NP\ PP \end{array}$$

State 9

$$\begin{array}{lll} VP & \rightarrow & VP\ PP\ \cdot \end{array}$$

State 10

$$\begin{array}{lll} NP & \rightarrow & Det\ N\ \cdot \end{array}$$

State 11

$$\begin{array}{lll} VP & \rightarrow & V\ NP\ \cdot \\ VP & \rightarrow & V\ NP\ \cdot\ NP \\ NP & \rightarrow & NP\ \cdot\ PP \\ NP & \rightarrow & \cdot\ Det\ N \\ NP & \rightarrow & \cdot\ Pron \\ NP & \rightarrow & \cdot\ NP\ PP \\ PP & \rightarrow & \cdot\ P\ NP \end{array}$$

State 12

$$\begin{array}{lll} VP & \rightarrow & V\ NP\ NP\ \cdot \\ NP & \rightarrow & NP\ \cdot\ PP \\ PP & \rightarrow & \cdot\ P\ NP \end{array}$$

State 13

$$\begin{array}{lll} PP & \rightarrow & P\ NP\ \cdot \\ NP & \rightarrow & NP\ \cdot\ PP \\ PP & \rightarrow & \cdot\ P\ NP \end{array}$$

Figure 3.11: The internal states of Grammar 1

State	Action							Goto			
	Det	N	NP	P	Pron	V	eos	NP	PP	S	VP
0	s2		s1		s3			1		4	
1				s8		s6			7		5
2		s10									
3	r7		r7	r7	r7	r7	r7				
4							acc				
5				s8			r1		9		
6	s2		s11	r2	s3		r2	11			
7	r8		r8	r8	r8	r8	r8				
8	s2		s13		s3			13			
9				r5			r5				
10	r6		r6	r6	r6	r6	r6				
11	s2		s12	s8/r3	s3		r3	12	7		
12				s8/r4			r4		7		
13	r9		r9	s8/r9	r9	r9	r9				

Figure 3.12: The LR parsing tables for Grammar 1

Using these tables we can parse the sentence *John sees a book* as follows:

Action	Stack	String
init	[0]	*John sees a book*
s1	[1, NP, 0]	*sees a book*
s6	[6, V, 1, NP, 0]	*a book*
s2	[2, Det, 6, V, 1, NP, 0]	*book*
s10	[10, N, 2, Det, 6, V, 1, NP, 0]	ϵ
r6	[11, NP, 6, V, 1, NP, 0]	ϵ
r3	[5, VP, 1, NP, 0]	ϵ
r1	[4, S, 0]	ϵ
accept	[4, S,0]	ϵ

Initially State 0 is added to the empty stack. The noun phrase (NP) corresponding the word "John" is shifted onto the stack and the parser transits to State 1 by adding it to the stack. Next, the verb (V) corresponding to the word "sees" is shifted onto the stack and the parser transits to State 6 by adding it to the stack. Next, the determiner (Det) corresponding to the word "a" is shifted onto the stack and the parser transits to State 2 by adding it to the stack. Next, the noun (N) corresponding to the word "book" is shifted onto the stack and the parser transits to State 10 by adding it to the stack. Next, the noun and the determiner on top of the stack are reduced to a noun phrase using Rule 6 ($NP \rightarrow Det\ N$), which is added to the stack, and the parser transits to State 11 by adding it to the stack. Next, the noun phrase and the verb on top of the stack are reduced to a verb phrase (VP) using Rule 3 ($VP \rightarrow V\ NP$), which is added to the stack, and the parser transits to State 5 by adding it to the stack. Next, the verb phrase and the noun phrase on top of the stack are reduced to a sentence (S) using Rule 1 ($S \rightarrow NP\ VP$), which is added to the stack, and the parser transits to State 4 by adding it to the stack. Finally, the input string is accepted.

3.8.3 Compilation

Compiling LR parsing tables consists of constructing the internal states (i.e., sets of dotted items) and from these deriving the accept, shift, reduce and goto entries of the transition relation.

New states can be induced from previous ones; given a state `S1`, another state `S2` reachable from it by `goto(S1,Sym,S2)` (or `shift(S1,Sym,S2)` if `Sym` is a terminal symbol) can be constructed as follows:

1. Select all items in state `S1` where a particular symbol `Sym` follows immediately after the dot and move the dot to after this symbol. This yields the kernel items of state `S2`.

2. Construct the non-kernel closure by repeatedly adding a so-called non-kernel item (with the dot at the beginning of the RHS) for each grammar rule whose LHS matches a symbol following the dot of some item in `S2`.

Using this method, the set of all parsing states can be induced from an initial state whose single kernel item has the top symbol of the grammar preceded by the dot as its RHS. In figure 3.11 this is the item $S' \rightarrow \cdot S$.

The accept, shift and goto entries fall out automatically from this procedure. Any dotted item where the dot is at the end of the RHS gives rise to a reduction by the corresponding grammar rule. Thus it remains to determine the lookahead symbols of the reduce entries.

In *Simple LR (SLR)* the lookahead is any terminal symbol that can follow immediately after a symbol of the same type as the LHS of the rule. In *LookAhead LR (LALR)* it is any terminal symbol that can immediately follow the LHS given that it was constructed using this rule in this state. In general, LALR gives considerably fewer reduce entries than SLR, and thus results in faster parsing.

3.8.4 Relation to decision-tree indexing

A simplified version of LR parsing is decision-tree indexing. The procedure for indexing a grammar in a decision tree is the following: First one tree is constructed for each rule in the grammar. The root of each tree is the LHS symbol of the rule, and from this there is an arc labelled with the first RHS symbol of the rule to another node, and from this node in turn there is an arc labelled with the second RHS symbol of the rule, etc. Then trees whose roots are labelled with the same symbol are merged, allowing rules with the same LHS and the same initial RHS symbols to be processed together until they branch off.

Each node in the decision tree corresponds to some state(s) in the LR parsing tables.[4] The arcs correspond to the shift and goto entries where the end node of each arc corresponds to the new state. The arcs labelled with terminal symbols correspond to shift actions and the ones with nonterminal symbols correspond to goto entries. The leaves of the tree correspond to reductions.

This parsing scheme also requires a stack to keep track of where to transit once a rule has been applied and a phrase corresponding to the label of the root has been found. At the other end, if a nonterminal arc is encountered, the tree with a root with that label must be searched.

This does not give maximal prefix merging. Rules with different LHS are not processed together, nor are rule combinations yielding the same prefix. Also, some mechanism must be introduced for handling recursion to avoid the following scenario: If a tree with root label X has an arc labelled X, then the parser will return to the root node, find its way to the arc labelled X, return to the root node again and thus loop.

3.8.5 Extensions

Local-ambiguity packing and employing a graph-structured stack are two important techniques for extending the basic LR-parsing scheme which are due to Tomita. These are described in more detail in [Tomita 1986]. Employing a non-deterministic LR-parsing scheme with these two extensions is often referred to as "generalized LR parsing".

Local-ambiguity packing can be visualized by imagining that one performs all possible derivations in parallel, maintaining a set of stacks, while synchronizing the shift actions. If two stacks are identical (apart from the internal structure of the symbols), their origin will differ in the derivation of one (or more) nonterminal symbol. For subsequent parsing they are treated as a single stack and only when recovering the parse trees are they distinguished. Thus this parsing is done only once.

Using a graph-structured stack can then be viewed as merging the set of stacks in the parallel processing into a graph.

Realizing these efficiently in Prolog requires using a well-formed-substring table as described in Section 3.2.

[4] This correspondence is in general not one-to-one in either direction.

Chapter 4

The Core Language Engine

The name of this booklet contains the term "natural-language", which normally is used by computer scientists as a contrast to formal and programming languages. Thus a field within the computer science subarea of Artifical Intelligence has traditionally been called Natural-Language Processing — a field whose practitioners have come from an AI background and have realised that in order to make "truly intelligent" systems some kind of language understanding must be included. Even though AI now is a more or less dead area in that very few scientists like to be referred to as working within it, NLP is very much alive.

Quite a few of the researchers working with language-processing computers do, however, have a background in the humanities, being linguists which have realised that large-scale and applied linguistics can only be addressed if using computers. This subfield within linguistics is commonly referred to as Computational Linguistics, a term which sometimes is used in quite a wide sense, including all work in linguistics in which a computer is used for anywhich purpose.

The boundaries between Natural-Language Processing on one hand and Computational Linguistics on the other have become more and more blurred in that computer scientists have realised the need for linguistic knowledge within their systems and linguists are starting to realise that much of the programming techniques and theories of computer science are useful for linguistic purposes, as well. Of course this development is beneficiary for all parties and will certainly continue, making the distinction between the two areas obsolete, which it mostly already is.

Some aspects of the NLP–CL distinction still remain, however. The main difference nowadays is often on a empiricist versus theoretician basis, NLP being the area more concerned with building large-scale systems.

This brings us back to the title of the booklet. The reference to NLP is there since the authors think the only way to prove their theories (sic!) is by building large, commercially feasible systems. The system for Swedish which we use is based on one for English originally developed by a group at SRI International, Cambridge, England and is called the “Core Language Engine” (CLE).

The purpose of this chapter is to introduce that system and its grammar formalism. The description will by necessity be quite brief, to get a complete picture of the CLE system the reader is referred to the CLE book [Alshawi (ed.) 1992]. The Swedish version of the CLE used at SICS is described in [Gambäck & Rayner 1992].

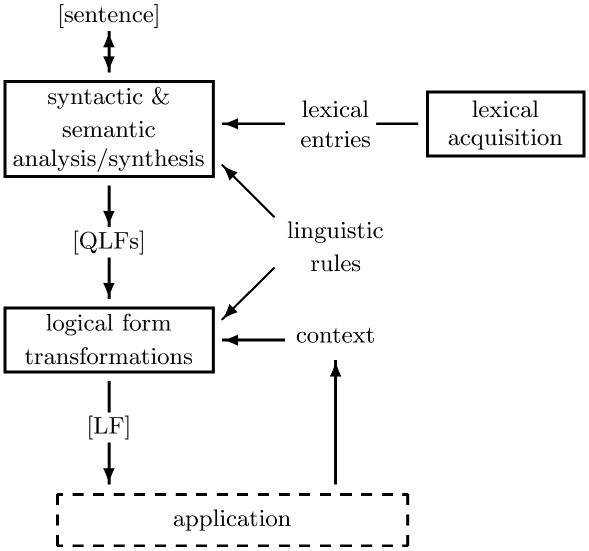

Figure 4.1: Broad overview of the CLE architecture

4.1 Overview of the CLE

The name Core Language Engine reflects the purpose of the system, it is intended to be the “core” (the heart) of an NLP system of any kind, for example a query-interface to a database, a machine translation system, or as the back-end of a speech understanding system. As an “engine” it should be a well-functioning work-horse.

The CLE is written purely in Prolog and based on unification as the main mechanism. The grammar formalism is a feature-category type with declarative bidirectional rules, which means that the same grammar can be used for both language analysis and generation.

Other hall-marks of the CLE are already familiar from the previous chapters: as much information as possible is put in the lexicon (rather than in the grammar rules) and the system employs a left-corner shift-reduce parser. A broad overview of the CLE architecture is shown in Figure 4.1.

Apart from using unification as the underlying mechanism in all parts of the system, the main design decision behind the CLE was to build it as a number of modules or processing steps, each with well-defined input from and output to the other modules. This has resulted in the design of a number of interface formalisms, one of which now can be taken as one of the main concepts of the CLE: the notion of "Quasi-Logical Form" (QLF) which is the formalism used for representing natural-language (compositional) semantics.

The QLF is a conservative representation of the meaning of an input sentence based on purely linguistic evidence. Semantics (mainly QLF-based) will be discussed later on, here we will only show the CLE's first processing steps which produce that form from an input NL-sentence (see Figure 4.2).

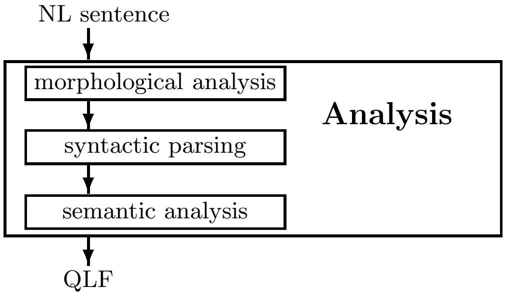

Figure 4.2: The analysis steps of the CLE

Other processing steps will then take application and situation dependent information into account when converting the QLF into a "pure" logical form, LF. The processing beyond QLF-level will, however, only be discussed in passing in this booklet. For a full description of this, the reader is again referred to [Alshawi (ed.) 1992].

The usefulness of any NLP system will depend largely on its ability to adjust (be customized) rapidly to new domains. This was reflected in the CLE design by equipping the system with a "core" grammar which should cover a large portion of English, but which should still be adjustable to a new register (sublanguage) without too large an effort.

On the lexicon side, ease of extendability is even more important. Thus the CLE has a special lexicon acquisition component, shown on the right-hand side of Figure 4.1. The idea being that a potential user with little or no linguistic knowledge should be able to expand the lexicon easily by herself.

4.2 The CLE grammar formalism

We now turn to the rule-formalisms used in the CLE. Actually all the formalisms used for rules of different kinds in the various models have the same general appearance. This section will discuss the syntactic grammar and lexicon rules in particular, but as we shall see in the sections following the morphology and semantic rules are more or less equivalent. A syntactic rule is schematically

$$\texttt{syn(}\langle\textit{syntax-rule-id}\rangle\texttt{, }\langle\textit{rule-group}\rangle\texttt{,}$$
$$\texttt{[}\langle\textit{mother-category}\rangle\texttt{,}$$
$$\langle\textit{daughter-category}_1\rangle, \ldots, \langle\textit{daughter-category}_n\rangle\texttt{]).}$$

The functor name `syn` indicates that this is a syntactic grammar rule, while its first argument is the actual name of the rule. This could be almost anything, but for ease of readability and debugging, the convention for rule names is that the first part lists the mother (i.e, RHS) and daughter (LHS) categories, then the portion (if any) after the first upper case letter provides a brief comment.

The second argument indicates the rule group to which the rule belongs. This is important for example when customizing the system to a new domain, but will not be discussed here.

The third argument is the really important part of the rule giving the categories and feature-value pairs of both the left-hand and right-hand side of the grammar rule as such. In the rules, the mother node and the daughter nodes form a Prolog-type list, so the → of the previous chapters has now been "replaced" by the first comma, while the commas following separate the daughters from each other.

Each constituent (both mother and daughters) in the schema above consists of a colon-separated pair `Category:FVList` where the `Category` of course is the category name, while `FVList` is a Prolog-type list of feature-value pairs given in the same fashion as introduced in Section 2.2.2, that is, each constituent is

$$\texttt{Category:[Feature}_1\texttt{=Value}_1\texttt{,...,Feature}_n\texttt{=Value}_n\texttt{]}$$

It should be obvious that having the LHS and RHS separated by a comma rather than an arrow and giving the rule name as an argument rather than putting it at the far right is just a notational variant. The formalism of the CLE and the formalism used for the example "toy" grammars earlier in the text are in fact completely equivalent.

4.2.1 A toy grammar for Swedish

A simple grammar for a fragment of Swedish in the CLE formalism is shown in Figure 4.3. Note that this grammar makes use of its features in two ways described in some detail in Section 2.4.2.

Firstly, the rule `vp_v_comp_Normal` leaves the verb's complements as a function of the value of the feature `subcat`, thus allowing the subcategorization scheme to be instantiated in the lexicon at run-time. This of course means that the parse-table compiler must be designed with care, but that is really of no concern of the grammar writer, who instead rejoices at the thought of being able to capture several grammar rules in only one rule schema.

The Swedish CLE grammar distinguishes between some 50 different types of verbs, all of which can be captured by a single rule looking like the one in the toy grammar. Having to write specific rules for each of these verb types would of course be both tedious and error-prone.

Apart from verb subcategorization, both adjectives and some nouns show a similar behaviour and can be treated by other rule schemas in a parallel fashion.

Another use of features described before is given by `vform`, which is a feature which ensures that the top-level (σ) rule for declarative sentences only is applicable if the (main) verb is finite. This feature is no longer boolean (as it was when called just `fin` in Section 2.4.2), but can rather take on a range of values corresponding to all the inflectional forms a Swedish verb can take.

The lexicon formalism

A corresponding lexicon is shown in Figure 4.4. The lexicon formalism mainly follows the one of the syntax rules, but with the functor name `lex` rather than `syn`. The lexicon rules parallels the syntax in containing a `Category:FVList` pairs list as described above. Each lexicon entry can actually contain several different categories, thus in effect defining several words, but the entries we will discuss here will only introduce one word at the time.

As shown in the lexicon, the feature `agr` carries gender agreement, which in Swedish manifests itself as either common or neuter gender (nowadays commonly referred to as "n-genus" and "t-genus", respectively, reflecting their definite-form suffixes).

Abstractly, an entry in the lexicon can be defined as

```
lex(⟨terminal-root⟩,
    [⟨terminal-category⟩:
     [⟨Feature_1 =Value_1,  ...,  Feature_n =Value_n⟩]
    ]).
```

where the only really new acquaintance is *terminal-root*, which is the base form of the terminal, that is the non-inflected form.

So far, the lexicon base form has been identical to the form in the input string. Considering the example from Icelandic given in Figure 2.10 this is obviously not an approach which recommends itself. In Section 4.3 below we will discuss how inflectional morphology helps to produce the lexicon base form of a word from all its possible surface forms.

First, however, we shall look at some examples of how a real-sized unification grammar actually may look in practise.

```
syn(sigma_decl, core,
 [sigma:[],
   s:[vform=fin]
 ]).

syn(s_np_vp_Normal, core,
 [s:[vform=Vform],
   np:[],
   vp:[vform=Vform]
 ]).

syn(vp_v_comp_Normal, core,
 [vp:[vform=Vform],
   v:[vform=Vform,
      subcat=Complements]
   | Complements
 ]).

syn(np_det_nbar, core,
 [np:[],
   det:[agr=Agr],
   nbar:[agr=Agr]
 ]).

syn(np_name, core,
 [np:[],
   name:[]
]).
```

Figure 4.3: A Swedish grammar in the CLE formalism

```
lex(kalle,[name:[]]).

lex(lena,[name:[]]).

lex(bil,
 [nbar:[agr=common]
 ]).

lex(garage,
 [nbar:[agr=neuter]
 ]).

lex(snarkar,
 [v:[vform=fin,
     subcat=[]]
 ]).

lex(gillar,
 [v:[vform=fin,
     subcat=[np:[]]]
 ]).

lex(ger,
 [v:[vform=fin,
     subcat=[np:[],
             np:[]]]
 ]).

lex(en,
 [det:[agr=common]
 ]).

lex(ett,
 [det:[agr=neuter]
 ]).
```

Figure 4.4: A lexicon in the CLE formalism

4.2.2 The "real" grammar

In the real Swedish grammar used in the CLE, the rules are of course quite a lot more complex, even though the follow the general scheme outlined above. The main cause of complexity is the number of features. In the toy grammar not even a hand-full of features were used; in the real grammar some 1500 different features are passed around.[1]

Feature defaults and value spaces

As shown by for example the feature agr above, the features need not be binary valued. Instead, we can specify the ranges ("syntactic feature value space") and default values of some of the features:

```
syn_feature_value_space(agr,
   [[plur,sing],
    [1,2,3],
    [common,neuter]]).

feature_default(nullmorphn,n).
```

A few features have specified "value spaces". The feature agr thus has a value space reflecting its function of carrying number, person, and gender agreement. The sublists in the definition are to be interpreted as Cartesian products of the elements, giving the feature $2 * 3 * 2 = 12$ possible values.

If no feature value space is defined, the feature is assumed to be binary valued, or rather to have the value space [y,n,_], i.e., "yes", "no" or "uninstantiated".

There is no requirement that a feature must have a declaration of any kind — just using it in the grammar with a specific category is enough; however, default declarations are used for some features, so e.g. nullmorphn is defined to be n per default. If no default value is present for a feature, it is assumed to be _, i.e., uninstantiated. Using default values for some features allows the grammar writer to leave them out from rules where their use simply follows from their default values, thus simplifying the grammar rules.

[1] The actual number of distinct features of a grammar does in some respect depend on how they are counted. Is for example the feature agr on an *NP* the same as the feature agr on a *VP*, or are they two different features which just happen to have the same name (and also oftentimes unify with each other)? The number 1500 given here comes from counting all features on different categories as distinct.

In passing, we should point out something which following the above example should be obvious, but actually is a common misconception even among skilled computational linguists: the number of features or the number of grammar rules are of no real importance when measuring the complexity or coverage of a grammar (a claim like "my grammar is better than yours, since it has more rules" or a question like "how many grammar rules do you have?" are nonsensical).

What really counts when "bolstering" your own grammar is proven coverage percentage figures on unseen parts of standardized corpora and the inherent properties of the formalisms used. For a suggested measurement method of the latter, see e.g. [Gambäck *et al* 1991].

Classification of sentence types

As a typical example of the usage of features in real unification grammars, we will now discuss how the different types of sentences can be separated from each other by an elaborate use of features.

The type of a sentence (WH-question, `q` or normal, `norm`) is defined by that of the subject, via passing of a feature `type`. The features `type`, `whmoved`, and `inv` (inverted) all default to `n` and classify sentences as shown in Figure 4.5.

Sentence type	`type`	`whmoved`	`inv`	Example
Declarative	`norm`	`n`	`n`	kalle lever
Yes-No question	`norm`	`n`	`y`	lever kalle
Non-subject WH-question	`q`	`y`	`y`	vad gör kalle
Subject WH-question	`q`	`n`	`n`	vem hyrde bilen

Figure 4.5: Sentence classification by feature values

All the sentence types in the table must contain a finite main verb, i.e., they must have `vform=fin`, so simple declarative sentences have a feature setting looking like

```
[type=norm, whmoved=n, inv=n, vform=fin]
```

That is, they must contain a finite verb, but may not contain any movement.

Yes-No questions look like declaratives, but are inverted (`inv=y`), i.e., the main verb comes *before* the subject.[2]

Wh-questions come in two types, depending on whether the WH-word is the subject of the sentence or not. Non-subject WH-questions are treated as could be expected, i.e., as having undergone two movements. One of the verb to obtain an inverted sentence and one WH-word movement. Subject WH-questions are however treated as having undergone no movement at all.

A grammar rule

As an (rather hairy) example of a rule from the real grammar consider the following, which is the non-simplified Swedish CLE grammar version of a rule which should by now almost be like an old friend, the “normal” main sentence rule $S \rightarrow NP\ VP$:

[2] As pointed out in Section 2.4.1, any verb (not just auxiliaries as in English) can be moved in Scandinavian languages like Swedish. Thus the Yes-No question structure, rather than the declarative, has sometimes been assumed to be the basic sentence structure for this language group. This assumption is the one underlying so called “Diedrichsen schemata”, which will not be used here, see however for example [Diderichsen 1966].

```
syn(s_np_vp_Normal, core,
    doc("[Jag flyger]", [d1,d3,d4,e1,e3,q2,q4],
        "Covers most types of finite
         and subjunctive, uninverted clause"),
[s:[hascomp=n, sententialsubj=SS, conjoined=n
    | Shared],
  np:[nform=Sfm, vform=(fin\/att), agr=Ag,
      reflexive=_, pron=_, sentential=SS, wh=_,
      whmoved=_, passive=P, temporal=_
      | NPShared],
  vp:[vform=(fin\/inf\/presp\/att), subjform=Sfm,
      agr=Ag, modifiable=_, headfinal=_, passive=P
      | VPShared]
])

:- NPShared = [relagr=RelAgr, type=Type, case=Case],
   VPShared = [gaps=Gaps, vform=VForm,
               vpellipsis=VE, svi=FrontedV],
   append(NPShared,VPShared,Shared).
```

Some parts of the rule are new, apart from the introduction of a number of new features which will not be discussed here. One new part is the third argument doc which is used for documentation. It is actually optional and was thus not mentioned above.

A more important aspect of the rule is the usage of lists for features which are simply shared between the constituents. The sentence shares some features with the noun-phrase and some with the verb-phrase. These features are represented by the two lists NPShared and VPShared and appended together to form the list Shared. This is completely equivalent to writing out all the features explicitly in the feature-value lists, but the usage of the shared lists reflects one of the overall endeavours of unification grammar theories: to keep the grammar rules proper as "clean" as possible.

Yet another new concept in the rule can be seen if the values of the feature vform are scrutinized. Here they suddenly contain the symbol \/ which is used as a type-writer variant of the normal logical disjunction (OR, $\cup$). The CLE formalism also allows features to be given values containing conjunctions (AND, $\cap$, written as /\) or negation ($\neg$, written \).

The feature vform is in the real system defined as

```
syn_feature_value_space(vform,
    [[fin, impera, perfp, presp, supine,
      inf, att, stem, supine_stem, n],
     [present,imperf,fut]]).
```

that is, as a Cartesian product of the second list (giving present, past, and future tense, respectively) and the first list giving a number of different verb-forms such as finite, imperative, past- and present participle, supine (a Swedish-specific verb-form), etc.

A grammar rule could thus contain the following feature-value setting for `vform`

```
vform=(inf\/(fin/\(present\/imperf)))
```

defining it to be either infinite *or* finite and either present or past tense (the latter part could thus in this case have been written as `(fin/\(\(fut)))`, i.e., finite and not future, which following the declaration of `vform` of course would be equivalent).

4.3 Morphology

The CLE contains a rather rudimentary treatment of Swedish morphology, mainly treating the most common types of pre- and suffixing. Given this, we will not go into much detail describing the CLE morphological processing; however, for sake of completeness of the system description, we will in the following give a short overview of it.

The strategy outlined in this section does work quite nicely for English, but it should be noted that a full-fledge treatment of morphology for a "real" language[3] most certainly would involve a methodology akin to that of Koskenniemi's "two-level morphology" [Koskenniemi 1983]. For suggestions on how to implement such a strategy in Prolog, see [Pulman 1991] or [Abramson 1992].

As implemented, the parts of the CLE which in Figure 4.2 were (following normal practise) lumped together as the morphology component really consists of several different modules, of which the most important are called

Segmentation: splits words into bits (lexemes / morphemes)

Morphology: assigns syntactic categories to words

Derivation: assigns semantic categories to words

Even though it forms one distinct module in the CLE, the field of morphology as such generally includes the others, but is also commonly divided into at least three subfields: inflectional and derivational morphology and the formation of compounds. We will only describe segmentation and inflectional morphology in this text, since the other parts would not add anything interesting to the discussion.

[3] As far as inflectional morphology is concerned, English can be viewed as more or less a toy example, thus (at least partially!) motivating the rather hash judgement of it not being a "real" language in this sense. Swedish morphology is actually quite a simple case, too, albeit it is substantially much more rich than the English one.

4.3.1 Segmentation

The purpose of the segmentation component is to locate all possible root-form and affix combinations in the input sentence. This is accomplished with segmentation rules, which define suffixes, prefixes, and infixes in terms of the surrounding letters.

Although the exact details of how this is done is not really that important, consider as an example the following segmentation rule for the suffix '-et' forming the definite form of second, fourth or fifth declension nouns (as in *bord* → *bordet*) and the perfect participle of fourth conjungation verbs (e.g., *riv* → *rivet*)

```
suffix_rule(['-et'],
  [pair(v1t,v1),             % leendet = leende + et
                             % hjärtat = hjärta + et
   pair(l0l2c5et,l0l2ec5)],  % offret = offer + et
                             % tecknet = tecken + et
   pair(et,[])).             % tåget = tåg + et
```

In the rule `v1` and `c5` are interpreted as variables over classes of letters, in this case all vowels resp. 'r', 'l', and 'n'. `l0` and `l2` match all Swedish letters and all letters but 'm' (since 'm' follows some rather special spelling rules).

The rule consists of three arguments, defining the letters of the suffix (`et`), non-default cases, and the default case, respectively. The third argument here simply says that the '-et' ending by default is obtaine adding the two letters to the word without deleting anything from it (e.g., *tåg* → *tåget*).

The second argument is the most interesting of this rule example. It defines non-default cases which may optionally be applied in addition to the default case. Each `pair` defines a range of letters to add and delete from the word when forming the '-et' ending form. The non-defaults of this rule give different syncopation[4] cases, so for example the first `pair` says that a word ending with a vowel could have the suffix 'e' removed (as in *hjärta* → *hjärtat*), while the second pair basically says that a word ending with 'er', 'el' or 'en' could have the root-form 'e' removed.

4.3.2 Inflectional morphology

The, for our purposes, most interesting part of the morphology component is the (syntactical) morphological inflection rules. These rules define how different affixes can be added to root-forms while taking into account restrictions introduced by the values of several features. The formalism used for the rules closely follows the one given before for the grammar, now with `morph` as the functor name.

Abstractly, a morphology rule is:

[4] *syncope* is the loss of one or more sounds or letters in the interior of a word.

```
morph(⟨morph-rule-id⟩,
      [⟨mother-category⟩,
       ⟨daughter-category1⟩,...,⟨daughter-categoryn⟩]).
```

Here, the daughters normally consist of a root-form followed by affixes; all the parts of the rule are given as categories with feature-value pairs, as usual. The root-form can be either a lexicon form or the output of another morphology rule. The affixes are treated as normal lexicon items and defined by rules like the following, which gives '-ar' as the plural ending for second declension nouns:

```
lex('-ar',['PLURAL':[synmorphn=2,lexform='-ar']]).
```

The category of the affix is here simply given as PLURAL, while the feature synmorphn holds the "syntactical morphological category of a noun", i.e., the noun declension.

An example of a morphology rule is the noun plural one, nbar_nbar_plural which in essence says that adding the right ending to the singular stem gives the plural, changing only the value of the agreement feature agr. All the other features are the same on the mother and the daughter noun and passed as a list of shared (unified) features in the same fashion as in the grammar:

```
morph(nbar_nbar_plural,
[nbar:[agr=plur | Shared],
 nbar:[agr=sing | Shared],
 'PLURAL':[synmorphn=Morph]
])

   :- Shared=[def=n, mass=n, measure=M,
              nn_infix=Inf, synmorphn=Morph,
              temporal=T, subcat=S,
              lexform=L, paradigm=Par, simple=y].
```

What the "right ending" for a specific noun is depends on its declension, i.e., on the value of synmorphn, which as we can see must unify between all the components of the rule. Its value on the root-form (daughter) nbar is defined in the noun's lexicon entry.[5]

4.4 Compositional semantics

Language theory is commonly divided into syntax, semantics and pragmatics. The distinctions between these fields are not at all clear-cut, and several different definitions of them have been given over time. Here we will adopt the following division:

[5] As the morphology rules really can be viewed upon as being part of the grammar, the general philosophy of keeping the grammar clean from information which can be lexicalized applies to them as well.

Syntax: defines how signs are related to each other.

Semantics: defines how signs are related to things.

Pragmatics: defines how signs are related to people.

Semantics in turn can be divided into *compositional semantics*, which defines the (in some sense) abstract "meaning" of a sentence from the meaning of the parts, and *situational semantics* which borders pragmatics in adding context-dependent information to the interpretation.

As indicated already at the beginning of Chapter 1 the main purpose of natural-language systems is to translate an input utterance from natural language to some type of internal representation, i.e., to *interpret* the utterance. To accomplish this, the semantic processing of the CLE really consists of several steps,

Semantics: assigns semantic categories to phrases.

Reference resolution: gets references for pronouns, etc.

Scoping: determines the scope of quantifiers and such.

the first of which, i.e., the part which really is the compositional semantics, is the subject of this section. The others are context-dependent and will be briefly discussed later on (in Section 4.5).

However, we will start out by describing what we are aiming for, that is, the internal representation, the logical forms. The discussion of logical forms for the CLE will by necessity be an abbreviated version of the one given in [Alshawi & van Eijck 1989] and [Alshawi (ed.) 1992]. The CLE formalism and its treatment of semantics in general in turn builds on the work originally done by Richard Montague described in [Thomason 1974]. For an introduction to Montague semantics see [Dowty *et al* 1981] or [Gamut 1991].

4.4.1 The logical formalism

The logical formalism of the CLE is called "Quasi-Logical Form" (QLF), indicating that it is not a "pure" logical form (LF). In particular, transforming the QLF expressions formed by the semantic processing into such "pure" (true) LF expressions requires:

1. Fixing the scopes of quantifiers and operators.

2. Resolving pronouns, definite descriptions, ellipsis, underspecified relations and vague quantifiers.

3. Extracting the truth conditional information.

After these steps, partially described in Section 4.5, we would get what Hiyan Alshawi likes to refer to as "fully instantiated QLF" [Alshawi & Crouch 1992]. In the following, however, we will somewhat sloppily call this form just LF (logical form) and discuss it first before moving on to QLF.

A particular (uninstantiated) QLF expression may correspond to several, possibly infinitely many, LF expressions, i.e., instantiations. However, the LF language as it will be defined here is in fact just a sublanguage of the QLF language; there are additional "quasi logical" constructs for unscoped quantifiers, unscoped descriptions, unresolved references and unresolved relations, to be discussed shortly.

Logical form requirements

When using logical forms as the internal representation of natural-language utterances, we must make sure that they satisfy the following requirements:

- LFs should be expressions in a disambiguated language, i.e., alternative readings of natural language expressions should give rise to different logical forms.

- LFs should be suitable for representing the literal "meanings" of natural-language expressions, i.e., they should specify the truth conditions of (appropriate readings of) the original natural-language expressions.

- LFs should provide a suitable medium for the representation of knowledge as expressed in natural language, and they should be a suitable vehicle for reasoning.

The predicate logic part

The formalism used in the CLE is a higher order logic, in which extensions to first order logic (FOL) have been motivated by trying to satisfy the above requirements with respect to the range of natural-language expressions covered. For now we will restrict ourselves to the predicate logic part (in BNF-like rules in Figure 4.6) and introduce the higher order extensions later on: In this notation, the logical form:

```
[or,
   [anka1,kalle1],
   [struts1,kalle1]
]
```

expresses the proposition that Kalle is a duck (*anka*) or an ostrich (*struts*), with `anka1` and `struts1` being one place predicates, `kalle1` a constant, and `or` the usual disjunction operator.

$$
\begin{array}{rcl}
\langle \mathit{formula} \rangle & \texttt{::=} & \texttt{[}\langle \mathit{predicate} \rangle\texttt{,}\langle \mathit{argument}_1 \rangle\texttt{,} \ldots \texttt{,}\langle \mathit{argument}_n \rangle\texttt{]} \\
 & | & \texttt{[not,}\langle \mathit{formula} \rangle\texttt{]} \\
 & | & \texttt{[and,}\langle \mathit{formula} \rangle\texttt{,}\langle \mathit{formula} \rangle\texttt{]} \\
 & | & \texttt{[or,}\langle \mathit{formula} \rangle\texttt{,}\langle \mathit{formula} \rangle\texttt{]} \\
 & | & \texttt{[impl,}\langle \mathit{formula} \rangle\texttt{,}\langle \mathit{formula} \rangle\texttt{]} \\
 & | & \texttt{quant(}\ \langle \mathit{quantifier} \rangle\texttt{,}\langle \mathit{variable} \rangle\texttt{,} \\
 & & \qquad \langle \mathit{formula} \rangle\texttt{,}\langle \mathit{formula} \rangle\texttt{)} \\
\langle \mathit{predicate} \rangle & \texttt{::=} & \texttt{snarka1} \mid \texttt{anka1} \mid \texttt{struts1} \mid \texttt{geq} \ldots \\
\langle \mathit{argument} \rangle & \texttt{::=} & \langle \mathit{term} \rangle \\
\langle \mathit{term} \rangle & \texttt{::=} & \langle \mathit{variable} \rangle \\
 & | & \langle \mathit{constant} \rangle \\
\langle \mathit{variable} \rangle & \texttt{::=} & \texttt{X} \mid \texttt{Y} \ldots \\
\langle \mathit{constant} \rangle & \texttt{::=} & \texttt{kalle1} \mid \texttt{lena1} \ldots \\
\langle \mathit{quantifier} \rangle & \texttt{::=} & \texttt{forall} \mid \texttt{exists}
\end{array}
$$

Figure 4.6: BNF definition of the predicate logic part

The notation allows restricted first order quantifiers. For a simple Swedish sentence like *Alla pojkar ser Lena*, a logical form translation in the notation above contains quantified variables:

```
quant(forall,P,[pojke1,P],
      quant(exists,H,[event,H],
            [se1,H,P,lena1]))
```

Here `P` and `H` are variables bound by the familiar first order logic quantifiers (i.e., ∀`P` and ∃`H`). This logical form can be paraphrased as *För varje pojke* `P`*, existerar det en händelse,* `H`*, sådan att* `P` *ser Lena.*

4.4.2 The semantic rule formalism

The semantic rules indicate how the meaning of a complex expression is composed of the meanings of its constituents. Every syntactic rule of the CLE has one or more corresponding semantic rule. Each semantic rule thus has a semantic rule identifier distinguishing it from other cases for the semantics of the same syntax rule:

$$
\begin{array}{l}
\texttt{sem(}\langle \textit{syntax-rule-id} \rangle\texttt{, } \langle \textit{semantic-rule-id} \rangle\texttt{,} \\
\quad \texttt{[(}\langle \textit{logical-form} \rangle\texttt{, } \langle \textit{mother-category} \rangle\texttt{),} \\
\qquad \langle \textit{daughter-pair}_1 \rangle, \ldots, \langle \textit{daughter-pair}_n \rangle\texttt{]).}
\end{array}
$$

To distinguish the semantics from the syntax, the semantic cases will in the following be referred to as ⟨*syntax-rule-id*⟩`*`⟨*semantic-rule-id*⟩, thus including the semantic rule identifier in the rule name.

The logical form paired with the mother category forms a QLF expression, normally not fully instantiated, corresponding to the semantic analysis of the constituent analysed by the syntax rule whose identifier is ⟨*syntax-rule-id*⟩. The format of the semantic rule mother category follows that of the syntax rule, i.e., it is `Category:FVList`.

The mother logical form typically contains variables that are unified with the semantic analyses of the daughter constituents. Such variables appear as the left-hand elements of the daughter pairs:

`(`⟨*daughter-qlf-variable*⟩`,`⟨*daughter-category*⟩`)`

In the example semantic rule below, the variable `Nbar` stand for the semantic analysis of the daughter, and by unification its value appear in the logical form template associated with the mother:

```
sem(np_nbar_Def, mass,
[( term(_,ref(def,bare,mass,l(Ain)),V,Nbar),
   np:[handle=v(V), anaIn=l(Ain)
      | Shared]),
  (Nbar,
   nbar:[agr=sing, mass=y, anaIn=l([V-sing|Ain])
         | Shared])
])

   :- Shared=[anaOut=Aout, semGaps=Gaps].
```

Categories appearing in semantic rules may include specifications for the values of syntactic features (i.e., features that have appeared in some syntax rule, for example `mass` in the rule above). This is commonly used when distinguishing different semantic cases of the same syntactic rule.

The semantic rule above is actually a version of a rule

$$NP \rightarrow \overline{N}[+def, +mass, +sing]$$

for forming complete noun-phrases from simple definite form nouns. This semantic case of the rule can be used for singular agreement mass nouns, as indicated by the semantic-rule identifier `mass` and shown by the values of the features `agr` and `mass`. The setting of the feature `def` is, however, not shown in the semantic rule, since it is the same for *all* the cases of the syntactic rule (it does show up in the syntactic rule, of course).

Forming noun-phrases from simple nouns is quite common and there are other semantic cases of the corresponding syntax rule treating singular non-mass nouns and plural nouns. There are also specific NP-formation rules for indefinite plural and mass nouns which also can form NPs on their own.

The syntactic features in the rule are complemented with specific semantic features. Semantic features are often used to hold logical form fragments passed between mother and daughter constituents. So for example `handle` in the rule above holds a logical-form variable `V` which appears both in the QLF for the mother NP and in the feature `anaIn` on the daughter $\overline{N}$.

In a parallel fashion to the corresponding syntactic rules, there are cases where the set of daughter items in a semantic rule is empty, or where a daughter position may be unified with part of the feature structure of another daughter. Empty constituents are treated by gap-threading (as in the syntax) and passed as the value of the feature `SemGaps` (which holds a difference list just like its corresponding syntactic feature `gaps`). The anaphora features `anaIn` and `anaOut` actually together form another difference list, giving the possible intra-sentential referents — the rule above adds a new one.

Semantic features will be exemplified and more carefully described in the rest of this section, even though they of course are not substantially different from the syntactic ones in any sense. Quite importantly, for example, just like the syntactic features, the semantic features can have default declarations associated with them. These defaults have the general form

`semantic_feature_default(`⟨*feature*⟩, ⟨*value*⟩`).`

4.4.3 Semantic analysis

The syntactic parsing phase generates *all* possible syntax trees of a given input sentence. These trees are fed to the semantic analysis phase, which turns the trees into appropriate quasi logical forms.

If, for example, the sentence

Flyget avgår (*The flight leaves*)

was analysed, the CLE syntactic parser would return the only parse:

```
[sigma_decl-1,
 [[s_np_vp_Normal-2,
   [[np_nbar_Def-3,
      [[lex-4,[flyg,-et]]]],
    [vp_v_comp_Normal-5,
      [[lex-6,[avgå,-r]]]]
   ]]]]
```

The CLE parser output is just another form of the implicit parse tree described in Section 2.4, thus listing the syntactic rules which have been applied in the parse. The numbers in the tree simply identify rule-applications, while the splitting of the terminals show the lexemes and suffixes used by the morphology rules.

The semantic analysis would then apply semantic rules with the same names as the syntactic ones in a top-down fashion, giving the QLF:

```
[dcl,
 form(l([flyget,avgår]),verb(pres,no,no,no,y),A,
      B^
      [B,
       [avgå_2p,A,
        term(l([flyget]),
             ref(def,bare,sing,l([])),_,
             C^[flyg1,C],_,_)]],
      _)]
```

The details of this `form` will be explained in due time; for now, we only note that `avgå_2p` and `flyg1` are semantic sense names corresponding to the words in the input string. The `2p` and `1` extensions simply distinguish the sense names from the lexemes and could have been chosen rather arbitrarily; however, e.g., the `2p` actually indicates that the verb *avgå* is a two-place predicate, i.e., an intransitive verb (the first place of the predicate is for the event itself, as described in detail in Section 4.4.13 below).

4.4.4 Semantic rules

The analysis in the previous section would have been obtained by the application of a number of semantic rules, all of which will be described in this section. Thus we will at the same time see a substantial portion of the most important rules of the real CLE grammar; however, to improve readability (and hopefully also intelligibility), the rules will in the following be stripped of most of their features. We will discuss the rules in the same order as in the implicit parse tree above, starting with the top-node:

```
sem(sigma_decl, only,
 [([dcl,S],sigma:[]),
   (S,s:[type=norm, whmoved=n, inv=n,
         vform=fin, semGaps=([],[])])
 ]).
```

Many syntactic features show up again in the semantics. As discussed above, the only new feature here is `semGaps` — the semantic counter-part of the syntactic gap-list feature `gaps`. As in the syntax, the two arguments form a difference list specifying the `gapsIn` and `gapsOut` values, respectively.

The variable `S` simply passes the logical-form value of the `s`-node up to the `sigma`-node. The extra operator `dcl` added to the QLF in the σ simply states that this is a declarative sentence (operators like this one are really higher-order extensions and will thus be further discussed in Section 4.4.9).

The semantic rule-identifier `only` indicates that this is the only semantic rule that can match the syntactic rule `sigma_decl`, or in other words, that there is a one-to-one correspondence between syntax and semantics in this case.

```
sem(s_np_vp_Normal, only,
[(Vp,s:[anaIn=Ain, anaOut=Aout | Shared]),
  (Np,np:[semGaps=([],[]),
          anaIn=Ain, anaOut=Anext]),
  (Vp,vp:[anaIn=Anext, anaOut=Aout | Shared])
])

 :- Shared=[subjval=Np, movedv=FrontedV,
           eventvar=E, semGaps=Gaps].
```

This rule states that the S-meaning is the VP-meaning with the meaning of the subject (`subjval`) plugged in. `subjval` is thus a feature carrying a QLF-fragment. This is also true for `eventvar`, which carries a logical form variable matching the event (verb) itself.

The values of the features `anaIn` and `anaOut` (which together form a difference list) shows that the possible anaphoric referents are "threaded" through the rule in the same fashion as gaps normally are. The semantic gaps, however, are not treaded here, but rather only unified between the VP and the S. No empty constituents are allowed in the noun-phrase.

The feature `movedv` performs the same function as a corresponding syntactic feature `svi` (subject-verb inversion) and is used for the left-movement of verbs. Both `movedv` and `svi` keep the information necessary for getting the correct complements to the moved verb, and subject-verb agreement.

The value of `movedv` is a pair of the same format as the semantic rule-constituents:

`pair(`⟨*qlf-variable*⟩`,` ⟨*category*⟩`)`

The *qlf-variable* carries the meaning of the V down.[6] A rule for empty verb-constituents, `v_gap*only` will plug it in where it was "moved" from:

```
sem(v_gap, only,
[(V,v:[svi=v:SynShared, movedv=pair(V,v:SemShared)
     | Shared])])

 :- SemShared=[subjval=A, eventvar=E,
              arglist=Comp, semGaps=SemGaps,
              anaIn=Ain, anaOut=Aout,
              @shared_tense_aspect(TenseAspect)],
    SynShared=[agr(Agr), gaps=Gaps, subcat=S],
    append(SynShared,SemShared,Shared).
```

[6] Yes, "down". Since we are talking proper unification, the direction is hardly that important, but remember anyhow that semantic analysis in the CLE is a top-down process!

By using shared features for both syntax and semantics in the rule, we clearly see that the verb-gap has the same feature values as the moved verb. These values are held by `svi` and `movedv`, respectively.

Of the other new features shown in the rule, `@shared_tense_aspect` passes any tense, aspect and mood information needed, while `arglist` performs the same function as the familiar syntactic feature `subcat`, i.e., holds the verbs subcategorization scheme (`Comp`) as instantiated in the lexicon.

```
sem(np_nbar_Def, sing,
[( term(_,ref(def,bare,sing,l(Ain)),V,Nbar),
   np:[anaIn=l(Ain) | Shared]),
  (Nbar,
    nbar:[agr=sing, mass=n,
          anaIn=l([V-sing|Ain]) | Shared])
])

 :- Shared=[anaOut=Aout, semGaps=Gaps].
```

This rule is just another case of the `np_nbar_Def*mass` rule which has already been discussed (in Section 4.4.2). The case above is used for forming an `np` from a singular definite `nbar`. The main reason for having three cases (the third one is for plurals) of this rule shows up in the new referent added to the anaphora list. This rule imposes the restriction on it that it must be singular. The logical form of the NP formed by a `term` (again, a higher-order expression which is discussed in more detail later) also indicates this: `ref(def,bare,sing,...)` shows that this is a singular definite form referential expression without a determiner (i.e., "bare").

```
sem(vp_v_comp_Normal, mainv,
[(V,vp:Shared),
  (V,v:[semanticAux=n, arglist=Comp | Shared])
  | Comp
])

 :- Shared=[eventvar=E, subjval=A, movedv=FrontedV,
            @shared_tense_aspect(TenseAspect),
            semGaps=Gaps, anaIn=Ain, anaOut=Aout].
```

The final rule used in the analysis is the one for forming a verb-phrase from a verb and its complements. As can be seen, most information is unified between the `v` and the `vp` (e.g., the logical form passed in the variable `V`). The rule seen above is thus quite simple, but it is actually only the main verb case of the corresponding syntactic rule. The syntax rule treats almost any kind of verb and obligatory verb modification; the semantic rules, however, distinguish between whether the verb semantically behaves like an auxiliary or not.

Main verbs (and non-finite[7] modals) are `semanticAux=n`, while tense auxiliaries and finite modals are `semanticAux=y`, meaning that they will not fully influence the semantics of the verb-phrase. For those verbs, the semantics of the verb-phrase will also need to take into account any information from the modified non-finite verb.

4.4.5 Sense entries

The semantic rules of the previous section have to be complemented with a semantic lexicon in order to obtain the QLF shown on Page 85. The semantic lexicon entries are commonly referred to as "sense entries", since they give the semantic senses of the syntactic lexicon items.

A sense entry in the CLE looks as the following one for the count noun `flyg`, which as we can see has empty gap- and anaphora-lists. The logical form fragment of the entry contains the sense name (`flyg1`).

```
sense(flyg,
      (A^[flyg1,A],
       nbar:[semGaps=(G,G), anaIn=Al, anaOut=Al])).
```

For an intransitive verb like `avgår`, the sense entry is a bit more complicated:

```
sense(avgår,
      (form(_,verb(T,Pf,Pg,M,A),Event,
            P^[P,[avgå_2p,Event,Subject]],_),
       v:[vform=impera, tense=no, modal=imp,
          perf=Pf, prog=Pg, active=A,
          eventvar=v(Event), subjval=Subject,
          semGaps=(G,G), anaIn=Al, anaOut=Al,
          arglist=[],
          subcat=[]])).
```

Firstly, `arglist` and its syntactic counter-part `syncat` show the subcategorization to be empty. Secondly, a number of features are added to cope with tense and aspect; of these note only that the lexicon verb-form in the Swedish CLE is the imperative, since most other verb-forms can be formed easily from it. The imperative is special in more or less standing on the side of the tense-system, thus it has `tense=no`.

Finally, the logical-form fragment of the verb contains a list with the sense name (`avgå_2p`) as well as variables for the event itself and for the subject. For non-intransitive verbs, this list would also contain the QLFs of the arguments. Note that when forming a verb-phrase (with the rule `vp_v_comp_Normal*mainv`) we do not know what the subject of the verb is — but we do know that it is going to be unified with the value of the feature `subjval` (which it is for example in `s_np_vp_Normal*only`, see above)!

[7] The non-finite verb-forms in Swedish are the infinites and the supine.

4.4.6 Higher order extensions

After having acquinted ourselves with the first order logic part of the QLF language and how it is used, we now turn our attention to its higher order logic extensions which can be specified with the following additional BNF-like rules:

$$
\begin{array}{rcl}
\langle \mathit{formula} \rangle & \texttt{::=} & \texttt{[}\langle \mathit{mood\_op} \rangle\texttt{,}\langle \mathit{formula} \rangle\texttt{]} \\
\langle \mathit{mood\_op} \rangle & \texttt{::=} & \texttt{dcl} \mid \texttt{ynq} \mid \texttt{whq} \mid \texttt{imp} \\
\langle \mathit{argument} \rangle & \texttt{::=} & \langle \mathit{formula} \rangle \\
\langle \mathit{argument} \rangle & \texttt{::=} & \langle \mathit{abstract} \rangle \\
\langle \mathit{quantifier} \rangle & \texttt{::=} & \texttt{en} \mid \texttt{wh} \ldots \\
\langle \mathit{quantifier} \rangle & \texttt{::=} & \langle \mathit{variable} \rangle\texttt{\^{}}\langle \mathit{variable} \rangle\texttt{\^{}}\langle \mathit{formula} \rangle \\
\langle \mathit{abstract} \rangle & \texttt{::=} & \langle \mathit{variable} \rangle\texttt{\^{}}\langle \mathit{lambda\_body} \rangle \\
\langle \mathit{lambda\_body} \rangle & \texttt{::=} & \langle \mathit{formula} \rangle \\
\langle \mathit{lambda\_body} \rangle & \texttt{::=} & \langle \mathit{abstract} \rangle
\end{array}
$$

The kinds of $\langle \mathit{formula} \rangle$ that are distinguished at top level are declaratives, yes/no questions, wh-questions, and imperatives, as marked by the mood operators `dcl`, `ynq`, `whq`, and `imp` respectively. However, the `dcl` operator is sometimes omitted.

4.4.7 Abstraction and application

Lambda abstraction is used to construct functions of arbitrary complexity (properties of objects, relations between objects, and so on).

In the LF notation `X^[ful,X]` corresponds to the more usual notation for lambda-abstraction, $\lambda x.ful(x)$.[8]

The BNF rules show that terms, formulae and abstracts can all act as arguments to a functor-expression. For every functor expression the types of the arguments that it takes are fixed and the only way of forming higher order expressions is by means of the abstraction functor "`^`".

The logical counterpart of the abstraction functor "`^`" is the functor `apply` for lambda-application. This functor expresses the result of applying an abstract to an appropriate argument.

Thus

```
[apply,X^[kvinna1,X],lena1]
```

reduces to

```
[kvinna1,lena1]
```

[8] The CLE variant is actually a type-writer version of the lambda-operator notation $\hat{x}$ used by Montague.

In the LF representations produced by the CLE, the only uses of `apply` reduce properties to formulae by applying them to terms. This special case can be expressed as follows in a specific BNF rule:

$$\langle argument\rangle \quad \texttt{::=} \quad \texttt{[apply,}\langle variable\rangle\texttt{\^{}}\langle formula\rangle\texttt{,}\langle argument\rangle\texttt{]}$$

4.4.8 Generalized quantifiers

Logical form quantifiers are not restricted to existentials and universals; these are simply special cases of generalized quantifiers. A generalized quantifier is a relation Q between two sets A and B (where A is called the *restriction set* and B the *body set*) that satisfies some specific requirements, as shown in Figure 4.7.

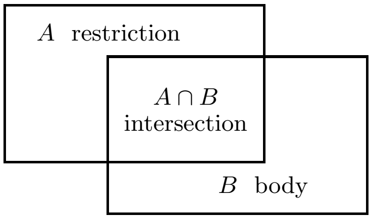

Figure 4.7: Generalized quantifiers

For present purposes it is enough to note that the requirements can be summarized as the condition that Q be insensitive to anything but the cardinalities of the sets A (the *restriction set*) and the set $A \cap B$ (the *intersection set*). Thus a generalized quantifier with restriction set A and body set B is fully characterized by a predicate $\lambda n \lambda m.\mathbf{Q}(n, m)$ on n and m, where $n = |A|$ and $m = |A \cap B|$, as examplified by:

- *Alla pojkar snarkar* is true if and only if the restriction set (the set of boys) equals the intersection set (the intersection of the set of boys and the set of people who snore).
- *Minst tre pojkar snarkar* is true if and only if the intersection set contains at least three individuals.
- *Inte alla pojkar snarkar* is true if and only if the the restriction set does not equal the intersection set.
- *De flesta pojkar snarkar* is true (in neutral contexts) if and only if the size of the intersection set is greater than half the size of the restriction set.
- *Minst fyra, men som mest tio pojkar snarkar* is true if and only if the intersection set contains at least four and at most ten individuals.

4.4.9 Statements and questions

The logical forms that the CLE assigns to questions are similar to those for declarative statements.

Logical forms for yes/no questions are distinguished from those for declarative statements by the top level (mood) operator `ynq`.

For example, the logical form for the sentence 4.2 is the same as that for 4.1 except that the operator `ynq` has replaced the operator `dcl`.

Flyget avgår. (4.1)

```
[dcl,
 form(l([flyget,avgår]),verb(pres,no,no,no,y),A,
      B^
      [B,
       [avgå_2p,A,
        term(l([flyget]),ref(def,bare,sing,l([])),_,
             C^[flyg1,C],_,_)]],
      _)]
```

Avgår flyget? (4.2)

```
[ynq,
 form(l([avgår,flyget]),verb(pres,no,no,no,y),A,
      B^
      [B,
       [avgå_2p,A,
        term(l([flyget]),ref(def,bare,sing,l([])),_,
             C^[flyg1,C],_,_)]],
      _)]
```

4.4.10 “Quasi-logical” constructs

The basic constructs by which the QLF language extends the LF language are the following:

1. Terms (`term`) for unscoped quantified expressions (*ingen man*, *varje man*), definite descriptions (*mannen*, *de där männen*), and unresolved references, such as pronouns, reflexives, etc. (*han*, *det*, *sig*).

2. Formulae for implicit relations (`form`). These are used for among other things genitives (*Kalles bok*) and unresolved temporal and aspectual information (*idag*).

3. "island" constructions (`island`) to block the raising of quantifiers outside "island" constituents, i.e., constituents which are isolated from the rest of the tree (see also Section 2.3.2).

The BNF rules for the additional QLF term and formula constructs are given in Figure 4.8.

```
     ⟨formula⟩  ::=  form( ⟨string⟩,⟨category⟩,⟨index⟩,
                           ⟨restriction⟩,⟨form_ref⟩)
                 |   [island,⟨formula⟩]
        ⟨term⟩  ::=  term( ⟨string⟩,⟨category⟩,⟨index⟩,
                           ⟨restriction⟩,⟨quantifier⟩,⟨term_ref⟩)
                 |   ⟨index⟩
      ⟨string⟩  ::=  l(⟨words⟩)
       ⟨index⟩  ::=  ⟨variable⟩
                 |   ⟨constant⟩
 ⟨restriction⟩  ::=  ⟨variable⟩^⟨lambda_body⟩
    ⟨form_ref⟩  ::=  ⟨variable⟩
                 |   ⟨formula⟩
    ⟨term_ref⟩  ::=  ⟨variable⟩
    ⟨category⟩  ::=  q(⟨index⟩)
                 |   ref(def,⟨definite⟩,⟨number⟩,⟨antecedents⟩)
                 |   ref(pro,⟨pronoun⟩,⟨number⟩,⟨antecedents⟩)
      ⟨number⟩  ::=  plur | sing | mass
    ⟨definite⟩  ::=  den | bare ...
     ⟨pronoun⟩  ::=  han | hon ...
 ⟨antecedents⟩  ::=  l(⟨variables⟩)
```

Figure 4.8: BNF definition of the "quasi-logical" part

The ⟨*category*⟩ arguments are categories in the sense of collections of linguistic attributes. They are used to pass linguistic information, including syntactic information, to the scoping and reference resolution phases. This can include information on number, reflexivity, the surface form of quantifiers, and so on.

There are several types of categories identified by constants; the ones shown in the definition above are, for example, anaphoric expressions (e.g., `ref` for noun phrase reference) and phrase types (e.g., `pro` for pronoun, `def` for definite description).

The quantifier `term` notation above (together with the ⟨*quantifier*⟩ definition on Page 89) thus replaces the simpler first-order quantification treatment (`quant`) given in Section 4.4.1.

4.4.11 Quantified terms and descriptions

In quantified `term`s the category gives the lexical form of the determiner, and marks the singular/plural distinction. The QLF analysis of *Några flyg avgår* is:

```
[dcl,
 form(l([några,flyg,avgår]),verb(pres,no,no,no,y),A,
      B^
     [B,
       [avgå_2p,A,
        term(l([några,flyg]),q(tpc,någon,plur),_,
             C^[flyg1,C],_,_)]],
      _)]
```

Information in `term` categories is used by scoping and reference resolution to decide the scope of a determiner, the quantifier it corresponds to, and whether a collective interpretation is possible.

The `island` operator shown in the BNF rules serves to prevent unscoped quantifiers in its range from having wider scope than the operator. In particular, it is used to block the raising of `term`s out of relative clauses during the scoping procedure (Section 4.5.2).

Definite descriptions are also represented as quantified terms in QLF. For example the compund noun phrase in *Dallas-flyget avgår* is translated as the `term`:

```
term(l([dallas-flyget]),ref(def,bare,sing,l([])),_,
     C^
     form(l([dallas-flyget]),nn,_,
          D^
          [and,[flyg1,C],
           [D,C,
            term(_,proper_name(_),_,
                 E^[name_of,E,Dallas_Stad],_,_
                )]],
          _),
     _,_)
```

The reference resolution phase (Section 4.5.1) determines whether to replace a definite description with a referent (giving a referential reading) or whether to convert it into a quantification (giving an attributive reading).

4.4.12 Anaphoric terms

Pronouns are represented in QLF as `term`s in which the restriction places constraints on a variable corresponding to the referent, and the category contains linguistic information which guides the search for possible referents.

For example, in the QLF for *Han avgår*, the `term` for *han* is:

```
term(l([han]),ref(pro,han,sing,l([])),_,
     C^[sex_GenderOf,Male,C],_,_)
```

while the representation of *honom* in *Kalle ser honom* is:

```
term(l([honom]),ref(pro,han,sing,l([C-sing])),C,
     E^[sex_GenderOf,Male,E],_,_)]],
```

where `C` is the variable bound to *Kalle*.

The value of ⟨*antecedents*⟩ in the `term` category is a list of possible antecedents within the same sentence. It contains indices to the translations of noun phrases that precede or dominate the given pronoun in the sentence, allowing, for example, reference to bound variables.

As can be seen in the example above, the antecedent list can impose restrictions on the bindings of the variables; here, the variable `C` is restricted to bind to a singular referent.

4.4.13 Event variables

Verbs are not treated simply as relations between a subject and a number of VP complements. The event being described is introduced as an additional argument.

The presence of an event variable allows optional verb phrase modifiers to be treated as predications on events, in first order fashion, which in turn permits a uniform interpretation to be given to prepositional phrases, whether they modify verb phrases or nouns.

Take as an example the sentence

Avboka biljetten i Boston. (4.3)

which has two readings, depending on the attachment of the prepostional phrase *i Boston*. On one reading it expresses a property of the ticket and on the other a property of the event of cancelling the ticket.

Several examples of this notorious *PP-attachment problem* has been seen before in the text and this is certainly no coincidence: this is one of the most common types of ambiguity found in natural languages, but one which is quite neatly represented in the logical form notation used in the CLE.

The reading of in which *i Boston* is taken to modify the verb phrase gives rise to the first logical form following, while the interpretation in which the prepositional phrase takes part in the noun phrase modification gives the second QLF:

```
[imp,
 form(l([avboka,biljetten,i,Boston]),
      verb(no,_,_,imp,_),A,
      B^[B,
      form(l([i,Boston]),prep(i),_,
           C^[C,v(A),term(l([Boston]),
                     proper_name(_),_,
                     D^[name_of,D,Boston_Stad],
                     _,_)],
           _),
      [avboka_Något,A,
       term(_,ref(pro,imp_subj,_,l([])),
            E,F^[personal,F],_,_),
       term(l([biljetten]),ref(def,bare,sing,l([E-_])),
            _,G^[biljett1,G],_,_)]],
   _)]
```

```
[imp,
 form(l([avboka,biljetten,i,Boston]),
      verb(no,_,_,imp,_),A,
      B^[B,
      [avboka_Något,A,
       term(_,ref(pro,imp_subj,_,l([])),
            C,D^[personal,D],_,_),
       term(l([biljetten,i,Boston]),
            ref(def,bare,sing,l([C-_])),
            E,F^[and,[biljett1,F],
            form(l([i,Boston]),
                 prep(i),_,
                 G^[G,E,term(l([Boston]),
                             proper_name(_),_,
                             H^[name_of,H,Boston_Stad],
                             _,_)],
                 _)],
            _,_)]],
      _)]
```

4.5 Later-stage processing

The final steps of the CLE processing can be thought of as forming the situational semantic part in the sense that they include information which is situation dependent. These steps are the ones shown in Figure 4.9.

This section will very briefly discuss and exemplify the first two steps, reference resolution and scoping; plausibility checking simply involves checking the

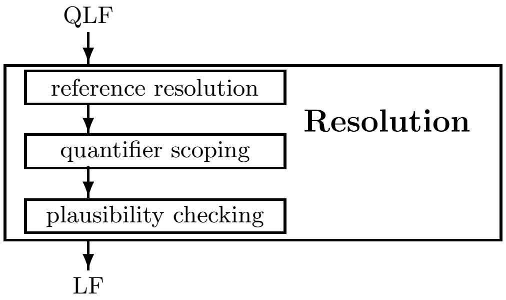

Figure 4.9: The resolution steps of the CLE

relevance of the logical form produced in the given context.

4.5.1 Reference resolution

The main purpose of the reference resolution component is to determine the referents to underspecified "referring expressions" (`terms` and `forms`) (names, pronouns, etc.). We have seen some (partial) examples of this already; here, we will not go into any detail on how this is done, but simply give a full example.

If the English CLE would be fed with the sentence

Who works on CLAM-BAKE?

the semantic processing would produce the following unresolved QLF

```
[whq,
 form(l([who,works,on,CLAM-BAKE]),
      verb(pres,no,no,no,y),
      A,
      B^
      [B,
       [work_on_BeEmployedOn,
        A,
        term(l([who]),
             q(tpc,wh,C),D,E^[personal,E],F,G),
        term(l([CLAM-BAKE]),
             proper_name(H),I,
             J^[name_of,J,CLAM-BAKE],K,L)]],
      M)]
```

Reference resolution would process the QLF above and replace the name "CLAM-BAKE" with its corresponding referent. Supposing the name in a given situation actually referred to a project, the unscoped, resolved logical form would be:

```
[whq,
 [work_on_BeEmployedOn,
  term(A,exists,B,C^[event,C],D,E),
  term(F,q(tpc,wh,G),H,I^[personal,I],J,K),
  SF(c(x^[project_Activity,x],CLAM-BAKE))]]
```

4.5.2 Scoping

The scoping component follows an algorithm which very simply can be said to take the following steps:

1. rewrite a formula containing a quantifier `term` into one in which the `term` has been given a scope as a quantifier
2. do this for all the quantifier `term`s in an unscoped logical form in all possible ways
3. apply *local* constraints to block certain alternatives (or to suggest a preference ranking among alternatives)

Again, we only give a full example of the functionality of the component. Continuing the example from the previous section, the scoped QLF produced would be

```
[whq,
 quant(wh,
       A,
       [personal,A],
       quant(exists,
             B,
             [event,B],
             [work_on_BeEmployedOn,
              B,
              A,
              SF(c(x^[project_Activity,x],
                   CLAM-BAKE))])))]
```

Where we can note that the quantifiers `wh` and `exists` have been determined to have scope over the entire sentence and over the verb phrase (event), respectively.

Chapter 5

The Basics of Information Retrieval

Sorting documents so that they can be found easily is difficult, especially if more than one reader is expected to be able to use the document collection.

5.1 Manual methods

The traditional way of organizing documents and books are sorting them physically in shelves after categories that have been predetermined. This generally works well, but finding the right balance between category generality and category specificity is difficult; the library client has to learn the categorization scheme; quite often it is difficult to determine what category a document belongs to; and quite often a document may rightly belong to several categories.

Some of these drawbacks can be remedied by installing an index to the document collection. Documents can be indexed in several ways and can be reached any of several routes. *Indexing*, i.e. establishing correspondences between a set, possibly large and typically finite, of index terms or search terms and individual documents or sections thereof. Indexing is both difficult and often quite dull; it poses great demands on consistency from indexing session to indexing session and between different indexers. This is the sort of job which always seems to be a prime candidate for automatization.

5.2 Automatization

5.2.1 Words as indicators of document topic

The basic assumption of automatic indexing mechanisms is that the presence or absence of a word - or more generally, a term, which can be any word or combination of words - in a document is indicative of topic. The basic scenario of *information retrieval* is then finding the correct combination of query words combined together in a logical structure using Boolean operators such as AND OR or NOT, and then examining the set of documents that fit the query. The documents in the set match the query, the documents outside do not. If a document matches a query it is presented, if it does not, it will not be. This set algebraic approach works reasonably well for document bases where the documents are short and concise: a list of literature abstracts or document titles, for instance. For full texts it is too imprecise: full documents contain too many spurious terms for Boolean matching to be practicable.

Boolean systems are still in use, especially for trained documentalists; for untrained users, the Boolean approach has been largely abandoned in full-text retrieval systems - although the name *retrieval* has been retained for an activity which only in its extreme cases resembles retrieval - in favor of a more *probabilistic* approach, which ranks the retrieved documents by likelihood of providing relevant data for the resolution of the query. The basic improvement is the *weighting* of terms by assumed importance for a document.

5.2.2 Word frequencies - tf

The first steps to find index terms automatically is to build a list of words in a text, and sort them in order of frequency of occurrence. The more frequent terms are considered more valuable in proportion to their observed frequencies. This suggestion is first made by Hans Peter Luhn (1957, 1959), and the measure is commonly called *term frequency* or, imaginatively, *tf* for short. For this text, for instance, the list will be as shown in table 1.

```
92 the
72 of
64 and
62 to
60 a
55 in
51 is
30 for
26 terms
25 documents
24 be
23 that
23 as
22 words
22 term
21 text
20 information
19 document
```

```
17 this
17 retrieval
16 are
   ...
```

Table 1. Frequency table of words in this text.

An obvious improvement in the enterprise to build a semantic representation from a list such as the one in Table 1 is to filter out certain words that seem to have little to do with topic. A list of such words, most often grammatical form words and other closed class words, is commonly called a *stop list.* Another is to note - as Luhn does in his 1959 paper - that the most frequent words seldom are significant for this sort of enterprise, and that thus it might be possible to filter them out automatically.

```
|
| \
|  \
|   \
|     \
|       \        ___--___
|         \ ___--        --___
|          /\                 \
|         /  \                 \
|        /    \                 \
|       /      \_                \
|      /         \__              \
|     /             \___           \
|    /                  \____       \
|   /                        \____   \
|  /                              \___\__
| /                                    \ \_______
|/                                      \
+-------------------------------------------------
```

Figure 1. Significance vs frequency. From Luhn (1959).

```
the
a
and
that
one
it
two
may
could
```

```
such
next
just
half
both
of
to
in
for
...
```

Table 2. Stoplist.

26	terms
25	documents
22	words
22	term
21	text
20	information
19	document
17	retrieval
14	idf
14	frequency
11	technical
10	word
10	indexing
10	collection
9	table
8	single
8	query
	...

Table 3. Frequency table of words in this text, filtered with stoplist.

5.2.3 Normalizing inflected forms

As can be seen in tables 1 and 3 the words ”document” and ”documents” both show up in the beginning of the list. The words ”indexed” and ”indexing” do not, and probably should - they show up further down in the list. Word form analysis, or *morphological analysis* would conflate these forms, and raise their combined weight.

It is unfortunate for the generality of the results in the field that the research and business language of the world currently is English, with an exceedingly spare morphology. Most information retrieval systems today make use of simple "stemming" - i.e. stripping suffixes from word forms without further analysis. This is sufficient, but of debatable utility, for English, but not for most other languages of the world. In comparison, language with richer morphologies such as Finnish or French show much better gains from morphological analysis (Koskenniemi, 19xx; Jacquemin and Tzoukermann, 1997).

5.2.4 Uncommonly frequent words - idf

If we try to determine what terms in a document are significant for representing its content, we find that terms that are common in a document, but also common in all other documents are less useful than others. The question is how *specific* a term is to a document.

Collection frequency, *inverse document frequency* or *idf* is a measure of term specificity first proposed by Karen Sparck-Jones (1972). This is defined as a function of *N/di*, where *N* is the total number of documents in the collection and *di* is the number of documents where term *i* occurs - the *document frequency.* This measure gives high value to terms which occur in only a few documents. Used alone, it gives about as useful results as term frequency used alone - *idf* is vectored towards high precision while *tf* gives better recall or indexing exhaustivity.

A modified *idf* measure - the *weighted idf* or *widf* is suggested by Tokunaga and Iwayama (1994). Their measure is weighted for term frequency in the documents which it occurs in: the *widf* is calculated as a function of *dfi* rather than *di*, where *dfi* is the frequencies of term *i* in the respective documents. Their experiments seem to indicate an improvement in performance.

5.2.5 Methodological problems with idf

The problem with *idf* as a measure is that it is unclear what universe the document frequency should be calculated over. The measures all depend on an *N*, a total number of documents, and establishing what general usage of a term is may be difficult, if not impossible. In some cases a collection is so well defined that a collection internal *idf* is quite adequate; in others, where potential readers may not be aware of the collection setup or if the collection is very heterogenous it may not. In table 2 you will find *idf* scores for words in this document; the scores are calculated with respect to the top twenty documents retrieved by Altavista for the query "information, retrieval, algorithm".

```
0.019 the
0.020 and
0.020 retrieval
```

```
0.020 to
0.021 information
0.021 for
0.022 is
0.022 of
0.023 a
0.023 this
0.023 with
0.024 in
0.025 as
0.026 from
0.026 or
0.026 not
0.027 search
0.029 be
0.030 use
0.031 but
0.031 language
0.032 example
0.033 form
0.033 on
0.033 text
0.033 web
0.034 documents
0.034 first
0.034 queries
0.034 query
0.034 words
      ...
1 behavior
1 behaviour
      ...
1 karlgren
      ...
1 mathematical
1 mathematically
1 mathematics
      ...
1 meteorological
      ...
1 microwave
      ...
1 miscellaneous
      ...
1 morphology
```

```
      ...
1 nationwide
      ...
1 navigation
      ...
1 pulmonary
      ...
1 radio
      ...
1 redundancy
      ...
```

Table x. Inverted document frequencies for terms in this collection.

5.2.6 Combining tf and idf

There are various ways of combining term frequencies and inverse document freqeuencies, and from empirical studies (e.g. Salton and Yang, 1973), we find that the optimal combination may vary from collection to collection. Generally, *tf* is multiplied by *idf* to obtain a combined term weight. Alternatives would be for instance to entirely discard terms with too low *idf* - which seems to be slightly better for high precision searches.

5.2.7 Document length effects

As the term weight is defined in the tf component of the combined formula, it is heavily influenced by document length. A long document about a topic is likely to have more hits than a short one will for a relevant term; this may not improve its likelihood of being relevant.

Most algorithms in use introduce document length as a normalization factor of some sort. In table y the formulae for OKAPI and the basic formulation of the SMART *cosine formula* are given. (Robertson and Sparck Jones, 1996; Singhal et al, 1995). "k" in the OKAPI formula in Table y is a constant to be set after experimentation with the particular collection of texts. If it is set to zero, the OKAPI formula reduces to idf.

```
OKAPI:

tf(w,d) * idf(w) * (k + 1) / [k * dl(d)/dlbar + tf(w,d)]

Cosine:

tf(d,w)/sqrt(sum(j)(tf(d,j)^2)
```

Table y. Document length normalization in SMART and OKAPI

5.3 Words as indicators of information need

Given that we have a document representation in the form of a list of terms with attached weights - often called a *term vector* - however these weights are computed, and a similar representation of the query, the question is how to match the two term vectors to find documents that fit the query.

One approach is to use the conventional scalar product of the two vectors, and simply add the pairwise products of each the weight of each term under consideration.

Documents and queries are very different. Luhn's original model was for the searcher to compose an essay of approximately the same form as the sought for documents, but in practice queries are of different lenght and different type than the target documents. The way the respective term vectors are to be treated can be expected to be very different. Empirical tests on small collections of texts of different types (Salton and Buckley, 1988), conclude i.a. that length normalization for queries does not affect the result; using an *idf* factor improves the result; when the document lengths vary - as they usually do - length normalization is useful. However, lately it has seemed that if the length varies over a great range, the document length normalization should be damped somewhat (Singhal et al, 1996).

5.4 Query expansion

As has been established both by informal observation and several formal experiments, queries to information retrieval systems tend to be very short; the majority being of three words or less (Rose and Stevens, 1996; Rose and Cutting, forthcoming). This gives very little purchase to most linguistically oriented methods, and one would wish to find methods which would encourage searchers to produce longer queries (Karlgren and Franzen, 1997).

5.4.1 Using retrieved documents

One method to get more textual material for fleshing out a short query is to submit the query, use the first few retrieved documents as a renewed query, and hope that the first few documents indeed are relevant.

5.4.2 Relevance Feedback

Alternatively, a retrieval system can present the retrieved set, and have users note which documents seem useful at a glance. These relevant documents are then used as a query. Analogously, non-relevant documents can be discarded in the first iteration, and the terms in them weighted down in subsequent iterations. This technique - *relevance feedback* - can be extended by clustering the retrieved documents in similarity sets, if the system has an efficient clustering algorithm.

Cutting et al have implemented the *Scatter/Gather* which does this. Users can select not only single documents for relevance feedback, but entire clusters of documents, represented by terms common to the entire cluster.

5.4.3 Extracting terms from retrieved documents

Automatic query expansion (e.g. Robertson; Strzalkowski et al, 1997).

Using relevance feedback (e.g. Robertson).

5.5 Beyond single words

Counting solitary words is fine, but the idea that lone words by themselves carry the topic of the text is one of the more obvious over-simplifications in the model so far. Indexing texts on ice cream on "ice" and "cream" is intuitively less useful than looking at the combination "ice cream", just to take an obvious example. However, in experiments designed to test the usefulness of multi-word terms, any addition past single word indexing is cumbersome and expensive in memory and processing, while adding comparatively little to performance. On balance, the methods are useful, but single word indexing is and will remain the most efficient method to index texts.

The usefulness of searching a document base for multi-word terms rather than limiting the search to single terms is under debate. "It may be sensible, for some files, to index explicitly on complex or compound terms ... In general these elaborations are tricky to manage and are not recommended for beginners." (Robertson and Sparck Jones, 1996). In any case, the discriminatory power of single word terms is much stronger than that of any other information source (Strzalkowski et al, 1997) and given the relatively low level of granularity of the text description simple words are quite likely to be sufficient.

5.5.1 Collocations and Multi-word technical terms

One way of expanding the search to words beyond single terms is simply tabulating words that occur adjacently in the text - *n-grams.* For instance, Magnus Merkel has implemented a tool for retrieving recurrent word sequences in text (1994).

¡!– FRASSE table for this text. –¿

Using more theoretical finesse, other types of arbitrary and recurrent combinations in the text - *collocations* - can be recognized and tabulated as well. Frank Smadja has implemented a set of tools (1992) for retrieving collocations of various types using both statistical and lexical information; he identifies three major types of collocations: predicative relations such as hold between verbs and their objects in recurrent constructions, set noun phrases, and phrasal templates, where only a certain slot varies from instance to instance.

To extract collocations of the second type, Justeson and Katz (1995) have added lexical knowledge simple statistics, and use it to extract *technical terms*. Technical terms are a specific category of words which behave almost like names. They cannot easily be modified - their elements cannot be scrambled or replaced by more or less synonymous components, and they usually cannot be referred to with pronouns. Thus, the technical terms tend to stay invariant throughout a text, and between texts.

Justeson's and Katz' appealingly simple algorithm to spot multi-word technical terms tabulates all multi-word sequences with a noun head from a text, and retains those that appear more than once. This method gives a surprisingly characteristic picture of a text topic, given that the text is of a technical or at least non-fiction nature. Their major point is well worth noting: the fact that a complex noun phrase is used more than once identically is evidence enough for its special quality as a technical term. It is repetition, not frequency, that is notable for longer terms.

Strzalkowski has experimented using head modifier structures from fully parsed texts to extract index terms (1994): this normalizes phrases such as "information retrieval" and "retrieval of information" to the same index representation.

5.5.2 Concept Spotting

Names of people, organizations, places, and other entities can be spotted automatically with some degree of success (Strzalkowski and Wang, 1996).

5.5.3 Information Extraction

5.5.4 Text is not just a bag of words

All indexing approaches mentioned so far assume that words appear in a text somewhat randomly, in a Poisson-style distribution. This is naturally a gross simplification: words appear in a text not following a memory-less distribution but in a pattern governed by the textual topic progression (Karlgren, 1976).

This can conceivably be made use of in indexing: if text segments more likely to be topically pertinent are chosen and terms within them weighted up as compared to terms from other sections this weighting would reflect the topical make-up of the text better than a non-progressional model. Techniques potentially useful for this include summarization, text segmentation (e.g. Hearst) and clause weighting approaches, for instance the foreground/background experiments performed by Karlgren.

5.5.5 Text is more than topic

Besides the topical analysis touched on in the preceding sections, it is worth noting that textual data are so much more than topic and information: texts belong to one or several genres, are of varying quality, are written for various purposes, and adhere or transcend stylistic conventions.

5.5.6 Merging several information streams

As has been indicated above, the optimal way of putting index streams to use may vary from stream to stream. Justeson and Katz find that repetition, not frequency is what makes a multi-word technical term an interesting index term; in experiments on single words mostly frequencies tend to be more efficient as weightings.

In experiments in recent TREC evaluations our group has used a multi-stream approach, where every stream is indexed and searched separately; the resulting searches are merged to present a unified result (Strzalkowski et al, 1996).

A numerical optimization for weighting several knowledge sources has been done by Bartell et al. (19xx)

5.6 Evaluating information retrieval

Information retrieval has a unusually well defined set of evaluation tools.

5.6.1 How exhaustive is the search? - Recall

If one has a good picture of how many relevant documents a document base contains for some query, one can calculate the proportion of the total set of relevant documents found and retrieved by an algorithm. This proportion is called *recall*.

5.6.2 How much garbage? - Precision

The proportion of relevant documents in a retrieved set is called *precision* and is a complement measure to recall.

5.6.3 Combining precision and recall

Trivially, if an algorithm always retrieves all documents in a document base, it has one hundred per cent recall. However, it presumably has low precision. Typically recall and precision are plotted against each other; in the TREC evaluations, e.g., a "11-point" average measure is used, with precision measures at every 10 percent of recall, and the average figure is used as a total measure (e.g. Harman, 1996).

5.6.4 What is wrong with the evaluation measures?

- We do not have a good picture of how many documents are relevant in a document base.
- Indeed, often we cannot determine what the 'entire document base' is.
- A query is well defined experimentally, but what its counterpart in real life is less well defined. Users often cannot pose their information needs in succinct search terms.
- The retrieved set is not delimited. A list of several thousand documents is presumably not very useful to a human user; should set an arbitrary cutoff point at some figure?
- Averaging recall-precision trade offs in e.g. 11-point averages mask algorithm differences: some algorithms may do very well in high-precision searches and less well in high recall cases; some may do well in cases where there are very few documents to be found, others do better when the document base is saturated with material for the topic at hand.
- Most importantly: what is "relevant"?

5.7 References

Nicholas J. Belkin. 1994. “Design Principles for Electronic Texytual Resources: Investigating Users and Uses of Scholarly Information”, *Studies in the memory of Donald Walker*, Kluwer.

Benedict du Boulay, Tim O’Shea, and John Monk. 1981. “The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices”, *International Journal of Man-Machine Studies*, 14:237-249.

Naoufel Ben Cheikh and Magnus Zackrisson. 1994. “Genrekategorisering av text för filtrering av elektroniska meddelanden” *Stockholm University Bachelor’s thesis in Computer and Systems Sciences* Stockholm University.

Jussi Karlgren. 1990. “An Algebra for Recommendations”, *Syslab Working Paper* 179, Department of Computer and System Sciences, Stockholm University, Stockholm.

Robert Kass. 1991. “Building a User Model Implicitly from a Cooperative Advisory Dialog”, UMUAI 1:3, pp.203-258

Ann Lantz. 1993. “How do experienced users of Usenet News select their information?”. *IntFilter Working Paper No. 3*, Department of Computer and Systems Sciences, University of Stockholm.

Sadaako Miyamoto. 1989. *Fuzzy Sets in Information Retrieval and Cluster Analysis.* Dordrecht: Kluwer.

Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergström, John Riedl. 1994. “GroupLens: An Open Architechture for Collaborative Filtering of Netnews”, *Procs. CSCW 94*, Chapel Hill.

Elaine Rich. 1979. "User Modeling via Stereotypes" *Cognitive Science*, vol 3:pp 329-354.

Gerald Salton and Michael McGill. (1983). *Introduction to Modern Information Retrieval* New York: McGraw-Hill.

Gerard Salton and James Allan. 1994. "Automatic Text Decomposition and Structuring", *Procs. 4th RIAO – Intelligent Multimedia Information Retrieval Systems and Management*, New York.

Donald E. Walker. 1981. "The Organization and Use of Information: Contributions of Information Science, Computational Linguistics, and Artificial Intelligence", *Journal of the American Society for Information Science* **32**, (5), pp. 347-363.

Hans Iwan Bratt, Hans Karlgren, Ulf Keijer, Tomas Ohlin, Gunnar Rodin. (1983). "En liberal datapolitik", *Tempus*, 16-19/12, Stockholm.

Sadaako Miyamoto. (1989). *Fuzzy Sets in Information Retrieval and Cluster Analysis.* Dordrecht: Kluwer.

Erik Andersson. 1975. "Style, optional rules and contextual conditioning. In *Style and Text - Studies presented to Nils Erik Enkvist.* Håkan Ringbom. (ed.) Stockholm: Skriptor and Turku: Åbo Akademi.

Douglas Biber. 1988. *Variation across speech and writing.* Cambridge University Press.

Douglas Biber. 1989. "A typology of English texts", *Linguistics, 27*:3-43.

Chris Buckley, Amit Singhal, Mandar Mitra, Gerard Salton. 1996. "New Retrieval Approches Using SMART: TREC 4". *Proceedings of TREC-4.*

John Dawkins. 1975. *Syntax and Readability.* Newark, Delaware: International Reading Association.

Nils Erik Enkvist. 1973. *Linguistic Stylistics.* The Hague: Mouton.

Donna Harman (ed.). 1995. *The Third Text REtrieval Conference (TREC-3).* National Institute of Standards Special Publication. Washington.

Donna Harman (ed.). 1996. *The Fourth Text REtrieval Conference (TREC-4).* National Institute of Standards Special Publication 500-236. Washington.

Donna Harman (ed.). forthcoming. *The Fifth Text REtrieval Conference (TREC-5).* National Institute of Standards Special Publication. Washington.

Fahima Polly Hussain and Ioannis Tzikas. 1995. "Ordstatistisk kategorisering av text för filtrering av elektroniska meddelanden" (Genre Classification of Texts by Word Occurrence Statistics for Filtering of Electronic Messages) *Stockholm University Bachelor's thesis in Computer and Systems Sciences*, Stockholm University.

George R. Klare 1963. *The Measurement of Readability.* Iowa Univ press.

Irving Lorge. 1959. *The Lorge Formula for Estimating Difficulty of Reading Materials.* New York: Teachers College Press, Columbia University.

Robert M. Losee. forthcoming. "Text Windows and Phrases Differing by Discipline, Location in Document, and Syntactic Structure". *Information Processing and Management.* (In the Computation and Language E-Print Archive: cmp-lg/9602003).

Tomek Strzalkowski. 1994. "Robust Text Processing in Automated Information Retrieval". *Proceedings of the Fourth Conference on Applied Natural Language Processing in Stuttgart.* ACL.

Ellen Voorhees, Narendra K. Gupta, Ben Johnson-Laird. 1994. "The Collection Fusion Problem". *Proceedings of TREC-3.*

Brian T. Bartell, Garrison W. Cottrell, and Richard K. Belew. 19xx. Automatic Combination of Multiple Ranked Retrieval Systems. xxx...xxx.

Donna K. Harman (ed.) 1993. The First Text Retrieval Conference (TREC-1), NIST SP 500-207, Gaithersburg, MD: National Institute of Standards and Technology.

Donna K. Harman (ed.) 1994. The Second Text Retrieval Conference (TREC-2), NIST SP 500-215, Gaithersburg, MD: National Institute of Standards and Technology.

Donna K. Harman (ed.) 1995. Overview of The Third Text Retrieval Conference (TREC-3), NIST SP 500-225, Gaithersburg, MD: National Institute of Standards and Technology. [http://www-nlpir.nist.gov/TREC/]

Donna K. Harman (ed.) 1996. The Fourth Text Retrieval Conference (TREC-4), NIST SP 500-236, Gaithersburg, MD: National Institute of Standards and Technology. [http://www-nlpir.nist.gov/TREC/]

Donna K. Harman (ed.) 1997. The Fifth Text Retrieval Conference (TREC-5), NIST SP 500-xxx, Gaithersburg, MD: National Institute of Standards and Technology. [http://www-nlpir.nist.gov/TREC/]

John S. Justeson and Slava M. Katz. 1995. Technical Terminology: some linguistic properties and an algorithm for identification in text. Natural Language Engineering, 1, 1, 9-27.

Jussi Karlgren and Kristofer Franzń. 1997. Verbosity and Interface Design. Reptile working papers No. 2. [http://www.sics.se/ jussi/irinterface.html].

Hans Peter Luhn. 1957. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. IBM Journal of Research and Development 1 (4) 309-317. (Reprinted in H.P.Luhn: Pioneer of Information Science, selected works. Claire K. Schultz (ed.) 1968. New York:Sparta.

Hans Peter Luhn. 1959. Auto-Encoding of Documents for Information Retrieval Systems. In Modern Trends in Documentation, M. Boaz (ed) London: Pergamon Press. (Reprinted in H.P.Luhn: Pioneer of Information Science, selected works. Claire K. Schultz (ed.) 1968. New York:Sparta.

Magnus Merkel, Bernt Nilsson and Lars Ahrenberg. 1994. A Phrase-Retrieval System Based on Recurrence. In Proceedings from the Second Annual Workshop on Very Large Corpora. Kyoto.

S. E. Robertson and Karen Sparck Jones. 1996. Simple, proven approaches to text-retrieval. Technical report 356, Computer Laboratory, University of Cambridge. [http://www.cl.cam.ac.uk/ftp/papers/reports/TR356-ksj-approaches-to-text-retrieval.ps.gz]

Daniel E. Rose and Curt Stevens. 1996. V-Twin: A Lightweight Engine for Interactive Use. Proceedings of the fifth Text Retrieval Conference, TREC-5. Donna Harman (ed), NIST Special Publication, Gaithersburg: NIST.

Gerard Salton and Christopher Buckley. 1988. Term-Weighting Approaches in Automatic Text Retrieval. Information Processing and Management. 24 (5) 513-523.

Gerard Salton and C. S. Yang. 1973. On the Specification of Term Values in Automatic Indexing. The Journal of Documentation. 29 (4) 351 - 372.

Amit Singhal, Gerard Salton, Mandar Mitra, Chris Buckley. 19xx. Document Length Normalization. Cornell CS TR000.

Frank Smadja. 1992. Retrieving Collocations from Text: XTRACT. Journal of Computational Linguistics. Special issue on corpus based techniques.

Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation. December 1972. 28:1:11-20.

Tomek Strzalkowski. 1994. Building a Lexical Domain Map from Text Corpora. In Papers presented to the Fifteenth International Conference On Computational Linguistics (COLING-94), Kyoto.

Tomek Strzalkowski, Louise Guthrie, Jussi Karlgren, Jim Leistensnider, Fang Lin, Jose Perez-Carballo, Troy Straszheim, Jin Wang, Jon Wilding. 1997. Natural Language Information Retrieval: TREC-5 Report Proceedings of the fifth Text Retrieval Conference, TREC-5. Donna Harman (ed.), NIST Special Publication, Gaithersburg: NIST.

Tomek Strzalkowski and Jin Wang. 1996. A Self-Learning Universal Concept Spotter. In Papers presented to the Sixteenth International Conference On Computational Linguistics (COLING-96), Copenhagen.

Takenobu Tokunaga and Makoto Iwayama. 1994. Text categorization based on weighted inverse document frequency. Technical Report 94 TR0001. Department of Computer Science. Tokyo Institute of Technology.

Bibliography

[Abramson 1992]
Harvey Abramson. 1992. “A Logic Programming View of Relational Morphology”. In *Proceedings of the 14th International Conference on Computational Linguistics*, volume 3, pp. 850–859, Nantes, France, July. ACL.

[Alshawi & Crouch 1992]
Hiyan Alshawi and Richard Crouch. 1992. “Monotonic Semantic Interpretation”. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pp. 32–39, Newark, Delaware, June. ACL. Also available as SRI International Technical Report CRC-022, Cambridge, England.

[Alshawi (ed.) 1992]
Hiyan Alshawi, editor, David Carter, Jan van Eijck, Björn Gambäck, Robert C. Moore, Douglas B. Moran, Fernando C. N. Pereira, Stephen G. Pulman, Manny Rayner, and Arnold G. Smith. 1992. *The Core Language Engine.* The MIT Press, Cambridge, Massachusetts, March.

[Aho 1968]
Alfred V. Aho. 1968. “Indexed grammars — an extension to context-free grammars”. *Journal of the ACM*, 4:647–671.

[Aho *et al* 1986]
Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts.

[Alshawi & van Eijck 1989]
Hiyan Alshawi and Jan van Eijck. 1989. “Logical Forms in the Core Language Engine”. In *Proceedings of the 27th Annual Meeting of the Association for Computational Linguistics*, pp. 25–32, Vancouver, British Columbia, June. ACL.

[Chomsky 1957]
Noam Chomsky. 1957. *Syntactic Structures.* Mouton, Haag, Holland.

[Chomsky 1981]
Noam Chomsky. 1981. *Lectures on Government and Binding.* Foris, Dordrecht, Holland.

[Chomsky 1986]
Noam Chomsky. 1986. *Barriers.* The MIT Press, Cambridge, Massachusetts.

[Chytil & Karlgren 1988]
Michail B. Chytil and Hans Karlgren. 1988. "Categorial grammars and list automata for strata of non-CF-languages". In W. Busszkowski, W. Marciszewski, and J. van Benthem, editors, *Categorial Grammar.* John Benjamins, Amsterdam, Holland.

[Colmerauer 1978]
A. Colmerauer. 1978. "Metamorphosis Grammars". In L. Bolc, editor, *Natural Language Communication with Computers.* Springer-Verlag, Berlin, Germany.

[Diderichsen 1966]
Paul Diderichsen. 1966. *Helhed og Struktur — udvalgte sprogvidenskabelige afhandlinger.* København. (in Danish).

[Dowty *et al* 1981]
David R. Dowty, Robert E. Wall, and Stanley Peters. 1981. *Introduction to Montague Semantics.* D. Reidel, Dordrecht, Holland.

[Earley 1969]
Jay Earley. 1969. "An Efficient Context-Free Parsing Algorithm". In *Readings in Natural Language Processing.* Morgan Kaufmann, San Mateo, California. Reprint.

[Gambäck *et al* 1991]
Björn Gambäck, Hiyan Alshawi, David M. Carter, and Manny Rayner. 1991. "Measuring Compositionality in Transfer-Based Machine Translation Systems". In J. G. Neal and S. M. Walter, editors, *Natural Language Processing Systems Evaluation Workshop*, pp. 141–145, University of California, Berkeley, California, June. ACL.

[Gamut 1991]
L. T. F. Gamut. 1991. *Logic, Language, and Meaning*, volume 2. The University of Chicago Press, Chicago, Illinois. (Gamut is a pseudonym for Johan van Benthem, Jeroen Groenendijk, Dick de Jongh, Martin Stokhof, and Henk Verkuyl).

[Gazdar *et al* 1985]
Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. 1985. *Generalized Phrase Structure Grammar*. Harvard University Press, Cambridge, Massachusetts.

[Gambäck & Rayner 1992]
Björn Gambäck and Manny Rayner. 1992. "The Swedish Core Language Engine". In L. Ahrenberg, editor, *Papers from the 3rd Nordic Conference on Text Comprehension in Man and Machine*, pp. 71–85, Linköping University, Linköping, Sweden, April. Also available as SICS Research Report R92013, Stockholm, Sweden and as SRI International Technical Report CRC-025, Cambridge, England.

[Horrocks 1987]
Geoffrey Horrocks, editor. 1987. *Generative Grammar*. Longman, London, England.

[Kay 1980]
Martin Kay. 1980. "Algorithmic Schemata and Data Structures in Syntactic Processing". In *Readings in Natural Language Processing*. Morgan Kaufmann, San Mateo, California. Reprint.

[Kay 1989]
Martin Kay. 1989. "Head-Driven Parsing". In *Proceedings of the 1st International Workshop on Parsing Technologies*, Pittsburgh, Pennsylvania.

[Kaplan & Bresnan 1982]
Ronald M. Kaplan and Joan Bresnan. 1982. "Lexical-Functional Grammar: A Formal System for Grammar Representation". In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*, pp. 173–281. The MIT Press, Cambridge, Massachusetts.

[Knuth 1965]
Donald E. Knuth. 1965. "On the Translation of Languages from Left to Right". *Information and Control*, 8(6):607–639.

[Koskenniemi 1983]
Kimmo Koskenniemi. 1983. *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production*. Doctor of Philosophy Thesis, University of Helsinki, Dept. of General Linguistics, Helsinki, Finland.

[Milne 1928]
Alan A. Milne. 1928. *The House at Pooh-Corner*. Methuen & Co Ltd.

[Matsumoto *et al* 1983]
Yuji Matsumoto, Hozumi Tananka, Hideki Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. 1983. "BUP: A Bottom-Up Parser Embedded in Prolog". *New Generation Computing*, 1(2):145–158.

[Pereira & Shieber 1987]
Fernando C. N. Pereira and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis*. Number 10 in Lecture Notes. CSLI, Stanford, California.

[Partee 1987]
Barbara H. Partee, Alice ter Meulen, and Robert E. Wall. 1987. *Mathematical Models in Linguistics*. Kluwer, Boston, Massachusetts.

[Pulman 1991]
Stephen G. Pulman. 1991. "Two Level Morphology". In Stephen G. Pulman, editor, *Eurotra ET6/1: Rule Formalism and Virtual Machine Design Study*, chapter 5. Commission of the European Communities, Luxembourg.

[Pereira & Warren 1980]
Fernando C. N. Pereira and David H. D. Warren. 1980. "Definite Clause Grammars for Natural Language Analysis". *Artificial Intelligence*, 13:231–278.

[Sells 1985]
Peter Sells. 1985. *Lectures on Contemporary Syntactic Theories*. Number 3 in Lecture Notes. CSLI, Stanford, California.

[Shieber 1985]
Stuart M. Shieber. 1985. "Evidence against the context-freeness of natural language". *Language and Philosophy*, 8:333–343.

[Shieber 1986]
Stuart M. Shieber. 1986. *An Introduction to Unification-Based Approaches to Grammar*. Number 4 in Lecture Notes. CSLI, Stanford, California.

[Thomason 1974]
Richmond Thomason, editor. 1974. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, Connecticut.

[Tomita 1986]
Masaru Tomita. 1986. *Efficient Parsing of Natural Language. A Fast Algorithm for Practical Systems*. Kluwer, Boston, Massachusetts.

[Voutilainen *et al* 1992]
Atro Voutilainen, Juha Heikkilä, and Arto Anttila. 1992. "Constraint Grammar of English". Publication 21, Dept. of General Linguistics, University of Helsinki, Helsinki, Finland.

[van Noord 1991]
Gertjan van Noord. 1991. "Head Corner Parsing for Discontinuous Constituency". In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, pp. 114–121, University of California, Berkeley, California, July. ACL.

[van Riemsdijk & Williams 1986]
Henk van Riemsdijk and Edwin Williams. 1986. *Introduction to the Theory of Grammar.* The MIT Press, Cambridge, Massachusetts.

[Wirén 1992]
Mats Wirén. 1992. *Studies in Incremental Natural-Language Analysis.* Doctor of Philosophy Thesis, Linköping University, Dept. of Computer and Information Science, Linköping, Sweden, December.

Easing the use of computers by allowing users to express themselves in their own ("natural") language is a field which has been given much attention during the last years. This booklet introduces natural-language processing in general and the way it is presently carried out at SICS.

The overall goal of any system for natural-language processing system is to translate an input utterance stated in a natural language (such as English or Swedish) to some type of computer internal representation. Doing this requires theories for how to formalize the language and techniques for actually processing it on a machine. How this is done within the framework of the Prolog programming langauge is described in detail.

The booklet is directed to an audience interested in user-friendly computer interfaces in general and natural-language processing in particular. The reader is assumed to have some knowledge of Prolog and of basic (school-book) grammar.

ISRN SICS/P--94/01--SE
ISSN 1100-4665