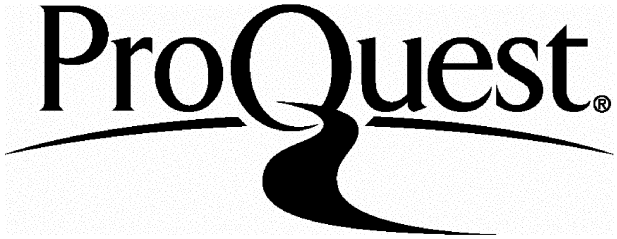
Dependency Parsing

Norman MacAskill Fraser

Thesis submitted for the degree of PhD

University College London

January 1993



Abstract

Syntactic structure can be expressed in terms of either *constituency* or *dependency.* Constituency relations hold between phrases and their constituent lexical or phrasal parts. Dependency relations hold between individual words. Almost all results in formal language theory relate to constituency grammars, of which the phrase structure grammars are best known. In the realm of natural language description, almost all major linguistic theories express syntactic structure in terms of constituency. This dominance carries over into natural language processing, where most parsers are designed to discover the vertical constituency relations which hold between words and phrases, rather than the horizontal dependency relations which hold between pairs of words.

This thesis introduces dependency grammars, their formal properties, their origins in linguistic theory and, particularly, their use in parsers for natural language processing. A survey of dependency parsers — the most comprehensive to date — is presented. It includes detailed discussions of twelve published dependency parsing algorithms. The survey highlights similarities and differences between dependency parsing and mainstream phrase structure grammar parsing. In particular, it examines the hypotheses that (i) it is possible to construct a fully functional dependency parser based on an established phrase structure parsing algorithm without altering any fundamental aspects of the algorithm, and (ii) it is possible to construct a fully functional dependency parser using an algorithm which could not be applied without substantial modification in a fully functional phrase structure parser.

Elements of a taxonomy of dependency parsing are outlined. These include variables in origin, manner, order, and focus of search, as well as in the number of passes made during parsing, techniques for the management of ambiguity, and the use of an adjacency constraint to limit search.

Computer implementations of a number of original dependency parsing algorithms are presented in an Appendix, together with new implementations of established algorithms.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis may bear one name on its title page but it represents an investment of time and effort, of wise advice and honest criticism, of practical support and unfailing love on the part of many people. I am grateful to them all.

First mention must go to Dick Hudson, who has been so much more than just a thesis supervisor. Over the years he has selflessly given me his time, enthusiasm and insight. He has listened patiently to all of my hair-brained ideas and helped me to have fewer of them. My heartfelt thanks go to him and to his family, Gay, Lucy and Alice, who have never failed to respond positively to my all too frequent disruptions of their domestic lives.

I am very grateful to Neil Smith and all members of the Department of Phonetics and Linguistics at University College London for supporting me so well during my time in their midst. Special thanks are due to Mark Huckvale, Monika Pounder, and a number of members of the Word Grammar seminar, including Billy Clark, John Fletcher, and And Rosta. I have also benefited enormously from the support and encouragement I have received as a member of the Social and Computer Sciences Research Group at the University of Surrey. I am grateful to all members of the group, and especially to Nigel Gilbert for enabling me to fit thesis-writing into a hectic research schedule, and to Scott McGlashan for his expert assistance with the LaTeX/ typesetting package. I have gained much from discussions with other people at the University of Surrey, particularly Grev Corbett and Ron Knott.

The finishing touches were added while I was a member of the Speech and Language Division of Logica Cambridge Ltd. I am grateful to Jeremy Peckham for his persistent belief in the value of NLP research and for his practical support, and to Nick Youd, Simon Thornton, Trevor Thomas and Ave Wrigley for daily stimulus.

A significant portion of this thesis is devoted to dissecting other people's dependency parsers. I would not have been able to do so without the help of those individuals who made otherwise unobtainable information available to me. Many of them have read drafts of parts of the thesis, and their comments have been invaluable. They include Doug Arnold, Paulo Baggia, Michael Covington, Peter Hellwig, Gerhard Niedermair, Claudio Rullent, Klaus Schubert, Stan Starosta and Job van Zuijlen.

I have lost track of the number of friends and relations who have helped me by providing practical support, by telling me to get on with it, and by making me laugh. The generous gift of Ian and Mair Bunting, who provided the perfect retreat in which to work without fear of interruption, has hastened the completion of this thesis by an enormous amount. Likewise, the practical support of Jim and Rilla Cannon, whose hospitality knows no bounds. My family have provided the sort of long-distance support which always feels close at hand.

Most of all, I want to thank Sarah for putting up with my nocturnal writing habits, for believing that I really would finish this thing, and for being my friend.

Thank you very much.

Abbreviations

APSG	augmented phrase structure grammar
ATN	augmented transition network
BFP	best fit principle
CCG	combinatory categorial grammar
CD	conceptual dependency
CFPSG	context-free phrase structure grammar
CG	categorial grammar
CNF	Chomsky normal form
DCG	definite clause grammar
DDG	daughter dependency grammar
DG	dependency grammar
DUG	dependency unification grammar
FUG	functional unification grammar
GB	government-binding theory
GPSG	generalized phrase structure grammar
HPSG	head-driven phrase structure grammar
ID	immediate dominance
LFG	lexical-functional grammar
LP	linear precedence
MT	machine translation
NLP	natural language processing
PSG	phrase structure grammar
TAG	tree-adjoining grammar
UCG	unification categorial grammar
WFST	well-formed substring table

Chapter 1

Introduction

> The intuitive appeals of the two theories cannot be discussed, since intuitions are personal and irrational. (Hays 1964: 522)

1.1 Scope of the thesis

There are, in contemporary linguistic theory, two different views of grammatical relations. The first of these sees relations of grammatical dependency as basic: syntactic structures are essentially networks of grammatically related entities. The second view denies grammatical relations basic status, instead seeing them as being derived from more fundamental structures, such as constituent structures. This latter view has predominated throughout most of this century, first in **Immediate Constituent** (IC) analysis (Bloomfield 1914, 1933), and later, from the mid-1950s onwards, in **Phrase Structure Grammar** (PSG) (Chomsky 1957).

The domination of constituency-based approaches has not been limited to theoretical linguistics. In computational linguistics also, the overwhelming majority of proposals which posit a distinct syntactic layer assume that that layer is based on constituent structure rather than dependency structure. This asymmetry can not legitimately be attributed to any established results showing the superiority of one system over the other in respect of descriptive adequacy, or any other substantive function: no such results exist. However, this is not to say that the asymmetry is inexplicable. Although the notion of

grammatical dependency is almost as old as the study of grammar, it has, for most of its existence remained just that: a notion.

The first rigorous formalization of a dependency grammar (DG) came just over thirty years ago (see Gaifman 1965), a few years after the first formalization of the class of PSGs (Chomsky 1956). By the time the formal definition of a DG was published in a wide circulation journal, the corresponding definitions of PSG had been in the public domain for a decade, with large international programmes of research in formal language theory and theoretical linguistics building on a PSG foundation. DG as an explicitly articulated system thus entered an arena in which PSG was already well-established. Given that the earliest published formal accounts of DG established its equivalence (weak and strong) with context-free PSG (CFPSG)[1], there was little incentive to abandon the now familiar and well-understood formalism in favour of the unfamiliar and comparatively less-well understood formalism.

A remarkable situation now obtains. Formal work in DG is virtually frozen in the state it was in around the mid-1960s, with only a handful of groups around the world making any (modest) advances since then (hardly any of which has ever been published in English). In contrast, a much larger — though still modest by PSG standards — number of theoretical linguists continues to assume some version of DG as the foundation of syntactic structure. Unfortunately, almost all linguistic theories based on DG have departed to some extent from the terra firma of formal definition.[2] Since the choice of DG as basic is a minority preference, those making the choice have gone to some lengths to argue the case for DG rather than PSG (for example, Hudson 1984: 92–8, forthcoming; Starosta 1988: 35–6). The opposite is generally not found: proponents of theories based on PSG do not typically support the choice of PSG with arguments for the superiority of PSG over DG (but

[1]Given a definition of equivalence to be described in Chapter 2 below.

[2]The passing allusion to Pullum's (1985) iconoclastic paper 'Assuming some version of the X-bar theory' is thus intentional.

see the debate in Hudson 1980a; Dahl 1980; Hudson 1980b; Hietaranta 1981; and Hudson 1981b for some responses to arguments against PSG).

The principal argument offered by proponents of DG is that PSG approaches introduce a redundant layer of structure. Lexical-Functional Grammar (LFG) offers a particularly clear illustration of this, with its c-structure (constituent structure) and separate f-structure (functional structure), the latter being constructed by reference to the former (Kaplan and Bresnan 1982). In a DG approach a single structure suffices. The position adopted by many advocates of PSG is that it is unnecessary, not to say impossible, to argue against moving targets such as the underformalized versions of DG on offer.

This is to present the issues as being neatly polarized. In fact, most linguists nowadays work with hybrid systems which express both dependency and constituency in a single structure, albeit one which owes more to the PSG tradition than to the DG tradition. The most widespread example is $\bar{X}$ grammar (originally proposed by Harris 1951) which augments a CFPSG by distinguishing one element in each constituent as the **head** of that constituent. However, there are complications here since a number of syntactic theories have been charged with uncritically adopting unformalized versions of $\bar{X}$ theory (Pullum 1985; Kornai and Pullum 1990) — the very charge laid at the door of certain DG theories!

The general paucity of formal results concerning DG carries over from theoretical to computational linguistics. Here DG is scarcely mentioned, far less argued against. In the small number of cases in which it achieves passing mention, the same reasons for not using DG are employed: first, the only existing formal results show the equivalence of DG and CFPSG so there is no incentive to work with the less familiar system; second, almost nothing else is known formally about DG so until such time as additional solid results become available there is no incentive to invest effort in trying to work within that framework.

Let us consider these points in turn. First, then, the equivalence of DG and CFPSG. In their monograph *Linguistics and Information Science* Sparck Jones and Kay provide a brief introduction to DG and then furnish an account for why DG is not mentioned again:

> We have put phrase structure and dependency together in the same class because it is easy to show that the differences between them are trivial from almost every point of view (see Gaifman 1965). It is also possible to write grammatical rules in a suitable notation which describes a single language and which assigns to each sentence of that language both phrase-structure and dependency trees (see Kay 1965; Robinson 1967). In this paper we shall make no further references to dependency grammar, intending what we say about phrase-structure grammar to be understood as applying also to dependency with occasional minor modifications" (Sparck Jones and Kay 1973: 83–4).

Sparck Jones and Kay's observation that it is possible to devise a meta-formalism which includes both dependency and constituency information is useful from a descriptive point of view. However, the point it misses is that the equivalence of the formalisms or the possibility of devising a meta-formalism leaves open the question of whether phrase structure parsing and dependency parsing can be achieved by means of identical algorithms. This is a question which has hardly ever been raised in the literature. Hays' claim that "a phrase-structure parser can be converted into a dependency parser with only a minor alteration" (Hays 1966b: 79) is presented without argument or illustration so its status is, at best, uncertain. A seminal text in computer science bears the title *Algorithms + Data Structures = Programs* (Wirth 1975). It is well understood that a change in data structure may necessitate a change in algorithm if the net effects of the program are to remain constant. "The development of the algorithm...is intimately linked to the choice of an appropriate data struc-

ture" (Goldschlager and Lister 1982: 65). Thus it cannot be taken for granted *a priori* that familiar phrase structure parsing algorithms will map effortlessly into the dependency parsing domain.

The second criticism of DG in computational linguistics is that where DG has been employed, for example in parsing systems, the resulting systems have not been constructed on a principled or even well-defined foundation. Winograd writes:

> The formal theory of dependency grammar has emphasized ways of describing structures rather than how the system's permanent knowledge is structured or how a sentence is processed. It does not address in a systematic way the problem of finding the correct dependency structure for a given sequence of words. In systems that use dependency as a way of characterizing structure, the parsing process is generally of an *ad hoc* nature (Winograd 1983: 75).

Once again, this claim is presented without further argument or evidence.

The absence of empirical data which characterizes these claims is not as surprising as it might first seem when it is understood that the number of dependency parsing systems in existence is severely limited in comparison with the number of phrase structure parsing systems. It is also the case that those descriptions of dependency parsing systems which have appeared in print have, on the whole, been published in relatively obscure sources or have only been circulated privately. Some accounts have been terse to the point of leaving most of the detail unreported. No survey or comparative account of dependency parsers is currently in existence.

One of the chief objectives of this thesis is to fill this gap in the literature by presenting an extensive survey of existing dependency parsing systems, the first such survey to be prepared.

The availability of this survey material presents a unique opportunity to consider from a base of empirical fact how the parsing algorithms employed

in dependency parsing compare with those which are widely used and well-understood in phrase structure parsing. This study focuses on two hypotheses:

Hypothesis 1

It is possible to construct a fully functional dependency parser based directly on an established phrase structure parsing algorithm without altering any fundamental aspects of the algorithm.

This hypothesis is a strong version of Hays' (1966b: 79) claim. It is motivated by Gaifman's definition of strong equivalence between DG and PSG which guarantees some measure of structural correspondence at each point in the DG and PSG parse trees (see Chapter 2 below). However, it is not the strongest possible hypothesis, since it stops short of predicting that a dependency parser can be constructed based on *any* phrase structure parsing algorithm.

Hypothesis 2

It is possible to construct a fully functional dependency parser using an algorithm which could not be used without substantial modification in a fully functional conventional phrase structure parser.

This hypothesis is motivated by an appreciation of the particular way in which DG rules encode information, as compared with the way in which PSG rules encode information.

As I have previously noted, most linguistically motivated DGs have proceeded beyond the limits of what has been defined in a mathematically rigorous way. It is impossible to undertake a survey of dependency parsing systems without encountering some of these devices of unknown formal power. While noting in passing these extensions where relevant, I shall concentrate my analysis on the parsing of the context free backbone of these theories (i.e. that which can be mapped onto a Gaifman grammar). I shall not be concerned in this thesis to make any qualitative judgements between DG and PSG qua descriptive devices.

1.2 Chapter outline

What follows divides conceptually into three parts.

1. Chapter 2 introduces dependency grammar. It presents a formal account of DG and outlines the equivalence relation used to compare DG with PSG. The development of DG from its origins in the classical world through to the present day are charted in the latter part of the chapter.

2. Chapters 3 to 11 present the most detailed review and critique of dependency parsers yet assembled. Chapter 3 describes the growth of the use of DG in computational systems for natural language processing. Chapters 4 to 11 are each devoted to the description and evaluation of a different dependency parser or closely related family of dependency parsers. The chapters are arranged in approximate chronological order; the oldest parser is presented first and the most recent parser is presented last. Needless to say, the development phases of some parsers overlapped so the ordering of chapters must be regarded as no more than a rough guide to the relative age of the systems reported therein.

3. Finally, drawing heavily on the preceding analyses of existing dependency parsers, Chapter 12 sets out some elements of a first taxonomy of dependency parsing, defines some technical vocabulary for the field and specifies the range of relevant variables. The two hypotheses stated above are examined in Chapter 13 in light of the survey of existing dependency parsing algorithms.

Chapter 2

Dependency grammar

"It all depends."
C.E.M. Joad,
BBC Radio 'Brains Trust',
1942–1948

2.1 Overview

Before proceeding with a survey of parsing systems based on DG it is necessary to be clear about exactly what a DG is. One of the dangers when working with a notion like grammatical dependency is that it can come to mean all things to all people. The purpose of this chapter is therefore to furnish an unambiguous definition of DG, to introduce some terminology, and to review where systems approximating to this definition of DG have been employed in theoretical linguistics.

Section 2.2 introduces Gaifman grammars, the only version of DG to be defined with full mathematical rigour. Accordingly, these systems are taken as a stable reference point in this thesis. The formal properties of Gaifman grammars are defined, together with a decision procedure for determining whether or not a given string is accepted or rejected by an arbitrary Gaifman grammar. Alternative conventions for portraying dependency structures diagrammatically are introduced. Although there is insufficient space here to reproduce the rather lengthy proof which establishes the strong equivalence of DG and

PSG, the equivalence relation employed is described and scrutinized.

In practice, very few — if any — linguists have used Gaifman's system in the description of natural language without making use of various augmentations of unknown formal power. These augmentations are flagged in Section 2.3. Those which must necessarily be examined in the course of the survey of dependency parsing systems are described in greater detail in later chapters. Section 2.4 charts the origins and development of DG in linguistic theory.

In Section 2.5, three grammatical formalisms bearing some similarities to DG are identified, namely Case Grammar, Categorial Grammar, and Head-Driven Phrase Structure Grammar. Although a full description of these frameworks is not appropriate here, their basic concepts are introduced and some reasons for excluding them from this study are provided.

2.2 Gaifman grammars

2.2.1 Definitions

The first formal definition of DG was offered by Haim Gaifman (1965). In this section, I present his definition along with illustrative examples.[1]

DEFINITION

A **dependency grammar** Δ is a 5-tuple

$$\Delta = (\mathcal{T}, \mathcal{C}, \mathcal{A}, \mathcal{R}, \mathcal{G})$$

where

1. $\mathcal{T}$ is a finite set of word symbols, i.e. the terminal symbols. For the purposes of exposition, the letters u, v, w, x, y, z, with or without subscripts, will denote members of this set.

[1] In this re-presentation of Gaifman's (1965) work, the logic and substance of his definition is maintained but the manner of exposition has been altered to render the material more transparent.

2. $\mathcal{C}$ is a finite set of category symbols. For the purposes of exposition, the letters *U, V, W, X, Y, Z*, with or without subscripts, will denote members of this set.

3. $\mathcal{A}$ is a set of assignment rules, whose elements are all members of $\mathcal{T} \times \mathcal{C}$. Every word belongs to at least one category and every category must have at least one word assigned to it. A word may be assigned to more than one category.

4. $\mathcal{R}$ is a set of rules which give for each category the set of categories which may derive directly from it with their relative positions. For each category X, there is a finite number of rules of the form

$$X(Y_1, Y_2 \cdots Y_l * Y_{l+1} \cdots Y_n)$$

(where Y_1 to Y_n are members of $\mathcal{C}$) indicating that $Y_1 \cdots Y_n$ may depend on X in the order given, where '$*$' marks the position of X in the sequence. A rule of the form $X(*)$ allows X to occur without any dependents.

5. $\mathcal{G}$ is a subset of $\mathcal{C}$ whose members are those categories which may govern a sentence, i.e. the start symbols.

EXAMPLE

Δ_1 is an example of a dependency grammar, where Δ_1 = ({people, robots, dislike, smart, stupid} , {N, V, A} , {(people, N), (robots, N), (dislike, V), (smart, A), (stupid, A) } , {N(*), N(A,*), V(N,*,N), A(*) }, {V}).

CONVENTION

By convention, the fact that some X is a member of $\mathcal{G}$ may be indicated thus: $*(X)$.

Following this convention, $\mathcal{G}$ of Δ_1 may be represented as $*$(V).

CONVENTION

By convention, $\mathcal{A}$ may be represented as follows: for each distinct category X in $\mathcal{C}$ create a correspondence of the form $X : L$ where L is the set of all words x such that (x, X) is in $\mathcal{A}$.

Thus, $\mathcal{A}$ of Δ_1 may be represented as {N:{people, robots}, V:{dislike} , A:{smart, stupid}}.

CONVENTION

To improve readability, a grammar of type Δ may be represented by writing each member of $\mathcal{G}$ on a line by itself, followed by each member of $\mathcal{R}$ on a line by itself, followed by each member of $\mathcal{A}$ on a line by itself. $\mathcal{T}$ and $\mathcal{C}$ are implicitly defined in $\mathcal{A}$.

Thus, Δ_1 may be represented as follows:

*(V)
N(*)
N(A,*)
V(N,*,N)
A(*)
N:{people, robots}
V:{dislike}
A:{smart, stupid}

The next definition elucidates the relationship between sentences of a language Λ and the grammar of type Δ which generates Λ.

In this definition it is necessary to make reference to occurrences of words or categories in a sequence. An occurrence is an ordered pair $\langle x, i \rangle$, where x is the word or category and i is the position number of x in the sequence. P, Q and R, with or without subscripts denote occurrences of words or categories. If $P = \langle X, i \rangle$ then $S(P)$, the sequence number of P, is defined to be i; P is

said to be of category X.

DEFINITION

A **sentence** $x_1x_2\cdots x_m$ is analyzed by a grammar of type Δ iff the following are true:

1. A sequence of categories $X_1X_2\cdots X_m$ can be formed such that x_i is of category X_i for $1 \leq i \leq m$.

2. A 2-place relation d can be established between pairs of words in $x_1x_2\cdots x_m$. PdQ signifies the fact that P depends on Q, i.e. the relation d holds between P and Q.

   For every d we define another relation d^* where Pd^*Q iff there is a sequence $P_0, P_1\cdots P_n$ such that $P_0 = P$, $P_n = Q$ and P_idP_{i+1} for every $0 \leq i \leq n-1$.

   The relation d is constrained in the following ways:

   (a) For no P, Pd^*P.

   (b) For every P, there is at most one Q such that PdQ.

   (c) If Pd^*Q and R is between P and Q in sequence (i.e. either $S(P) < S(R) < S(Q)$ or $S(P) > S(R) > S(Q)$), then Rd^*Q.

   (d) The whole set of occurrences is connected by the relation d.

3. If P is an occurrence of x_j and if the occurrences that depend on it are $P_1, P_2\cdots P_n$, also, if P_h is an occurrence of X_{i_h} where $h = 1\cdots n$, and the order in which these words occur in the sentence is $x_{i_1}, x_{i_2}, \cdots, x_{i_k}, x_j, x_{i_{k+1}}, \cdots, x_{i_n}$, then $X_j(X_{i_1}\cdots X_{i_k} * X_{i_{k+1}}\cdots X_{i_n})$ is a rule of R. In the case that no occurrence depends on P, $X_j(*)$ is a rule of R.

4. The occurrence which governs the sentence (i.e. which depends on no other occurrence) is an occurrence of a word whose category is a member of $\mathcal{G}$.

The structure corresponding to a sentence of a language generated by a grammar of type Δ is called a dependency tree.

DEFINITION

A **dependency tree** for a sentence $x_1 \cdots x_n$ consists of the string of categories $X_1 \cdots X_n$, together with the relation d.

DEFINITION

A **language** is weakly generated by a dependency grammar iff for every sentence in that language there is a corresponding dependency tree and no dependency tree exists for a sequence of words which is not a sentence. A language is strongly generated by a dependency grammar iff it is weakly generated by that dependency grammar and, for every syntactically correct interpretation, and only for these, there are corresponding dependency trees.

The above definitions can be summarized informally as follows. In the structure corresponding to a sentence of a language generated by a dependency grammar of type Δ:

1. one and only one occurrence is independent (i.e. does not depend on any other);
2. all other occurrences depend on some element;
3. no occurrence depends on more than one other; and
4. if A depends directly on B and some occurrence C intervenes between them (in linear order of string), then C depends directly on A or on B or on some other intervening element (Robinson 1970: 260).

To aid discussion, I shall adopt the following terminology. All occurrences of words in a sentence shall be called **words**. Where the intention is to refer to words in the lexicon, this will be stated explicitly. The single independent word in a sequence (i.e. the word which depends on no other) shall be called the **root**. One word W_1 is said to be a **subordinate** of another word W_2 if W_1 depends on W_2 or on another subordinate of W_2, i.e. W_1 depends directly or indirectly on W_2. The word on which another word depends shall be called its **head**. The requirement that a head-dependent pair either be next to each other or separated by direct or indirect dependents of themselves (point 4 above) is known as the **adjacency constraint**.

EXAMPLE

Given these definitions, the sentences in (1) belong to the language defined by Δ_1, whereas the sequences in (2) are outside of that language. (By convention, sequences which are not well-formed in respect of a particular grammar are prefixed by '*').

(1) a People dislike robots.
b Stupid people dislike smart robots.
c Smart robots dislike people.
d People dislike smart people.

(2) a *Smart people dislike.
b *Stupid dislike robots.
c *Stupid robots.
d *Robots people dislike.
e *Robots smart dislike people.

Example (2a) is ill-formed because *dislike* is a V, and Vs require two dependents, one preceding and one following. In this case, no following dependent is present. Example (2b) is ill-formed because all of the words are not connected together by dependency. The sequence is divided into two parts: *stupid* (which requires a head) and *dislike robots* (which requires a preceding dependent of

category N for *dislike*). None of the words in (2c) is missing a dependent. However, the independent word *robots* is of category N, but only words of category V may govern a sentence. In example (2d), none of the words is missing a dependent and the independent element *dislike* belongs to the required category V. However, the dependents of V are required to occur one on either side of V, whereas here they both occur before it. Example (2e) is ill-formed because of the inappropriate position of *smart*. Either it is a dependent of *robots*, in which case it should precede that word, or it is a dependent of *people*. If it is a dependent of *people* then it precedes it as it ought, but *smart* and *people* are separated by the word *dislike*, which is dependent on neither.

I shall henceforth refer to dependency grammars of type Δ as **Gaifman Grammars**.

2.2.2 A recognizer for Gaifman grammars

So far, I have characterized Gaifman grammars in terms of constraints on the well-formedness of grammar rules and dependency structures. In this section I describe a decision procedure — a **recognizer** — which accepts all and only the well-formed strings of the language described by a Gaifman grammar. The recognizer is based on one described by Hays (1964: 516–17).

The principal data structure used by the recognizer is a table. To determine whether or not a string is generated by a Gaifman grammar Δ proceed as follows:

1. Starting from 1, and counting upwards in units of 1, assign an integer to each word in the string, working from left to right. The integer assigned to a word shall be known as the *position* of that word. Let Max equal the position of the rightmost word.

2. Set up a table, having Max positions, numbered from 1 to Max. A *cell* $[a, b]$ shall occupy all the positions from P_a to P_b, where $1 \leq a \leq b \leq Max$.

3. For each word W_i in the string retrieve all the classes X_1 to X_n assigned to that word by assignment rules of the form $W : \{X_1, ..., X_n\}$ in Δ. If P_i is the position of W_i, write X_1 to X_n in the table at cell $[i, i]$.

4. For each word class X at cell $[j, j]$ in the table ($1 \leq j \leq Max$) determine whether a rule of the form $X(*)$ exists in Δ. If so, insert $X(*)$ in the table at cell $[j, j]$.

5. Let V be a variable. Set $V = 2$.

6. Consider each sequence of V adjacent cells in the table. For each sequence which consists of exactly one word class symbol X and V–1 trees, arranged in the order

$$Y_1, ..., Y_i, X, Y_j, ..., Y_{V-1}$$

search in Δ_1 for a corresponding rule of the form:

$$X(Z_1, ..., Z_i, *, Z_j, ..., Z_{V-1})$$

If the root of each tree Y_n in the table is identical to each dependent Z_n in the grammar rule then if Y_1 is located at cell $[Y_{1_{left}}, Y_{1_{right}}]$ and Y_{V-1} is located at cell $[Y_{V-1_{left}}, Y_{V-1_{right}}]$, insert a new tree in the table occupying cell $[Y_{1_{left}}, Y_{V-1_{right}}]$. The form of the new tree should be as follows:

$$X(Y_1, Y_2, ..., Y_i, *, Y_j, ..., Y_{V-1})$$

7. If $V = Max$ then go to step 8, otherwise increment V and go to step 6.

8. If a tree exists in the table occupying cell $[1, Max]$ then succeed if the root of the tree is of type X and a rule of the form $*(X)$ exists in Δ. Otherwise fail.

Hays presents his algorithm informally, so it has been necessary to reconstruct some of the details in the above account.

A Prolog implementation of this recognition algorithm can be found in the file `hays_recognizer.pl` in Appendix A.3.

Hays also outlines a generative procedure for enumerating all the strings generated by a Gaifman grammar (Hays 1964: 514–15). A Prolog implementation of a reconstructed version of Hays' procedure can be found in the file `hays_generator.pl` in Appendix A.3.

2.2.3 Representing dependency structures

There are at least three conventions for presenting dependency structures diagrammatically: *stemmas*, *tree diagrams* and *arc diagrams*.

The first representational scheme — due to Tesnière (1959) — presents words as nodes in a graph which is known as a **stemma** (see Figure 2.1, for example). Dependencies between word occurrences are signalled by links between nodes. By convention, heads are located nearer the top of the diagram than their dependents. The first occurrence in a sentence is positioned furthest to the left in a diagram and the nth occurrence appears to the right of the n–1th occurrence and to the left of the n+1th occurrence. For simplicity, category labels are usually omitted from diagrams of all types.

Although stemmas contain the appropriate amount of information, they can sometimes prove to be difficult to read, especially when the sentences represented are long and involve a lot of alternation between left-pointing and right-pointing dependencies.

In the second type of diagram, exemplified in Figure 2.2, dependency is represented by the relative vertical position of nodes in a tree; if a line connects a lower node to a higher node then the symbol corresponding to the lower node depends on the one corresponding to the higher node. I shall call diagrams of this kind **tree diagrams**. They are also known as **D-markers**.

The third diagrammatic convention represents dependency relations by

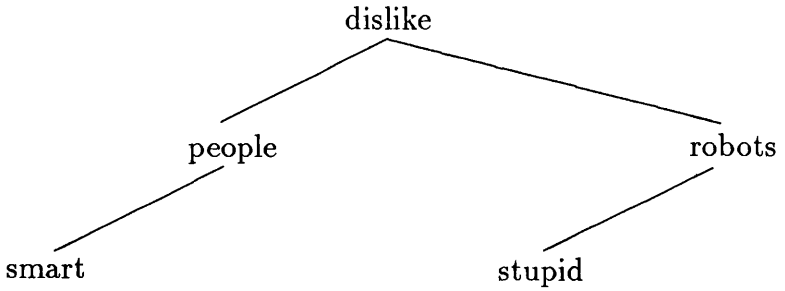


Figure 2.1: stemma for *Smart people dislike stupid robots*

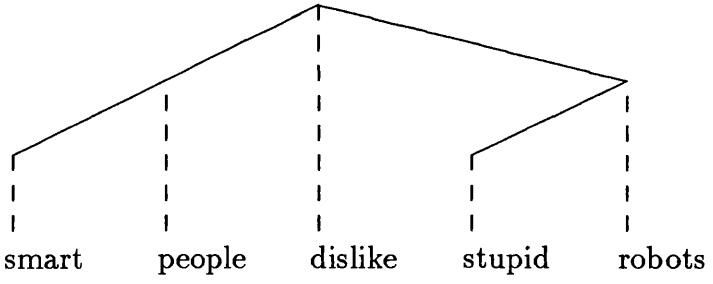


Figure 2.2: tree diagram (D-marker) for *Smart people dislike stupid robots*

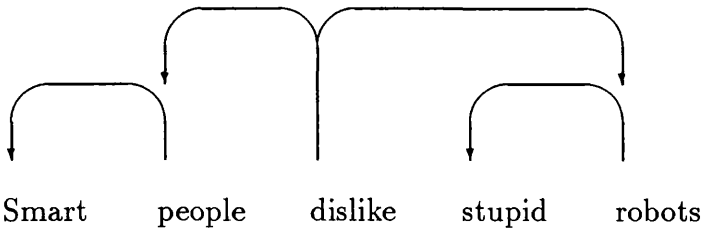


Figure 2.3: arc diagram for *Smart people dislike stupid robots*

means of directed arcs. I shall adopt the convention of directing arcs from heads to dependents, although (unfortunately) there is no generally accepted convention and it is not unusual to find examples in the literature of arcs being oppositely directed. I shall refer to diagrams of this kind as **arc diagrams**. Figure 2.3 is equivalent to Figures 2.1 and 2.2 in the information it expresses.

Some authors (such as Matthews 1981) draw arc diagrams with the arcs *below* the symbols in the sentence rather than above them as shown here. Hudson sometimes divides the arcs so that those having a designated function appear below the sentence symbols, whilst the rest appear above them (Hudson 1988b: 202; page 189 below).

The adjacency constraint is satisfied in the sentence *Smart people dislike stupid robots*, as can be seen in the dependency structure variously represented in Figures 2.1, 2.2 and 2.3. The constraint is violated in the dependency structure shown in Figure 2.4.

In Figure 2.4, *Stupid* violates the constraint. *stupid* is separated from its head *robots* by *dislike* which depends on neither *stupid* nor *robots*, neither is it a subordinate of *stupid* nor *robot*. In a tree diagram, the dotted line which connects a word with its node is called its **projection**. Note that in Figure 2.2, links and projections do not intersect. Such tree diagrams and their corresponding syntactic structures are said to be **projective**. In Figure 2.4 a link and a projection intersect at precisely the point where ill-formedness was detected. Diagrams like Figure 2.4, and the corresponding syntactic structures

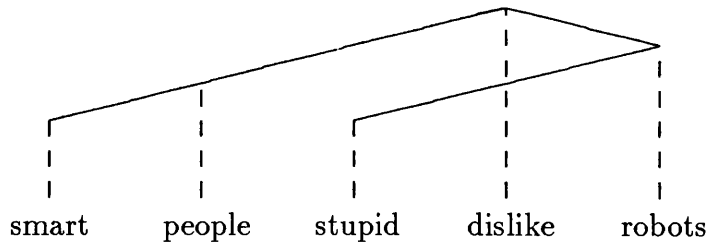


Figure 2.4: dependency tree for **Smart people stupid dislike robots*

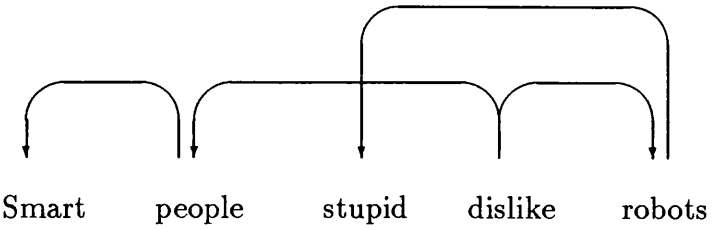


Figure 2.5: arc diagram for **Smart people stupid dislike robots*

are said to be **non-projective**.

The vocabulary of projectivity is rooted in the imagery of tree diagrams. I shall henceforth make use of the more neutral terms **adjacent** and **non-adjacent**.

The arc diagram corresponding to Figure 2.4 is shown in Figure 2.5. Notice that arcs never cross in arc diagrams of structures which satisfy the adjacency constraint, whereas arcs do cross where the structures violate the adjacency constraint. (The only exception to this generalization is discussed below).

In general, I shall use arc diagrams to represent dependency structures; when describing a particular dependency system reported in the literature I shall use the representation normally employed by proponents of that system.

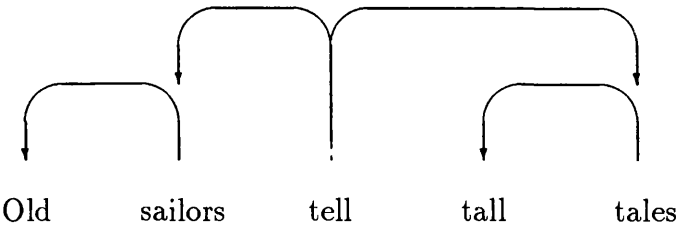


Figure 2.6: Dependency structure of *Old sailors tell tall tales*

2.2.4 The generative capacity of Gaifman grammars

As well as providing a formally explicit definition of one class of DG, Gaifman went on to investigate the generative capacity of the class. He did this by comparing his DG with phrase structure grammar.

He concluded that for every DG there is a strongly equivalent CFPSG and for a subclass of CFPSGs (in which every phrase is a projection of a lexical category) there is a strongly equivalent DG. His proof is too lengthy to reproduce here; it can be found in Gaifman (1965). Definitions of strong equivalence between the two systems can be found in Hays (1961b) and in Gaifman (1965: 320–25).

Let a **subtree** be a connected subset of a dependency tree. (This is what Pickering and Barry (1991) have recently called a 'dependency constituent'.) Let a **complete subtree** consist of some element of a tree, plus **all** other elements directly or indirectly dependent on it. Thus, the dependency tree in Figure 2.6 includes the subtrees shown in Table 2.1. Of these, only those shown in Table 2.2 are complete subtrees.

A phrase structure and a dependency structure, both defined over the same string, **correspond relationally** if every constituent is coextensive with a subtree and every complete subtree is coextensive with a constituent. Two structural entities are **coextensive** if they refer to exactly the same elements in a string.

Let each subtree have a **label**, where the label is that word in the subtree

Old
Old sailors
Old sailors tell
Old sailors tell tall tales
sailors
sailors tell
tell
tell tall tales
tell tales
tall
tall tales
tales

Table 2.1: Subtrees in Figure 2.6

Old
Old sailors
Old sailors tell tall tales
tall
tall tales

Table 2.2: Complete subtrees in Figure 2.6

LABEL	SUBTREE
Old	Old
sailors	Old sailors
tell	Old sailors tell tall tales
tall	tall
tales	tall tales

Table 2.3: Complete subtree labels in Figure 2.6

which depends on no other word in the same subtree. Labels for the complete subtrees of the dependency tree shown in Figure 2.6 are given in Table 2.3.

Let each phrasal constituent in a PSG also have a label, where the label is conventionally understood (for example, the label of a noun phrase is often given as 'NP', etc).[2]

In dependency theory, a string is said to **derive from** the label of the corresponding complete subtree. In phrase structure theory, a string is said to **derive from** the label of the corresponding constituent. A label **accounts for** the set of strings that derive from it. Two labels are **substantively equivalent** if they account for the same set of strings.

A phrase structure and a dependency structure **correspond** if (i) they correspond relationally and (ii) every complete subtree has a label which is substantively equivalent to the label of the coextensive constituent.

A DG is **strongly equivalent** to a PSG if (i) they have the same terminal alphabet, and (ii) for every string over that alphabet, every structure attributed by either grammar corresponds to a structure attributed by the other.

Let us consider, by way of example, the ambiguous sentence (3), the two phrase structure interpretations of which are shown in Figures 2.7 and 2.8. The linguistic plausibility of these analyses is not an issue here.)

(3) They are racing horces.

[2]All subtree and phrase labels must be unique within each sentence. If necessary this can be effected by providing labels with unique integer subscripts.

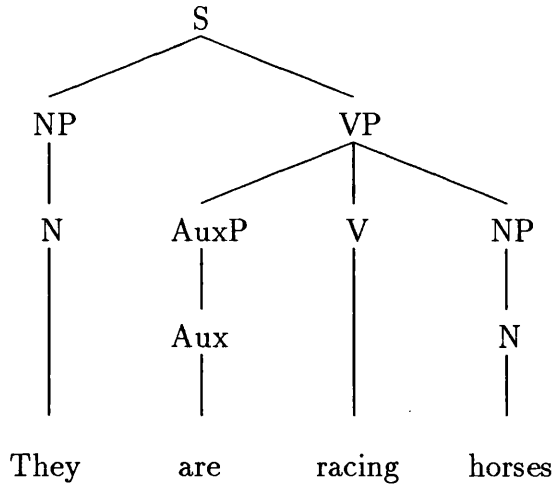


Figure 2.7: First phrase structure analysis of *They are racing horces*

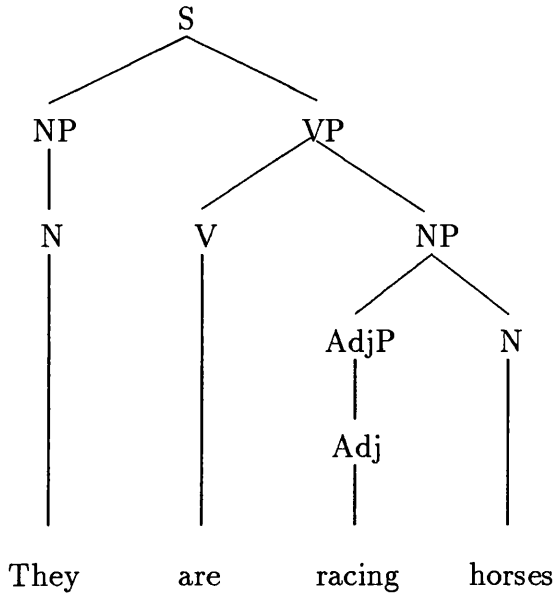


Figure 2.8: Second phrase structure analysis of *They are racing horces*

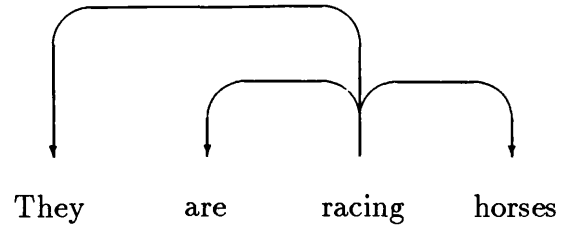


Figure 2.9: Dependency structure for *They are racing horses.* The sentence root is *racing.*

LABEL	SUBTREE
they	they
are	are
	they racing
	they are racing
racing	they are racing horses
	are racing
	are racing horses
	racing
	racing horses
horses	horses

Table 2.4: Subtrees and complete subtrees in the DG analysis of the sentence *They are racing horses* shown in Figure 2.9. Only complete subtrees are labelled.

Now consider the dependency structure in Figure 2.9. This includes the subtrees shown in Table 2.4.

The constituents in Figure 2.7 are shown in Table 2.5 (ignoring the initial category assignments).

Since every constituent in Figure 2.7 is coextensive with a subtree in Figure 2.9 and every complete subtree in Figure 2.9 is coextensive with a constituent, the structures correspond relationally. Since it is also the case that every complete subtree has a label which is substantively equivalent to the label of the coextensive constituent, the structures correspond. Close examination of Figure 2.8 reveals that it also corresponds relationaly to Figure 2.9.

LABEL	CONSTITUENT
NP	they
S	they are racing horses
AuxP	are
VP	are racing
VP	are racing horses
NP	horses

Table 2.5: Constituents in the phrase structure analysis of the sentence *They are racing horses* shown in Figure 2.7

However, only Figures 2.7 and 2.9 share substantively equivalent labellings so only these structures can be said to correspond.

2.3 Beyond Gaifman grammars

In presenting his work on PSGs, Chomsky frequently and explicitly represented them as a formalization of the structuralist Immediate Constituent model (e.g. Chomsky 1962). This claim has recently been contested by Manaster-Ramer and Kac (1990), thus highlighting some of the difficulties inherent in trying to formalize a pre-existing linguistic notion faithfully.

The issues are somewhat clearer in the case of DG, since Gaifman, as author of the formalization, makes no claims regarding its relation to any existing notion other than that embodied in a RAND Corporation machine translation program. Hays, on the other hand, represents Gaifman's work as being a formalization of the linguistic notion of dependency. For example, following a discussion of the different linguistic notions underlying IC theory and dependency theory in his 1964 *Language* paper, his summary of what is to follow includes the following statement:

> Section 2 presents a formalism for the theory, identifying the components of *any* dependency grammar (Hays 1964: 512, my emphasis).

I have been unable to find any discussions anywhere in the literature which investigate this assertion by reference to actual linguistic theories which claim to be based on some notion of dependency.

What is noticeable is that few of the self-proclaimed dependency-based theories of language have made use of Gaifman's formalism. This contrasts sharply with the uptake of Chomsky's PSG formalism, and particularly CF-PSG. The only DGs which incorporate a more or less intact version of Gaifman grammar are those which use it as the base component in a transformational grammar (Hays 1964: 522–4; Robinson 1970) or as the transcription system on one stratum of a stratificational grammar (Hays 1964: 522–4). Otherwise, alternative quasi-formalisms are employed.

It is common to find versions of DG which make use of complex feature structures instead of or as well as word category labels, with dependency rules being allowed to manipulate features in arbitrary ways (e.g. Starosta 1988; Covington 1990b). Consider the following illustrative example of a dependency rule for intransitive verbs which enforces subject-verb agreement (adapted from Covington 1990b: 234):

$$\begin{bmatrix} category: & verb \\ person: & X \\ number: & Y \end{bmatrix} \left(\begin{bmatrix} category: & noun \\ person: & X \\ number: & Y \\ case: & nominative \end{bmatrix} , * \right)$$

Here the head is of syntactic category 'verb', of person 'X' and number 'Y'. Its single dependent must be a preceding nominative case noun, also of person 'X' and number 'Y'. 'X' and 'Y' are variables over feature values.

This kind of augmentation could easily be formalized as an extension to Gaifman's definition of DG. So long as the feature structures are simply arrangements of symbols drawn from a finite set, the generative power remains unchanged. The proof is trivial: any arrangement of features may be 'frozen' and treated as though it were an atomic symbol.[3] This is directly analogous to

[3]Obviously, this is just a sketch of the proof. The proof itself would first have to define precisely the notational extension to Gaifman's formalism.

what happens when a PSG is augmented by the addition of feature structures. Gazdar has summarized this as follows:

> If we take the class of context-free phrase structure grammars and modify it so that (i) grammars are allowed to make use of finite feature systems and (ii) rules are permitted to manipulate the features in arbitrary ways, then what we end up with is equivalent to what we started out with (Gazdar 1988: 69).

Unfortunately, all of the DGs which introduce feature structures also introduce other extensions, whose effects on the generative capacity of the grammars are unknown. For example, in Hudson's Word Grammar, a word may depend on more than one head (Hudson 1990: 113–20). In Starosta's Lexicase, certain complete subtrees (e.g. prepositional structures in English) have two roots, or rather, a single root which is the union of the features of two of the words included in the subtree (Starosta 1988: 232–4). Hudson offers a revised version of the adjacency constraint whose definition includes a reference to multiple heads (Hudson 1990: 117), while Pericliev and Ilarionov (1986), Sgall et al. (1986), Schubert (1987), and Covington (1990b) advocate abandoning the adjacency constraint altogether!

A thesis of this kind can not proceed without giving some attention to these theoretical extensions. However, as previously indicated, these features must be regarded as lying on the periphery of the study.

2.4 Origins in linguistic theory

The concept of grammatical dependency is found in some of the earliest known grammars, for example those of the Greek scholars of the Alexandrian School, and especially Dionysius Thrax (c.100 B.C.) whose work drew heavily on the Stoic tradition of linguistic studies. Thrax's *Téchnē grammatikē* was the inspiration for the grammar of the later Alexandrian scholar Apollonius Dysco-

lus (second century A.D.) whose work "foreshadowed the distinction of subject and object and of later concepts such as government...and dependency" (Robins 1979: 37). The work of Thrax and Apollonius was further developed by a number of Latin grammarians, most notably Priscian (c. 450 A.D.). An independent (earlier) strand of grammatical study was pursued by the Sanskrit grammarians, most notably Pāṇini (some time between 600 and 300 B.C.). In Pāṇini's grammar "the verb, inflected for person, number, and tense, was taken as the core of the sentence... Other words stood in specific relations to the verb, and of these the most important were the nouns in their different case inflexions" (Robins 1979: 145).

Particularly clear early articulations of the central concepts of dependency can be found in the writings of the medieval Arabic grammarians, especially those of the Basra and Baghdad schools. In Arabic grammar, a governor (*ʿâmil*) was said to govern (*ʿamila* lit. 'do, operate') a governed (*maʿmûl*). Many of the details of modern DG are made explicit for the first time in the writings of Ibn Al-Sarrâj (died 928A.D.). For example, a word may not depend on more than one other. Sarrâj writes:

> It is not permitted to have two governors governing a single item. (Owens 1988: 43[4])

Heads and dependents were required to be adjacent. Again, Sarrâj writes:

> The separation between the governor and the governed by something not related to either is disliked. (Owens 1988: 46)

This finds support in the writings of Jurjânî (died 1078), who insists that:

> You cannot separate a governor and a governed with a foreign element. (Owens 1988: 49)

In common with modern versions of dependency theory, governors could have many dependents, although dependents could have only one head. Dependency

[4]All quotations use Owen's translation and reference Owens (1988) rather than the original sources.

was unidirectional and there was no interdependence. The mediaeval Arabic grammarians also observed that, for Arabic at least, governor-governed was the unmarked word order. A detailed guide to mediaeval Arabic grammar can be found in Owens (1988).

Dependency is also found in the work of mediaeval European scholars such as the modistic and speculative grammarians, and especially, in the work of Martin of Dacia and Thomas of Erfurt (more details of their work can be found on page 217ff below). According to Herbst et al. (1980: 33) who quote Engelen (1975: 37), some of the central ideas of DG were used in Germany by Meiner in the eighteenth century and later by others including Behaghel, Bühler and Neumann.

Most commentators agree that the most significant contribution to the development of DG was made in the 1950s by the Frenchman Lucien Tesnière. Tesnière was the first person to develop a semi-formal apparatus for describing dependency structures. Tesnière's ideas were initially collected in a slim and rather programmatic volume (Tesnière 1953) which was not very well received by reviewers (for example, see Garey 1954). Tesnière died in 1954 but Jean Fourquet edited his unpublished works into a single volume — *Éléments de Syntaxe Structurale* — which was published in 1959. This book presents a coherent and comprehensive account of Tesnière's work in DG.

Tesnière's posthumous volume consists of three parts labelled 'la connexion' (dependency), 'la jonction' (coordination) and 'la translation' (word class transformation). He argued that whereas all other constructions could be analysed in terms of word-word dependencies, coordinate constructions could not. This is now the standard view amongst dependency grammarians.[5] (A few dependency grammarians — including Mel'čuk (1988: 26ff) — hold that coordinate constructions can also be analyzed in terms of dependency). The 'Connexion' section of the book presents in axiomatic fashion many of the

[5]Brief descriptions of some approaches to coordination in DG can be found on pages 139, 168, and 188. For a useful overview see Hudson (1988b).

principles which have come to define and to distinguish DG. For example (my translation, Tesnière's emphases):

> The structural connections establish relations of **dependence** between the words. As a rule, each link unites a **superior** term with an **inferior** term.[6]
>
> The superior term is called the **regent**. The inferior term is called the **subordinate**.[7]
>
> The upward relation can be expressed by saying that the subordinate **depends** on the regent, and the downward relation by saying that the regent **commands** or **governs** the subordinate.[8]
>
> In principle, a subordinate can only depend on **a single** regent. In contrast, a regent can command several subordinates.[9]
>
> The node formed by the regent which commands all the subordinates of the phrase is the **node of nodes** or **central node**. It is the core of the phrase, of which it assures the structural unity by tying the separate elements into a single structure. It is identified with the phrase.[10] (Tesnière 1959: 13–15)

In a footnote on Tesnière's page 15 he tells how he first conceived of the idea of the stemma in June 1932. He started using it in his private research in 1933 and in his publications in 1934. In 1936, whilst on a trip to the U.S.S.R., he discovered that he was not the only person to have this idea.[11] Ušakov, Smirnova and Ščeptova had published an article using stemmas as early as 1929. Barkhudarov and Princip had done likewise in 1930, and Kručkov and Svetlaev had used stemmas in a book published in early 1936. In spite of this — in Western Europe at least — Tesnière is usually named as the originator of

[6]Les connexions structurales établissent entre les mots des rapports de **dépendance**. Chaque connexion unit en principe un terme **supérieur** à un terme **inférieur**.

[7]Le terme supérieur reçoit le nom de **régissant**. Le terme inférieur reçoit le nom de **subordonné**.

[8]On exprime la connexion supérieur en disant que le subordonné **dépend** du régissant, et la connexion inférieur en disant que le régissant **commande** ou **régit** le subordonné.

[9]En principe, un subordonné ne peut dépendre que d'**un seul** régissant. Au contraire un régissant peut commander **plusiers** subordonnés.

[10]Le nœude formé par le régissant qui commande tous les subordonnés de la phrase est le **nœud des nœuds** ou **nœud central**. Il est au centre de la phrase, dont il assure l'unité structurale en en nouant les divers éléments en un seul faisceau. Il s'identifie avec la phrase.

[11]"J'ai eu la joie de constater que l'idée du stemma y avait germé de façon indépendante".

DG as an explicit system for linguistic description. Certainly it was Tesnière's work which did more than anyone else's to publicize DG. Had his volume been published any time other than in the immediate aftermath of the publication of Chomsky's *Syntactic Structures*, DG might have been taken seriously by a much wider audience.

Amongst the most influential of Tesnière's ideas were those relating to *valency*. The valency of a verb is its potential for having other words depend on it. Thus, an intransitive verb takes one dependent, a transitive verb takes two, a ditransitive three, etc. In addition to these complements which must be present in a well-formed structure, a verb may also take some number of adjuncts. Complements subcategorize the verb, whereas adjuncts modify it. The term 'valency' was borrowed from molecular physics where it is used to describe the attractive potential of a molecule.[12] Tesnière is often cited as the originator of the term 'valency' in linguistics but, according to Schubert (1987: 61), it can be found in the earlier writings of Kacnel'son (1948: 132) ['sintaksičeskaja valentost'] and de Groot (1949: 111) ['syntactische valentie']. Baum (1976: 32) claims that Hockett (1958: 249) uses the term 'valence' independently of Tesnière.

The relationship between valency and dependency is rather opaque. Early dependency theorists tended to concentrate on formal issues and to see verbal valency as just one specific example of the general case of dependency — in other words, all words have a valency. Valency theorists, on the other hand, concentrate on the pivotal role played by the main verb in natural language sentences. They tend to focus particularly on the case relations—in the general sense of Fillmore (1968)—of the verb. Two largely disjoint research communities have sprung up. In a recently published bibliography of

[12] "On peut ainsi comparer le verbe à une sorte d'atome crochu susceptible d'excercer son attraction sur un nombre plus ou moins élevé d'actants, selon qu'il comporte un nombre plus ou moins élevé de crochets pour les maintenir dans sa dépendance. Le nombre de crochets que présente un verbe et par conséquent le nombre d'actants qu'il est susceptible de régir, constitue ce que nous appellerons la **valence** du verbe" (page 238).

valency grammar [valenzbibliographie] which includes 2377 entries, only 294 are indexed as relating to 'dependency' (Schumacher 1988). It is somewhat difficult to see why these separate communities still exist since a number of linguistic theories appear to bridge the perceived gap quite effectively (e.g. Heringer 1970; Anderson 1977; Starosta 1988).

The influence of Tesnière's work has reached into almost every part of the world where language is studied, but the effects have not always been the same. In Tesnière's native France and throughout the Romance language areas his insights have been frequently applauded but seldom adopted. In Schubert's words:

> In works written in French, Spanish, Italian and other Romance languages, Tesnière is referred to as a classic of linguistics, but hardly anybody has taken up the essence of his ideas and written for example a dependency syntax of French or a valency dictionary of Spanish (Schubert 1987: 22).

Maurice Gross is sometimes cited as a French dependency grammarian but he has not been active in the DG field since the early 1960s when he briefly examined dependency grammars from a computational point of view (Gross 1964).

Tesnière's work had more influence in Germany (East and West), where it was judged to be more appropriate for describing German word order variation and agreement patterns than the rather inflexible PSGs available in the 1960s and 70s. One of the first large-scale uses of dependency in the description of German was by Hans-Jürgen Heringer, who combined constituency and dependency in a single representation (Heringer 1970). Two schools arose within dependency-based studies of language in the late 1960s at Leipzig and at Mannheim. The Leipzig school — which is chiefly associated with Gerhard Helbig — concentrated on the compilation of valency dictionaries for German verbs (Helbig and Schenkel 1969), adjectives (Sommerfeldt and Schreiber 1974), and nouns (Sommerfeldt and Schreiber 1977). The Mannheim school under

Ulrich Engel and Helmut Schumacher began by producing an alternative valency dictionary of German verbs (Engel and Schumacher 1976) but they progressed to apply the insights of DG in the general description of languages. Engel's grammar of German (Engel 1977) was possibly the first attempt to describe all of the major phenomena of a single language within a dependency framework. Other German dependency theorists include Jürgen Kunze and his colleagues in East Berlin (e.g. Kunze 1975) and Heinz Vater who developed a transformational generative version of DG (Vater 1975).

From Germany, interest in DG spread throughout Northern Europe, often being promulgated by Germanists. It was introduced in Finland by Kalevi Tarvainen (Tarvainen 1977), in Sweden by Henrik Nikula (Nikula 1976), and in Denmark by Catharine Fabricius-Hansen (Fabricius-Hansen 1977).

In Great Britain, John Anderson (also a Germanist) developed 'Case Grammar', a combination of DG and localist case (Anderson 1971; Anderson 1977). More recently, Anderson has been involved in the development of a dependency-based theory of phonology (Anderson and Durand 1986). Richard Hudson's theory of 'Daughter Dependency Grammar' (DDG) (Hudson 1976) grew out of his earlier research in Systemic Grammar (Hudson 1971) and was a combination of constituency and dependency. He subsequently abandoned DDG in favour of a new theory, 'Word Grammar' (Hudson 1984), which is based on dependency alone. Hudson has recently published what is probably the first major theoretically-motivated DG of English (Hudson 1990). In addition to these dependency theories, at least two British scholars have used DG in syntax textbooks (Matthews 1981 and Miller 1985) and Rodney Huddleston has published two grammars of English which incorporate insights from DG (Huddleston 1984; Huddleston 1988).

In the early 1960s a number of Soviet scholars — including Sergei Fitialov and Igor Mel'čuk — used dependency as the basis of machine translation systems. Since then, Mel'čuk (now at the University of Montreal)

has been developing his dependency-based 'Meaning-Text Model' of language (Mel'čuk and Žolkovkij 1970; Mel'čuk 1979; Mel'čuk 1988). Petr Sgall's group at Charles University in Prague has produced a general theory of language structure called 'Functional Generative Description' in which dependency is basic and constituent structure plays no part (Sgall et al. 1986). I am aware of some ongoing dependency research in Bulgaria but I have not seen any English papers other than those by Pericliev and Ilarionov (1986) and Avgustinova and Oliva (1990). A number of slavists working in the West have also used some version of DG in their work, e.g. David Kilby (Atkinson et al. 1982) and Johanna Nichols (Nichols 1978; Nichols 1986).

It is worth pausing to reflect that so far in our discussion of the development of DG we have considered only European scholars (with the exception of Pāṇini, the mediaeval Arabic grammarians, and Johanna Nichols who is based at Berkeley). If constituency grammar can be regarded as the product of North American scholarship (and especially of Bloomfield and Chomsky) then DG can be regarded as a distinctively European development. However, although the vast majority of work in DG has been carried out in Europe, some work has been done in North America and Japan.

The main figures associated with DG in North America are David Hays, Jane Robinson, and Stanley Starosta. Hays worked for the RAND Corporation in the early 1960s, on a large Russian-English machine translation project. He explored the uses of DG for machine translation and also investigated the formal properties of DGs. His work is described in more detail in Chapter 4. In the late 1950s, Haim Gaifman, had collaborated with Bar-Hillel and Shamir in a study of Categorial Grammars and PSGs which proved for the first time the generative equivalence of the two formalisms (Bar-Hillel et al. 1960). In the early 1960s, while he was based in the Mathematics Department of the University of California at Berkeley, Gaifman undertook consultancy work for the RAND Corporation. It was there, while working with Hays, that

Gaifman carried out the work described at the beginning of this chapter. His seminal paper *Dependency systems and phrase structure systems* appeared as a RAND internal report in May 1961, although it was not published in a major journal until 1965, a year after Hays' article making Gaifman's work accessible to linguists had appeared in *Language* (Hays 1964). Robinson's work in DG was carried out at the end of the 1960s while she was employed at IBM's Watson Research Center. The main objective of her work was to explore the ways in which Fillmorean case grammar could fit into a transformational framework. Her conclusion was that a transformational grammar should have a DG rather than a PSG base component (Robinson 1969; Robinson 1970). The work of Vater mentioned above (Vater 1975) was a development of Robinson's ideas.[13] The largest single contribution to DG in North America in recent years has been made by Starosta at the University of Hawaii. Since the early 1970s Starosta has been developing a dependency-based theory of language called 'Lexicase' (Starosta 1988). Lexicase has been used in the description of around fifty different languages, many of them so-called 'exotic' languages. It is unlikely that any other dependency-based theory has been so widely field-tested. For that matter, it is unlikely that many theories of any variety have been so widely field-tested. An extended description of Lexicase can be found in Chapter 8.

The main figure associated with DG in Japan is Tokumi Kodama (Kodama 1982). However, very little theoretical DG work has so far been done in Japan.

2.5 Related grammatical formalisms

A number of frameworks bearing similarities to DG have emerged during the last few decades. Three of these merit special attention here, namely Case Grammar, Categorial Grammar, and Head-Driven PSG.

[13]Robinson subsequently abandoned DG in favour of augmented PSG.

2.5.1 Case grammar

Consider the following sentences:

(4) a Punch hit Judy with a club.
b Punch used a club to hit Judy.
c Judy was hit with a club (by Punch).

Although these sentences vary considerably in their surface forms, the semantic relationships they express remain constant. Punch is the agent of the hitting action; Judy is on the receiving end of the hitting action; the club is the instrument of the hitting action.

Case grammar, developed in the late 1960s by Charles Fillmore (Fillmore 1968), formalizes these relationships. The semantic *deep structure* of a sentence is held to consist of two components, a *modality* and a *proposition.* The modality component carries features of tense, mood, aspect, and negation relating to the sentence as a whole. The proposition component records the *deep case* relations in the sentence. Typically these cases are associated with the main verb. In Fillmore's original version of case grammar there were six deep cases: AGENTIVE, INSTRUMENTAL, DATIVE, FACTITIVE, LOCATIVE, and OBJECTIVE. (This number has varied widely between different instantiations of case grammar). The *case frame* for the verb *hit* would include agentive, objective, and instrumental case slots, where each slot can be filled by phrases of the appropriate semantic type.

The similarities between case grammar and DG should be apparent. It is easy to envisage writing a set of case grammar rules in modified Gaifman format, or giving a graphic representation of case structures using arc diagrams. Fillmore himself acknowledges his debt to Tesniére and other dependency grammarians (Fillmore 1977: 60). However, I believe there are good reasons for keeping DG and case grammar clearly separated. DG as I have described it so far, is concerned with surface syntactic structure. Once a dependency structure has been found, one option is to use it as a guide to assign a case structure. However, grammatical relations and case relations are not

necessarily coextensive. In (4a), the OBJECTIVE case is realized by the object grammatical relation (*Judy*). In (4c), the OBJECTIVE case is realized by the subject grammatical relation.

The logical separation of dependency and case is demonstrated in practice by the fact that while some dependency grammarians make extensive use of case in their theories (e.g. Anderson 1971, 1977; Starosta 1988), others make use of alternative semantic frameworks (e.g. Covington 1990b; Hudson 1990). Our concern here is with the construction of *syntactic* structures and not *semantic* structures. The question of which semantic framework is most appropriate when starting from a dependency tree is an interesting one, but it is not the question we are tackling here. Dependency and case — though superficially similar — are logically distinct.

A useful introduction to case grammar is provided by Bruce and Moser (1987). Applications of case grammar in NLP are described in Somers (1987).

It is worth noting in passing that Conceptual Dependency (CD) (Schank 1972; 1975), which is a generalization of case grammar for describing relations holding between events and participants, is also outwith the scope of this thesis. The presence of the word 'dependency' in its title should not be allowed to lead to confusion: DG is concerned primarily with syntactic dependency relations; CD is not.

2.5.2 Categorial grammar

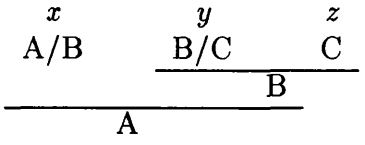
Categorial grammars (CGs) trace their origins from a number of devices developed in the field of logical semantics, specifically Leśniewski's theory of semantic categories (Leśniewski 1929) which brought together insights from Husserl's *Bedeutungskategorien* (Husserl 1900) and Whitehead and Russell's theory of logical types (Whitehead and Russell 1925). Leśniewski's theory was refined by Ajdukiewicz (Ajdukiewicz 1935) who applied the resulting system in the specification of Polish notation languages (parenthesis-free logical languages in which operators/functors are written immediately to the left of their argu-

ments). In a grammar of the sort envisaged by Ajdukiewicz, there are two distinct types of category: **primitive** or **fundamental** categories, which are denoted by unitary symbols (eg. S, N), and **derived** or **operator** categories, which are denoted by complex symbols of the form:

$$\alpha/\beta$$

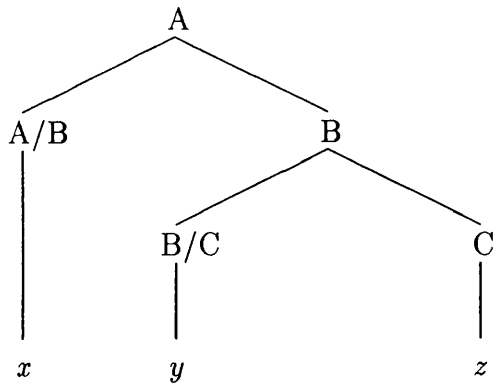
where α and β can be either variety of category, primitive or derived. When complex categories appear within a category, it is customary to place brackets around the embedded categories. A grammar consists of a single rule which states that, given any string of two category symbols α/β and β, replace the string with α. This rule is suggestive of cancelling in fractions, eg. $3/2 \times 2 = 3$.

Consider a language with three items in its alphabet: *x*, *y*, and *z*. *x* has category *A/B*; *b* has category *B/C*; *c* has category *C*. The string *xyz* would be analysed as follows:

(5)


By convention, a line is drawn below two adjacent categories which combine to form a composite category and the resulting category label is written below that line. The similarities to phrase structure should be obvious; in this case we can generate the same string and the same constituent structure (7) with the following set of PS rules:

(6) A → A/B B
B → B/C C
A/B → x
B/C → y
C → z



(7)

Three extensions to Ajdukiewicz's scheme were introduced by Bar-Hillel (Bar-Hillel 1953): (i) assignment of words to more than one category was allowed, (ii) a new kind of complex category — $\alpha\backslash\beta$ — was introduced, and (iii) a new composition rule was introduced to deal with the new kind of category: given a string of any two symbols α and $\alpha\backslash\beta$, replace the string with β.

A CG is *unidirectional* if its complex categories are all either of the form $\alpha\backslash\beta$ or of the form α/β. A grammar with both types of complex category is called a *bidirectional* CG.

In his seminal paper *The mathematics of sentence structure*, Joachim Lambek proposed four different CG rules: application, commutativity, composition, and raising (Lambek 1958). These rules — or minor variants of them — have now become the standard rules of CG.

Application

$$X/Y\ Y \rightarrow X$$
$$Y\ Y\backslash X \rightarrow X$$

These are the rules of combination we have already encountered. If a noun were assigned the category `N`, an intransitive verb would be assigned the category `N\S`. Thus, by the second clause of the application rule, a noun-intransitive verb sequence such as *John snores* would cancel to `S`.

Commutativity

$$(X\backslash Y)/Z \iff X\backslash(Y/Z)$$

Composition

$$X/Y\ Y/Z \rightarrow X/Z$$
$$X\backslash Y\ Y\backslash Z \rightarrow X\backslash Z$$

Raising

$$X \rightarrow Y/(X\backslash Y)$$
$$X \rightarrow Y\backslash(Y/X)$$

The motivation for raising is as follows. Suppose that the pronoun *he* is assigned the category `S/(N\S)` to indicate that it can only occur in subject position, and the pronoun *him* is assigned the category `(S/N)\S` to indicate that it can only occur in object position. The raising rule allows an unmarked noun to assume either of these categories and to appear in either subject or object position.

Interest in CG greatly increased during the 1970s due to the influence of Richard Montague's work in truth-conditional model-theoretic sematics and, in particular, his PTQ grammar (Thomason 1974; see also Dowty et al. 1981). Interest in CG continued to increase throughout the 1980s, due to the influence of David Dowty (Dowty 1982, 1988) Mark Steedman (Ades and Steedman 1982; Steedman 1985, 1986), and others. Many different variants of Lambek's rules are currently in circulation. Steedman's Combinatory Categorial Grammar (CCG) offers one of the most interesting examples. CCG analyses have been offered for particularly difficult non-context-free constructions such as the notorious Dutch cross-serial coordinate structures. Another claim for CCG is that it allows incremental (i.e. strict left-to-right) structure building, and thus it facilitates on-line interpretation (Haddock 1987).

CG and DG are widely held to be notational variants (Lyons 1968: 231). This is understandable, since there is an obvious similarity between a DG rule such as the one shown in (8a) and a CG category such as the one shown in (8b).

(8)

a S(N,*,N)
b N\S/N

However, behind these surface similarities lie the rules of CG. The basic rule of combination in DG is something like CG's rule of functional application. DG has nothing corresponding to the rules of commutativity, composition, or raising.

Most CG parsers adopt a basic shift-reduce strategy. The interest of these parsers lies not in their parsing strategy so much as in the particular form and effects of the combination rules they employ. Later I shall note in passing how some similarities emerge between DG and CG parsing in the context of incremental shift-reduce parsing.

2.5.3 Head-driven phrase structure grammar

Head-driven Phrase Structure Grammar (HPSG) is a theory of syntax and semantics developed by Carl Pollard and Ivan Sag (Pollard and Sag 1988). It differs from standard PSG in the extent to which information is stored in the grammar in relation to head words. For example, part of the lexical entry for *love* is shown below.

(9)

⟨love, V[BSE, SUBCAT ⟨NP, NP⟩]⟩

This states that *love* is a verb which, in its base (infinitival) form subcategorizes for two noun phrase complements. The SUBCAT list is ordered according to obliqueness, with more oblique arguments appearing to the left of less oblique arguments. This is further illustrated by the following example, which shows the by-phrase of passive *loved* on the left of the SUBCAT list.

(10)

⟨loved, V[PAS, SUBCAT ⟨(PP[BY]), NP⟩]⟩

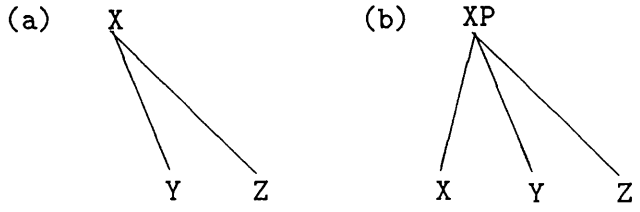


Figure 2.10: syntactic structure in DG (a) and in HPSG (b)

From the point of view of this survey, the identification of arguments by position on an obliqueness list is no more than an implementational detail.

At first sight, the HPSG representation appears to have just the right kind of information stored at the right (i.e. lexical) level to make it into a DG. It is certainly possible to envisage using an HPSG lexicon to produce standard dependency structures. However, in addition to its *lexical rules*, HPSG also makes use of a small number of *phrasal rules*. These effectively add an extra (phrasal) layer of structure above each head word. For every phrase of type X an XP is constructed. X becomes one of the daughters of XP and the features of X are copied to XP. Thus, where DG would build the structure shown in Figure 2.10 (a), HPSG would build the structure shown in Figure 2.10 (b).

The relationship between HPSG and DG is certainly very close. Just how close is a question which I shall not address further here. For information on HPSG parsing see Proudian and Pollard (1985).

2.6 Summary

This chapter has attempted to delineate exactly what is understood by the term 'dependency grammar' as used in this thesis. It has done so first by presenting a detailed formal definition of Gaifman grammar, whose purpose here is to act as a cardinal point to which other versions of DG may be related; and second, by chronicling the rise and spread in linguistics of theoretical approaches which, although they may include additional features, appear to

rest on a foundation which is expressible in terms of a Gaifman grammar. Key figures and schools in the development of DG were identified. To assist in identifying the boundary between that which is included in this study and that which is not, three examples of grammatical theories which lie just outside DG were isolated, and the reasons for their exclusion given.

Chapter 3

Dependency parsers

In the latter part of the last chapter I traced the origins and development of DG in theoretical linguistics. In this chapter I chart the origins and development of DG in computational linguistics.

The designer of a PSG parser has at his or her disposal the whole computational linguistics literature which describes a host of tried and tested techniques and algorithms. However, it does more than this. It defines the space of possibilities for PSG parsing. For example, the designer of a parser must decide whether to build syntactic structure top-down, bottom-up or some combination of the two. Not only is there a serious lack of published descriptions of dependency parsing techniques[1], but there is an even more serious absence of definitions of the problem space. It is sometimes naively assumed that DG parsing and PSG parsing are slight variations on a single theme. This may turn out to be the case but there can be no *a priori* guarantees. For example, it may not make sense to talk about top-down *versus* bottom-up dependency parsing when there are no non-terminal nodes in a tree.[2] One of the main objectives of this thesis is to begin the task of charting the dependency parsing problem space.

[1]The extensive bibliography of *Natural Language Processing in the 1980s* compiled by Gazdar et al. 1987 includes only 9 entries indexed under 'dependency' (excluding non-DG senses of the word).

[2]As we shall see below, a top-down/bottom-up distinction can be made in connection with dependency parsers but it is not exactly the same as the more familiar PSG distinction (cf. Chapter 12).

This chapter begins with an introduction to dependency in computational linguistics (Section 3.1). This is followed by an introduction to *PARS* (Section 3.2), a language for the description of dependency parsing algorithms, which I shall use for the sake of clarity in the survey of existing dependency parsers which follows this chapter.

3.1 Dependency in computational linguistics

Although computational DG is lacking in theoretical underpinnings, a number of systems have been developed from the late 1950s onwards. These can be roughly divided into machine translation systems, speech understanding systems, other applications, implementations of theories, and exploratory systems. The next eight chapters present some of these systems in more detail.

3.1.1 Machine translation systems

Dependency-based machine translation (MT) research has taken place in two periods: the first half of the 1960s and the second half of the 1980s.

The early 1960s

In the early 1960s, there were two major dependency-based MT projects. The first of these was based at the Moscow Academy of Sciences. Amongst the scholars associated with the project were Sergei Fitialov, O.S. Kulagina and Igor Mel'čuk. Very few — if any — documents describing the project in detail are available in English. However, an annotated bibliography on dependency theory released by Hays in March 1965 contains English abstracts of a number of the project papers (Hays 1965). Since these papers are not discussed elsewhere in the English literature and since Hays' bibliography is not in wide circulation, those most immediately relevant to our present concerns are reproduced below.

The coding of words for an algorithm for syntactic analysis
(Martem'yanov 1961)
Word classes ADr, ADl, AG, PGr, and PGl are defined, where A=active, P=passive, D=depend, G=govern, r=right, l=left. An active governor sweeps up passive dependents. A parsing routine is discussed in part, including the effect of English inflections on word class.

An algorithm for syntactical analysis of language texts — general principles and some results
(Mel'čuk 1962)
A dependency parser is outlined. The units of syntactic analysis are 'content combinations', i.e. syntagmas (governor and dependent), phraseological combinations, etc., given in the form of configurations, each giving a pair of objects to be sought, a search rule, conditions, actions, etc. These are listed in a syntactic dictionary. The algorithm that uses this list consists of 67 standard (Kulagina) operators. The Russian configuration list has 263 lines. About 250 auxiliary operators are used. A flowchart and configuration list are given.

Obtaining all admissible variants in syntactic analysis of text by means of computer
(Slutsker 1963)
Assume a grammar that specifies what pairs of words can be connected as governor and dependent. To find all projective parses of a sentence, first set up a square matrix with $w_{ij} = 1$ if the grammar allows word i to depend on word j. A parse of the sentence can be specified by a matrix with a single non-zero element in each row, chosen among those with $w_{ij} = 1$. Projectivity can be interpreted in terms of incompatibilities in the matrix; all elements incompatible with unit elements unique in the rows can be erased. Then, by a backward procedure, all parses can be found.

It is unfortunate that more information is not available on the Moscow MT project. However, on the basis of these brief abstracts it is possible to infer that a significant amount of effort was directed towards developing dependency-based NLP systems. (The abstracts tell us, for example, that at least three scholars were involved in the development of at least three parsing algorithms).

The second major dependency-based MT project was sponsored by the RAND corporation and led by David Hays. The RAND project aimed to build a Russian-English MT system. It appears that Hays had no contact with the work of Tesnière. Instead he learned about DG from the Soviet scholars. It seems that there was a surprising amount of communication between the two research groups (especially considering the prevailing Cold War climate and the defence significance of Russian-English MT). The RAND DG work is summarized in Chapter 4 of this thesis.

A third strand of dependency-based MT work was begun by Petr Sgall's group in Prague in the early 1960s (Sgall 1963). No information on this work is available at the time of writing.

The mid 1980s

In the mid 1980s three large dependency-based MT projects were undertaken. The first of these is the European Community EUROTRA project (Johnson et al. 1985; Arnold 1986; Arnold and des Tombe 1987) which has led more recently to an offshoot dependency-based project called MiMo based at the Universities of Essex and Utrecht (Arnold and Sadler forthcoming). The second, the Dutch Distributed Language Translation (DLT) project, is discussed in Chapter 7 of this thesis. Sgall's group in Prague has recently been developing an MT system based on the model of Functional Generative Description (Kirschner 1984; Hajič 1987; Sgall and Panevová 1987; Hajičová 1988). It is not clear whether this is a continuation of the work begun in the 1960s, or whether it represents a completely new venture.

3.1.2 Speech understanding systems

In at least two projects, dependency parsers have been used to process lattices output by speech recognition systems. The claimed advantages of DG are first, that its rules and structures are word-based and can readily be associated with the basic units of recognition in lattices, namely word hypotheses; and second,

that DG is well-suited to combining top-down and bottom-up constraints in a way which is particularly useful for processing lattices.

The first speech understander to make substantial use of dependency is the Italian CSELT system (late 1980s), which is a speech interface to a database. The CSELT parser is described in Chapter 11.

The second dependency-based speech understander was developed in Japan at NTT Tokyo and the University of Yamagata. It is described in Matsunaga and Kohda (1988).

The speech understanding system developed for the SPICOS project by Niedermair and his colleagues at Siemens in Munich (Niedermair 1986) is sometimes mentioned in discussions of DG. However, their system is a hybrid of PSG and basic valency theory. A first-phase augmented context free phrase structure parser identifies and builds the major phrases in a sentence. A second-phase parser establishes binary relations between the major phrases on the basis of semantic caseframe entries. This is an interesting approach, motivated by the particular problems encountered in parsing speech. However, it would be misleading to describe it as a dependency parser.[3]

3.1.3 Other applications

At least one major NLP project has investigated the use of dependency in a practical application other than MT or speech understanding. This is the Finnish Kielikone project whose aim is to produce a general-purpose natural language interface which — in theory at least — can sit on top of any database with minimal customization. The project has been running since 1982. The Kielikone parser is described in Chapter 6.

[3]This view has recently been confirmed by Gerhard Niedermair (personal communication).

3.1.4 Implementations of theories

So far we have considered only DG-based NLP systems which were designed with some particular application in mind, such as MT, speech understanding, or database access. However, a number of systems have been built in order to test the coverage and coherence of particular linguistic theories. The two theories which have most obviously spawned this kind of activity are Lexicase (Starosta 1988) and Word Grammar (Hudson 1984, 1990a). Their implementation has taken place only in recent years. The fact that more dependency-based theories have not been implemented reflects the fact that there has been a shortage of well-developed theories to implement. At least two parsers based on Lexicase have been produced so far (Starosta and Nomura 1986; Lindsey 1987). These are described in Chapter 8. Several parsers based on Word Grammar have been implemented (e.g. Fraser 1989a; Hudson 1989c). These are described in Chapter 9.

Some of the work done by Sgall's group in Prague is directed towards implementing the theory of Functional Generative Description (e.g. Petkevič 1988), although this seems to be less emphasized than the design of specific applications.

3.1.5 Exploratory systems

It would be misleading to suggest that all work in dependency parsing has been carried out with specific applications or theoretical linguistic objectives in mind. Some of the most interesting and useful results have emanated from exploratory research directed towards investigating the computational properties of DGs and trying out various novel parsing algorithms.

Early in the 1960s, a DG research group was formed at the EURATOM CETIS Research Centre in Ispra, Italy.[4] Other research was carried out by a group funded by EURATOM and under the leadership of Lydia Hirschberg

[4]EURATOM = European Atomic Energy Community. CETIS = Centre Européen pour le Traitement de l'Information Scientifique.

at the University of Brussels. Although the work these groups carried out is widely referenced, most of it is described in EURATOM internal reports and is otherwise unavailable. The following abstracts appear in Hays' annotated bibliography.[5]

> **Automatic analysis**
> (Lecerf 1960)
> The 'conflict' program tests each item against the adjoining, already constructed phrase and either subsumes it as an additional dependent or makes it the governor of a new, extended phrase. The result is a chameleon, looking like both a phrase structure diagram and a dependency diagram.

> **Conditional relaxation of the Projectivity Hypothesis**
> (Hirschberg 1961)
> When parsing is blocked and a subtree exists headed by a unit that demands a governor, remove that subtree and continue. When a tree for a sentence is otherwise complete, look for the governor in the subtree headed by the nearest preceding node. Many examples are given. There are also fixed non-projective combinations in many languages. An annex classifies French dependency types by value. The highest value obtains when governor and dependent require one another; the lowest, when neither calls specifically for the other.

For at least a decade and a half, Jürgen Kunze's group in East Berlin has been developing a version of DG for use in computer applications. This work could be expected to be of considerable significance in dependency parsing. Unfortunately, very little of Kunze's material has been available for inspection at the time of writing.

Since the early 1970s Peter Hellwig has been developing his PLAIN system, chiefly at the University of Heidelberg. PLAIN is a suite of programs centred around a dependency parser. While Hellwig is actively involved in a number of NLP projects to develop applications, the PLAIN system seems to be primarily a research environment. The PLAIN system is described in Chapter 5.

[5] Hays himself spent 1962-63 at the EURATOM CETIS Centre, Ispra, Italy on leave from RAND.

During the last few years, Michael Covington at the University of Georgia has developed a number of simple dependency parsers in order to explore the parsing of free word order languages. Covington's most recent parser is described in Chapter 10.

A simple dependency parser has been designed and implemented by Bengt Sigurd at the University of Lund. This work was inspired by Sigurd's reading of Schubert (1987).

Very recently a group at IBM's Tokyo Research Laboratory has begun to experiment with dependency-based NLP systems (Maruyama 1990; Nagao 1990).

I have presented a very brief historical overview of the field of dependency-based NLP. This is summarised in Figure 3.1. Projects identified by heavy lines are discussed in detail in Chapters 4–11. Notice how the early enthusiasm and associated research effort — much of it associated with MT — dwindled to almost nothing in the late 1960s and throughout the 1970s. It is interesting to see how interest has picked up throughout the 1980s and, at the start of the 1990s, the field is blossoming once more.

Chapters 4–11 present overviews and critiques of twelve parsers. I present a summary table for each algorithm noting the following features: search origin (top-down, bottom-up, etc), search manner (breadth-first, depth-first, etc), search order (left to right, right to left, etc), number of passes (single pass, multiple passes, etc), search focus (what is being searched for?), and ambiguity management (how are choice points and multiple analyses handled?). Verbal descriptions of the algorithms are presented but these can not always be as perspicuous as might be desired. Consequently, the informal verbal descriptions are accompanied by slightly more formal descriptions. In order to facilitate understanding and comparison of the parsers it is useful to abstract away from the many different notations used, and to represent the parsing algorithms in a clear and theory-neutral fashion. It would be an enormous

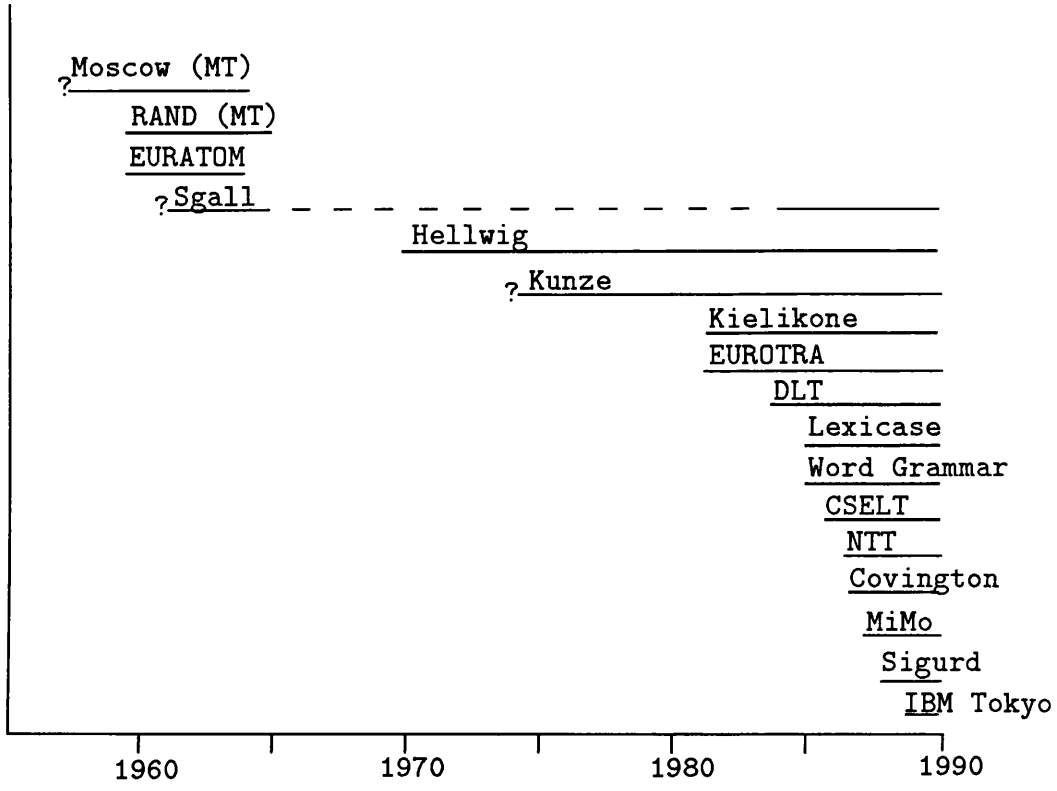


Figure 3.1: dependency-based NLP projects

task to do this thoroughly. First of all, it could involve the design of a whole new representation language whose syntax and semantics would have to be defined. Second, it would involve representing the knowledge pertaining to each parser in the kind of detail which would make the task comparable to re-implementing the algorithms. The solution adopted here is a compromise. A representation called PARS is introduced in the next section. It is intuitively simple but lacking in formal rigour. The primary purpose of PARS is to achieve expository clarity in descriptions of parsing algorithms. I make no stronger claims for the representation.

3.2 PARS: Parsing Algorithm Representation Scheme

In this section a simple quasi-formal language (PARS) for describing dependency parsing algorithms is outlined. Its purpose is exposition rather than implementation so it is defined rather less rigorously than would be required in a more formal specification. There is a tradition in computer science of using languages of this type (sometimes known as *pseudo-Pascal*) to describe algorithms (e.g. Goldschlager and Lister 1982). PARS is unusual in being designed specifically to serve as a general-purpose representation scheme for dependency parsing algorithms. I shall use PARS to describe many of the dependency parsing algorithms described in the following chapters.

3.2.1 Data structures

Constants

Integers, and lower case identifiers are allowed. Two list-related constants are recognized. '**0**' is the 'begin-list' marker. '*e*' is the 'end-list' marker.

Variables

Variables can be distinguished from other data structures in PARS by the fact that they all begin with an upper case character. All variables are global unless otherwise indicated.

By convention, the variable C is used to identify the current word in the input list of words by means of its sequential position in the list. Because of PARS's expository function, this variable is used fairly loosely. Sometimes it is used as a normal variable, sometimes as a pointer, sometimes it refers to the thing pointed to. The context ought to make the interpretation clear in each case.

Other naming conventions include List (a global list), Stack (a global stack), and Top (the top element on the stack).

As we shall see below, values are assigned to variables by means of the := (assigns) operator.

Variables can be used as pointers. When X is a pointer, X↑ is the element to which it points.

Stacks and Lists

A stack is a last-in-first-out data structure. The default name for a stack is Stack. The action

pop(Stack)

discards the topmost item on the stack. The action

push(Element)

pushes Element (any variable or constant) onto Stack. It is possible to push elements onto stacks other than the default stack by means of the action

push(Stack1, Element)

which pushes Element onto Stack1 (some stack).

The action

empty(Stack)

returns 'true' if Stack is empty, otherwise it returns 'false'. The action

top(Stack)

returns the top element of Stack without popping it.

A list is an ordered sequence of elements. The begin marker is '0'. The end marker is '*e*'. Elements in a list are addressed by pointers. If C is a pointer to a list element, then C–1 is the previous element and C+1 is the next element. An element can be added to the tail of a list by means of the action

append(Element) or append(List1,Element)

and an element can be removed from the list by means of the action

remove(Element) or remove(List1,Element).

The first and last elements of a list are returned by the following actions:

first(List1)

and last(List1).

The length of a list can be found by means of the action

length(List1)

3.2.2 Expressions

The basic components of PARS descriptions are expressions. Expressions can either be simple, consisting of one or more actions, or structured condition-action sequences, as shown below:

(11)

IF *condition(s)*
THEN *expression(s)*
(**ELSE** *expression(s)*)

In addition, expressions may be labelled, as follows:

(12)

N: *expression*

where 'N' is an integer.

Expressions end with a full stop.

Conditions

Conditions can be of several different varieties. Each variety is associated with a different operator. The general purpose operators are summarized in the table below.

Operator	Name
=	equals
→	depends
⊔	unifies

Equality The = (equals) operator is used to test two items for identity. The test succeeds it the items are identical.

Dependency The $\rightarrow$ (depends) operator is used to test for dependency. The test succeeds if the element on the RHS of the operator already depends on the element on the LHS of the operator or if it can be made to depend on the LHS element (i.e. there is nothing in the grammar or the sentence to prevent a dependency relation from being established). The detailed articulation of this operator will vary from system to system.

Unification The $\sqcup$ (unifies) operator is used to test whether or not two feature structures unify. The test succeeds if the structures unify. As well as producing a truth value, a successful test also results in the unification of the feature structures tested as a side effect.

Other As was noted above, the **empty(Stack)** action returns a truth value and can be used as a condition in expressions.

The condition **saturated(C)** succeeds if all of the valency requirements of some word C are satisfied.

Conjunctive and disjunctive conditions Conditions may be conjoined using the & (and) operator. For example:

(*condition1* & *condition2*)

Disjunctions of conditions are possible using the 'V' (or) operator. For example:

(*condition1* V *condition2*)

Actions

Assignment Values are assigned to variables using the := operator. Thus

C:=1

assigns the value 1 to C. If C equals, say, 5, then it is possible to reassign C thus

C:=6

or thus

C:=C+1

(the result is the same in both cases).

Record The record(X) action makes a record of X. For example,

record(C → C+1)

makes a record of the fact that a dependency has been established in which C is the head of the next word in the global queue.

Goto The goto(Label) action shifts control to the expression identified by Label. The state before a goto action is not stacked. It is not possible to return to a prior state once a goto action has been executed. Expressions are usually identified by integers. For example,

goto(3)

Length The length(List) action returns an integer corresponding to the number of elements in List, excluding the end-of-list marker.

Succeed and fail succeed signals that a parse has succeeded. fail signals that parsing has failed. Both actions terminate the parse immediately.

Others As noted above, other actions include the stack-related pop and push, and the list-related append and remove.

3.3 Summary

In this chapter I have charted the rise of 'applied' DG, i.e. DG in service of NLP. I have shown how an increasing number of NLP systems are being based on DG in MT, speech understanding, and database access systems. Separate strands of research are devoted to building NLP systems whose object is to explore novel parsing algorithms and to implement linguistic theories. Lack of published material (or lack of material published in a language accessible to me) renders it impossible to include here a detailed examination of every system named in the survey. The following eight chapters describe those parsers for which most information is currently available. At least one representative of each of the categories mentioned above is included in this collection. Where possible and helpful, parsing algorithms are described in the special-purpose description language, PARS.

The following chapters constitute the most thorough examination of the practice of dependency parsing yet assembled. Chapter 12 builds on this material with a view to outlining some elements of a general taxonomy of dependency parsing algorithms.

Chapter 4

The RAND parsers

4.1 Overview

In this chapter I present the earliest dependency parsers described in this survey. The parsers were produced in the early 1960s by researchers at the RAND Corporation, Santa Monica, USA and reported, for the most part, by David Hays. Most of the natural language work at RAND was centered on the development of a Russian-English MT system, of which a parser was considered to be a vital part. The choice of DG as the basis of the system could be regarded as natural considering the difficulties involved in writing PSGs for variable word order languages like Russian — especially as the RAND work preceded developments in PSG for handling variable word order such as scrambling transformations (Ross 1967; Saito 1989) or the ID/LP formalism (cf. Gazdar et al. 1985: 44–50). However, in 1961 — when RAND was just one of many groups involved in building Russian-English MT systems — DG was far from being the 'natural' choice. Hays claimed that "Phrase structure theories underlie all MT systems being developed in the United States, except that of the RAND Corporation" (Hays 1961b: 258). As a leading figure in MT in the United States who was soon to become president of the Association for Machine Translation and Computational Linguistics, Hays would almost certainly have known if there had been any other dependency systems in existence. For an overview of the NLP work carried out at RAND in the early

1960s, see Hays (1961c).

It is hard to over-emphasize the importance of the RAND work in the development of dependency parsing. It was probably the first major project in computational linguistics in the Western world to be based on DG. Although Tesnière's *Éléments de Syntaxe Structurale* was published shortly before the RAND MT project got under way, it is not referenced in any of the available publications by Hays or his colleagues. Instead, the RAND work seems to draw on an older Russian literature. In fact, Hays reports that several Soviet MT projects made use of the notions of dependency. Leading figures in these projects are named as Kulagina, Moloshnaya, Padučeva, Revzin, Shelimova, Shumilina and Volotskaya. Unfortunately, nothing has been found describing their DG work except Hays' abstracts presented on page 62 above. Their work in other areas of formal linguistics is described in Papp (1966) and Kiefer (1968). As the first widely publicized NLP system based on dependency, the RAND system set an agenda for future systems to follow. Almost all authors of the other systems described in this thesis acknowledge their debt to Hays and his colleagues.

It must be remembered that computational linguistics was rather different thirty years ago from its present-day condition. Firstly, there were hardware and software limitations which impaired prototyping and which, inevitably, coloured the way that researchers viewed the problems to be modelled. We shall see in this chapter some suggestions which seem rather old-fashioned to modern eyes. Secondly, many techniques of linguistic description which are nowadays taken for granted, were in 1960 still in their infancy or even waiting to be invented. For example, the RAND systems would almost certainly have looked different if their designers had been able to make use of complex feature unification. Thirdly, the prevailing views on what constituted difficult problems and what constituted easy problems were markedly different from present day views. These were days of great optimism in MT. Hays wrote in

1961:

> Machine translation is no doubt the easiest form of automatic language-data processing... In 10 years we will find that MT is too routine to be interesting to ourselves or to others. (Hays 1961c: 25)

Of course, events proved him wrong. The US National Academy of Sciences produced a damning report on MT in 1966 which resulted in all US government funds to MT projects drying up, and with them the dream of constructing fully functional MT systems. This precipitated the demise of the RAND MT project and the virtual disappearance of DG from Western computational linguistics until the emergence of a new wave of DG research in the 1980s.

In this chapter I present two parsing algorithms. One of these was implemented in the RAND MT system and could loosely be described as a 'bottom-up' algorithm. The other is described by Hays in abstract terms and it is not clear whether it was ever implemented. It could loosely be described as a 'top-down' algorithm. A third algorithm is described very briefly in Hays (1966b). Unfortunately, insufficient detail is given to reconstruct the algorithm.

4.2 The bottom-up algorithm

The bottom-up algorithm was embodied in the RAND SSD ('Sentence Structure Determination') program. The principle references are Hays and Ziehe and Hays (1961a).

4.2.1 Basic principles

There may, in fact, have been several distinct versions of the parser described here. Hays points to the fact that work centred around two 'basic principles' which could be 'preserved through a variety of technical variations'.

Basic principle 1: separate word-order and agreement rules

The first basic principle was that word-order rules should be isolated from agreement rules.[1] This principle led to the development of two sub-programs. The first program selected pairs of words which could serve as candidates to enter into a dependency relationship on the basis of their relative positions. The second sub-program tested to see whether a dependency relation was possible on the basis of the grammatical features and dependency requirements of each word. The sub-programs could thus be thought of as working alternately; the first program selected a pair for the second program to link or reject. If the linking program succeeded then the pair-selection program would try to find a new pair of candidates for linking. If the linking program failed then the pair-selection program would have to find an alternative pair to be linked.

Basic principle 2: adjacency

The second basic principle stated that 'two occurrences can be connected only if every intervening occurrence depends, directly or indirectly, on one or the other of them'. In other words, this was an explicit adjacency constraint. This, in turn, ensured that the class of languages recognized was exactly the class of context-free languages.

4.2.2 The parsing algorithm

The parsing algorithm iterates through the pair-selection/linking cycle until there are no more pairs left to select.

Pair selection

The pair selection procedure effectively embodies the control strategy of the parser. It works by attempting to link any two words which are immediate

[1] Hays (1961a: 368) states that "this principle has been invented, lost, and re-invented several times."

neighbours in the input string. Search for immediately adjacent pairs proceeds from left-to-right. An attempt is made to link the current word with its rightside immediate neighbour. If a dependency can be established between the two words, the dependent drops out of sight, thus creating a new pair of immediately adjacent elements to be tested. The word which is the head of the newly created pair becomes the current word. If a dependency is not established, the next word in the string becomes the current word. Leftside neighbours are only checked after a change of current word resulting from a failure to establish any dependency links.

The algorithm can be described more formally in PARS as follows.

INITIALIZATION: read input words into a list;
C:=1.

1. **IF** C+1=e

   THEN halt

   ELSE IF C $\rightarrow$ C+1

   THEN record(C $\rightarrow$ C+1),
   remove(C+1),
   goto(1)

   ELSE IF C+1 $\rightarrow$ C

   THEN record(C+1 $\rightarrow$ C),
   remove(C),
   goto(1)

   ELSE C:=C+1,
   goto(2).

2. **IF** C=e

   THEN halt

   ELSE IF C $\rightarrow$ C+1

   THEN record(C $\rightarrow$ C+1),
   remove(C+1),
   goto(2)

   ELSE IF C+1 $\rightarrow$ C

   THEN record(C+1 $\rightarrow$ C),
   remove(C),
   C:=C+1,
   goto(3)

   ELSE C:=C+1,
   goto(2).

3. **IF** C=1

   THEN goto(1)

   ELSE IF C–1 $\rightarrow$ C

   THEN record(C–1 $\rightarrow$ C),
   remove(C),
   C:=C–1,
   goto(2)

ELSE IF C → C–1
THEN record(C → C–1),
remove(C–1),
goto(3).

Algorithm 4.1: Hays' bottom-up parser

The parser succeeds in producing an analysis for the whole sentence if exactly one word remains visible in the input list at the end of the parse. This implies that all the other words have been successfully linked into the structure and so have disappeared from view.

The parser reported by Hays produces only a single analysis for an ambiguous sentence. This was a limitation imposed by the then existing technology.

> It has to be assumed by most designers that the cost of a search for all possible structures is too great to be borne in practice; heuristic devices of various types therefore appear in most SSD programs. (Hays 1961a: 370)

The parser favours closer attachments over more distant ones. Hays suggested three kinds of heuristic which could be used to increase the likelihood of the parser getting the attachments right first time. (Apparently this was vital: there are no references to the possibility of backing up after wrong choices.)

Word-centred ordering Hays' first suggestion was to specify for certain words a partial ordering for the establishing of their dependency relations. For example, in one trial version of the RAND SSD system a preposition could not be linked to its head until its object had been attached to it.

Dependency-centred ordering Dependency relations could be labelled according to grammatical type (such as subject). A partial ordering could then be established amongst types (for example, find subjects before objects).

Assign 'urgency' scores Dependency relations could be assigned 'urgency' scores. Whenever more than one possible link existed, the one with the highest urgency score was allowed to 'win'. This was a simple weighting system. Hays only suggests local scoring of alternative analyses. It would be interesting to investigate the use of global scoring techniques to choose between alternative analyses. Of course, both approaches presuppose that some reliable weights are available, for example, from a hand-analyzed corpus (see Chapter 7 for more on this approach to dependency parsing). Hays does not report the results of any trials which made use of 'urgency' scores and it seems unlikely that his suggestion was implemented.

Linking

The parsing algorithm presented above shows the order in which word-pairs should be examined to check the possibility of establishing a dependency relation between them. In a modern-day system this would constitute most of the work of the parser. The test for dependency would simply involve an attempt to unify two complex feature structures, one associated with each word to be tested. If the test succeeds then unification has already built the new composite structure, otherwise a simple failure is returned. However, in the early 1960s no such luxuries were available and so-called 'agreement tests' constituted a major part of the parsing problem. At least one of Hays' papers (Hays 1966a) is entirely devoted to this subject. If they were used, the heuristics mentioned above would be implemented in the agreement testing mechanism. The details of the various kinds of agreement testing are mostly of little relevance to modern readers. However, two of the strategies still hold some interest.

Table look-up Imagine a feature-based grammar including a large feature inventory covering all of the various distinctions possible in a grammar. Now imagine converting every possible feature permutation into a distinct atomic symbol. This is effectively what was done in the RAND SSD system. Each

word form was assigned to one of these symbols (or a disjunction of these symbols). For convenience the symbols used were integers. Assume that there were n distinct integer symbols. An $n \times n$ array was set up. In order to find out whether a dependency could be established between a word form of type i and another of type j it was necessary to look in the (i, j)-th cell of the matrix. This would indicate whether it was possible to link the words and, if so, what kind of dependency relation was involved and which word was the head. In the RAND system a 4000×4000 cell array was used and it was projected that a 50000×50000 array would eventually be required! It is little wonder that agreement testing came to be viewed as such a significant component of the parsing problem.

Bit encoding One of Hays' suggestions to improve the efficiency of agreement testing was a modification of the categorial grammar system that Lambek had recently developed (Lambek 1958). Hays' suggestion was to replace the atomic symbols in a category symbol (usually N and S, e.g. S/N) with complex symbols. He writes:

> In Russian, nouns and adjectives agree in number, gender and case; there are six cases, and the following gender-number categories: masculine singular, feminine singular, neuter singular, and plural. Let each bit-position of a 24-digit binary number correspond to a case-number-gender category, and use the appropriate number as a component of the grammar-code symbol of adjective or noun. Agreement is tested by taking the 'intersection'... If the intersection is zero, the occurrences do not agree. This method is faster in operation and requires no stored agreement tables; it is almost certain to be the method of future operational systems. (Hays 1961a: 373–4).

There is no evidence that this approach was ever tried at RAND. A recent parsing system which includes a similar strategy using a UCG is described in Andry and Thornton (1991) and Andry et al. (1992).

4.3 The top-down algorithm

In this section we examine Hays' other dependency parsing algorithm. It is not clear whether it was ever implemented at the RAND Corporation. Hays describes it in an introductory textbook on computational linguistics (Hays 1967) so it is possible that it was invented for purely pedagogical purposes.

4.3.1 The parsing algorithm

This parser is in the minority amongst the dependency parsers described in this survey in that it embodies a top-down control strategy. Hays' exposition does not describe the rule system employed by the parser so I shall assume that dependency rules are expressed in Gaifman format. Rules may thus take the following forms:

(13)

a $X_i(X_{j_1}, X_{j_2}, ..., *, ..., X_{j_n})$
b $X_i(*)$
c $*(X_i)$

where (13a) shows the case where X_i has dependents X_{j_1}–X_{j_n}. (13b) is the case where X_i can appear in a sentence without dependents. (13c) notates the case where X_i can appear in a sentence without depending on any other word, i.e. it is the sentence root.

The parsing algorithm begins by scanning the sentence for a word which can serve as the sentence root, i.e. for which there is an entry of type (13c) in the grammar. Having found the sentence root, the algorithm makes it the root of a dependency tree. Next, the grammar is searched for a rule of type (13a) listing possible dependents for the root, or a rule of type (13b) showing that the root can occur without dependents. For example, suppose that the sentence root is R; the grammar is searched for a rule of type $R(...)$. If a rule is found, it is matched against the words of the sentence. For example, if the rule $R(Q, *, S)$ is found, checks are made to see if the pattern '$Q...R...S$' is

present in the input sentence. If there is a match then the fact that these dependents have been found is recorded in the dependency tree. If there is no match then an alternative rule specifying dependents for the root is searched for in the grammar. The same is done for every word in the input string when it becomes a leaf in the dependency tree. If a rule of type (13b) matches any word X then no more rules of type $X(...)$ are searched for. A sentence has been successfully parsed if all leaves in the dependency tree have been matched against rules of type (13b) and no words remain in the input string which are not linked in the dependency tree.

I shall say that a word X for which a rule of type $X(...)$ is found and matched, has been *expanded.* If the dependency tree is represented as a nested list, then expansion replaces one symbol with more than one symbol. For example, consider the following sentence:

(14)

Simpson eats haggis

Assume that the sentence is pre-processed with a word class recognizer:

(15)

[N: simpson] [V: eats] [N: haggis]

If the grammar contains a rule of the form $*(V)$, the dependency tree will initially look like this:

(16)

([V: eats])

If the grammar contains a rule of the form $V(N, *, N)$, then the dependency tree can be expanded to look like this:

(17)

(([N: simpson]) [V: eats] ([N: haggis]))

Thus, it should be clear that successful expansion operations increase the size of the tree. Note, however, that the number of nodes in the final tree (17) is no greater than the number of symbols in the input string. In this respect,

top-down dependency parsing differs crucially from top-down PSG parsing: *in top-down dependency parsing an expansion can not add a symbol which does not appear in the input string.* In top-down PSG parsing, of course, extra non-terminal symbols can be inserted by expansion operations. This leads to the possibility in a top-down PSG parser of an infinite succession of non-terminal symbol insertions, as in the case of left recursion. The dependency parsing algorithm described here is capable of recognizing exactly the context free languages (recall Gaifman's result) but unlike a top-down CFPSG parsing algorithm which has not been heuristically constrained, it can never enter infinite loops, given an arbitrary grammar. Thus, it must be regarded as being more robust than a top-down CFPSG parsing algorithm which is always at the mercy of the grammar with which it works. If the CFPSG contains any left recursive rules then parser can expect, sooner or later, to blunder into an infinite loop.

The order in which symbols are expanded is not crucial to Hays' algorithm, although it may be important in some applications. If the leftmost available leaf were always to be expanded this would lead to a left-to-right depth-first search. If the rightmost available leaf were always to be expanded it would lead to a right-to-left depth-first search. If all nodes at distance d from the root were expanded before any nodes at distance $d + 1$ were expanded, a breadth-first search would be implemented. This could also be set up to progress left-to-right, right-to-left or middle-out, all at level d before moving on to level $d + 1$. However, these labels describe the ways in which the branches are added to dependency trees rather than the order in which words in the sentence are built into the trees. For example, a left-to-right depth-first parser would add the words of sentence (18) into the tree in the order: *like, giants, jolly, green, corn, golden.*

(18)

Jolly green giants like golden corn

Table 4.1: main features of Hays' bottom-up dependency parser

Search origin	bottom-up
Search manner	depth-first
Search order	left to right
Number of passes	one
Search focus	pair-based
Ambiguity management	first parse only (heuristics guide choices)

Hays top-down parser is intuitively simple but since it is best described formally in terms of recursive procedure calls, a PARS description of the algorithm is not particularly illuminating. The subject of top-down dependency parsing is addressed in Chapter 12, where a top-down algorithm is presented in detail.

4.4 Summary

Hays' first parsing algorithm processes sentences from left-to-right. It is bottom-up, in the sense that it starts building structure from the words in the sentence rather than from the rules in the grammar. Heads do not search for dependents; neither do dependents search for heads. Instead, the parser searches for potential head-dependent pairs and an agreement matrix ('belonging' to neither word) indicates whether the *potential* dependency can become an *actual* dependency. There is never an instance of one member of the pair searching for the other member. The parser produces at most a single analysis for each input sentence by means of depth-first search.

The main features of Hays' first parser are summarized in Table 4.1 (the exact meaning of entries in summary tables will be discussed in Chapter 12).

Hay's second parsing algorithm processes sentences from heads to dependents. It is top-down in the sense that it builds structure from the rules in the grammar rather than from the words in the sentence. Hays leaves many of the details of his algorithm unspecified or underspecified. I have attempted to show how different search strategies offer variations on the order in which

Table 4.2: main features of Hays' top-down dependency parser

Search origin	top-down
Search manner	unspecified
Search order	unspecified
Number of passes	one
Search focus	heads seek dependents
Ambiguity management	unspecified

a dependency tree is constructed although the resulting tree does not depend on the order in which branches are added. No strategy for handling ambiguity is offered.

The main (known) features of Hays' second parser are summarized in Table 4.2.

Chapter 5

Hellwig's PLAIN system

5.1 Overview

The PLAIN system ('Programs for Language Analysis and INference') is a suite of NLP computer programs developed by Peter Hellwig at the University of Heidelberg. The system originated in work Hellwig did in the early 1970s towards his dissertation (Hellwig 1974). Since then he has continued to develop his system. Although the PLAIN system has been implemented in several different locations around the world (e.g. Cambridge, Hawaii, Heidelberg, Kiel, Paris, Pisa, Surrey, Sussex, Zurich) and customized for at least three different languages (English, French and German), Hellwig remains the only author on the PLAIN bibliography (a copy of which is included in Hellwig 1985: 79).

Basically, the PLAIN system is a parser. I shall not describe any of its incidental capabilities here. Instead, I shall detail the form and content of the grammar that PLAIN uses. All linguistic knowledge is written in a single feature-based representation called 'Dependency Representation Language' (DRL). I shall examine the way in which the parser uses unification to build structures, including discontinuous constituents. I shall also show how a chart can be used to increase the efficiency of the parser.

5.2 Dependency Representation Language

Hellwig's primary motivation for basing his parser on dependency is his belief that DG provides a framework within which "functional, lexical, morphological and positional features can be processed smoothly in parallel" (Hellwig 1986: 198). This can be done within a single representation language and a single structure. Hellwig contrasts this with, for example, LFG (Kaplan and Bresnan 1982) which builds a c-structure to represent the syntactic constituent structure of a sentence and a distinct f-structure to represent the functional dependency relationships between functors and arguments. He describes his dependency system in the following way:

> The salient point of this formalism is that the functional, the lexematic and the morphosyntactic properties coincide in every term, as they do in the elements of natural language. To put it in the terminology of LFG: f-structure and c-structure are totally synchronized. Since this cannot be achieved in a phrase structure representation, it is often assumed that there is a fundamental divergence between form and function in natural language. (Hellwig 1986: 196).

In effect, Hellwig is offering an existence proof that form and function do coincide in natural language, at least to the extent that they have been modelled in the PLAIN system.

A secondary argument Hellwig offers for using DG is that it deals with discontinuous constituents rather more elegantly than PSG. There are, after all, no 'constituents' to be 'discontinuous' in DG. As we shall see, this claim takes us beyond the power of Gaifman grammars.

5.2.1 The form of DRL expressions

All linguistic information is represented in a unified framework, DRL. Hellwig describes it in the following terms:

> Grammar formalisms and computer languages are usually developed independently. DRL is both at the same time. In the same

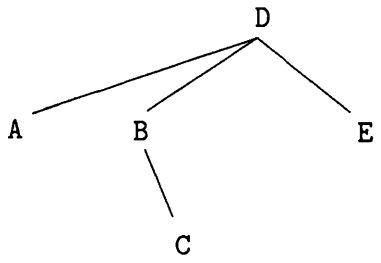


Figure 5.1: stemma showing a simple dependency structure

> spirit as Prolog is tailor-made for the purposes of logic, DRL has been particularly adapted to represent linguistic structures. Whereas the interpreter for Prolog includes a theorem prover, the interpreter for DRL is linked with a parser. (Hellwig 1986: 195)

The parser is described in the next section. Here, I pursue the question of linguistic representation. A DRL structure consists of a bracketed expression, where the bracketing represents a tree with nodes and directed arcs. Arcs are directed from the node represented by an outer bracketing to the nodes represented by each bracketing it contains. Each node is a lexical item. Thus, an expression representing the stemma shown in Figure 5.1 has the form shown in (19).

(19)

```
(D (A) (B (C)) (E))
```

In a DRL expression, the nodes A–E (called 'terms') correspond to single words but they are not expressed by atomic symbols. Rather, they consist of collections of features in the form of attribute-value pairs. Three types of attributes are grouped together in each DRL term, namely a role, a lexeme, and a complex morphosyntactic category.

Sentence (20) would be represented by the DRL expression shown in (21).

(20)

The cat likes fish

(21)

```
(ILLOCUTION: assertion: clse typ<1>
   (PREDICATE: like: verb fin<1> num<1> per<3>
      (SUBJECT: cat: noun num<1> per<3>
         (DETERMINER: the: dete))
      (OBJECT: fish: noun)
```

This example shows one term per line with indentation marking the hierarchical structure of the tree represented. The three different types of attribute in each term are separated by single colons. Roles are listed first. These are syntactico-semantic functions. They can be thought of as labels on arcs in the tree. So, for example, *cat* is the SUBJECT of *like* and *fish* is the OBJECT of *like*. Lexemes are listed next. Roles and lexemes express, respectively, the word's syntagmatic and paradigmatic relations. Together they constitute a semantic representation of the sentence. The third part of each term describes the surface properties of the associated word. This consists of a main category — usually a word class — followed by a set of attribute-value pairs. Attributes are, by convention, three-character strings. Values are coded as numbers inside angle brackets.

The analysis employed in PLAIN does not make use of any non-terminal constituents. Neither does it use empty categories. Every node in a dependency tree must correspond to an actual word in the sentence — with one exception. Hellwig argues that

> There must be something to denote the suprasegmental meaning that a clause conveys in addition to the semantics of its constituents. As a necessary extension of DG, the yield of a clause is — so to speak — lexicalized... and represented by a term that dominates the corresponding list (Hellwig 1986: 196).

In order to tether this 'clause' item to something which actually occurs in the sentence, Hellwig associates it with the sentence-final period. The period, after all, serves to mark the ending of a main clause and it can — if so desired — be viewed as a word in a written sentence. Several objections can be raised to this approach. (What about spoken language? What about subordinate

clauses?) Hellwig is aware of these but he argues that the advantage of treating the period as clause head is that it allows a fully consistent system in which all nodes correspond to actually occurring 'words' in the input sentence. He steps into much more dangerous territory when he goes on to suggest that "punctuation in written language can be interpreted as a similar lexicalization of clausal semantics" (Hellwig 1986: 196). However, he does not carry his suggestion any further in practice.

5.2.2 Word order constraints

In addition to the more familiar surface property features such as finiteness, person and number, a DRL term can also include positional features which act as constraints on the relative ordering of words in a sentence. Three such features are reported in the literature: 'seq', 'adj', and 'lim'. These constrain the relative positions of a dependent (D) and a head (H) as follows:

seq This feature relates to linear sequence. It has two possible values:

1. D precedes H
2. D follows H

adj This feature relates to the immediate adjacency of items. It has two possible values:

1. D immediately precedes H
2. D immediately follows H

lim This feature delimits the outermost dependents of a word and thus can be used to mark a 'barrier' across which other dependents of the same word may not be 'moved'. Once again, this feature has two values:

1. D is the leftmost dependent of H
2. D is the rightmost dependent of H

Hellwig presents the DRL term in (22) to illustrate the use of these word order features. The term describes sentence (23), due to Pereira 1981.

(22)

```
(ILLOCUTION: assertion: adj<1>
   (PREDICATE: squeak: adj<1>
      (SUBJECT: mouse: adj<1>
         (DETERMINER: the: seq<1>)
         (ATTRIBUTE: chase: adj<2>
            (OBJECT: that: lim<1>)
            (SUBJECT: cat: adj<1>
               (DETERMINER: the: adj<1>)
               (ATTRIBUTE: like: adj<2>
                  (SUBJECT: that: lim<1>)
                  (OBJECT: fish: adj<2>)))))))
```

(23)

The mouse that the cat that likes fish chased squeaks.

The purpose of these positional features is to produce analyses of sentences — including sentences with discontinuous constituents — which do not make use of transformations, metarules or SLASH feature passing, and which leave no gaps or traces. In this respect, Hellwig's system is similar to Covington's (described in Chapter 10 below): neither recognizes the existence of constituents, either explicitly by means of non-terminal phrase labels or implicitly by means of an adjacency constraint, so for them there is no difference between establishing a dependency between a head and an 'unmoved' word and establishing a dependency between a head and a word which has 'moved' out of its parent 'constituent'. Covington's system works without any positional constraints at all whereas Hellwig's system can use as many or as few positional constraints as are required. Both systems can be constrained to accept only contiguous groups of dependents if necessary. Hellwig's claim is to be able to set positional constraints so as to allow the kind of discontinuous constituency found in natural language and to disallow the sort of discontinuous constituency prohibited in natural language (e.g. movements across barriers). If Hellwig is correct then his system will be impressive indeed. In fact, there is a clue to indicate that

Hellwig's suggestions are fairly tentative since he proceeds to say that "It is likely that appropriate attributes can also be defined for more difficult cases of extraposition" (Hellwig 1986: 197), thereby suggesting that these have not yet been fully explored.

5.2.3 The base lexicon

A base-lexicon is required to associate word forms in the input sentence with lexemes and clusters of morphosyntactic features. The base lexicon consists of a collection of assignments. An assignment consists of a word form to the left of the assignment arrow, and a DRL term to the right of the arrow. The following examples (from Hellwig 1986: 197) show some entries in the base lexicon.

(24)

```
a  CAT    ->    (*: cat: noun num<1> per<3>);
b  CATS   ->    (*: cat: noun num<2> per<3>);
c  LIKE   ->    (*: like: verb per<1,2>);
d  LIKES  ->    (*: like: verb num<1> per<3>);
e  LIKE   ->    (*: like: verb num<2> per<3>);
f  FISH   ->    (*: fish: noun per<3>);
```

None of the entries has been assigned a role. This can only occur during parsing. Entry (24a) has a singular number feature num$< 1 >$ distinguishing it from the plural num$< 2 >$ in (24b). The person feature per$< 1, 2 >$ of (24c) has a disjunction of values. Entries (24d) and (24e) are required for subject-verb agreement. Entry (24f) has no number feature since *fish* can be either singular or plural. Since features are constraints, absence of a feature means absence of any associated constraint.

5.2.4 The valency lexicon

As well as a base lexicon it is necessary to maintain information detailing the kinds of dependents a word may have. It would be possible to enter the information directly in the base lexicon, for example:

(25)

```
(*: like: verb fin<1> num<1> per<3>
   (SUBJECT: _ : noun num<1> per<3> adj<1>)
   (OBJECT: _ : noun seq<2>));
```

The '_' characters are variables. In an analysis of a sentence they would be replaced with lexemes corresponding to the SUBJECT and OBJECT words. '_' variables are knows as 'slots' since they can be filled by dependents. The SUBJECT slot can be read as saying that the subject must be a singular third person noun which immediately precedes its head. The OBJECT slot requires that the object be a noun which occurs somewhere to the right of its head.

The technique of storing valency information in the base lexicon is effective but it fails to capture generalizations. Other forms of the verb *like* will have very similar slots and many other third person singular verbs will have identical slots. Generalizations can be made very simply by storing the shared information in 'completion patterns' and setting up a distinct 'valency lexicon' which associates completion patterns with words. For example, the following completion patterns would be set up for SUBJECT (a) and OBJECT (b):

(26)

```
a  (*: +subject: verb fin<1>
      (SUBJECT: _ : noun num<C> per<C> adj<1>));
b  (*: +object
      (OBJECT: _ : noun seq<2>));
```

The feature value 'C' is used to copy feature values from heads to dependents, i.e. (26a) says that the subject will agree with its head in person and number. Entries in the valency lexicon look like those in (27).

(27)

```
a  (: -> (*: squeak) (: +subject));
b  (: -> (*: like)   (& (: +subject)
                        (: +object)));
```

These state that *squeak* just has a subject slot (it is intransitive) whereas *like* has both subject and object slots (it is transitive). Entries in the valency lexicon control the unification of terms from the base lexicon with stored com-

pletion patterns. Unification is not confined to this task; it is the principal structure-building operation in the grammar. For this reason, Hellwig terms his grammar *Dependency Unification Grammar* (DUG). I prefer to retain this label to designate any DG based on the unification of complex feature structures, and to describe Hellwig's grammar as one variety of DUG (for example McGlashan 1992 describes another variety of DUG).

It is possible to have a disjunction of slots (indicated by a comma at the head of a list of disjuncts) where a dependent can be instantiated in more than one way. For example, Hellwig analyzes relative pronouns as the subjects of embedded sentences. Thus the `+subject` completion pattern can be expanded at least to the following:

(28)

```
(*: +subject: verb fin<1> per<3>
   (, (SUBJECT: _ : pron rel<1,C> lim<1>)
      (SUBJECT: _ : noun num<C> per<C> adj<1>)));
```

We have seen how words in the input string can be associated with role, lexeme and morphosyntactic information in DRL terms. We have also seen how words can be given slots into which dependents can fit. In the next section we shall see how potential dependencies are turned into actual dependencies by the parser.

5.3 The parsing algorithm

The literature does not contain a full, clear exposition of the PLAIN parsing algorithm. The content of this section has been constructed from Hellwig's 1986 COLING paper and from personal communication with Hellwig.

The parser maintains two data structures:

1. A list of DRL expressions corresponding to the words of the input sentence.

2. A queue indicating the order in which words are to be examined. The

queue contains an explicit end-of-queue marker. The parser begins at the left and works towards the right of the sentence so for a sentence with n words (including the period), the queue looks like this: (1, 2,..., n, end-of-queue).

The parsing algorithm uses these two data structures in the following way:

1. Make the word at the head of the queue the current word.

2. Try to find a slot in another word with which the current word can unify. Only adjacent words are tried. There are two possible outcomes:

   (a) A slot is found for the current word. In this case the current word is unified with the slot of its head to form a single partial dependency structure. The pointer to this new structure is placed at the end of the queue.

   (b) A slot is not found for the current word. In this case the pointer to the current word is moved to the end of the queue.

3. Goto 2 until end-of-queue is reached. When this happens move end-of-queue to the end of the queue and proceed to 4.

4. Try to find a slot in another word with which the current word can unify. Only words at one remove (i.e. $n-2$ or $n+2$ are tried. There are two possible outcomes:

   (a) A slot is found for the current word. In this case the current word is unified with the slot of its head to form a single partial dependency structure. The pointer to this new structure is placed at the end of the queue.

   (b) A slot is not found for the current word. In this case the pointer to the current word is moved to the end of the queue.

5. Goto 4 until end-of-queue is reached. When this happens move end-of-queue to the end of the queue and goto 2.

The process terminates when steps 2 and 4 are both executed with no change to the queue.

Hellwig (p.c.) describes this as an island parser. It builds up structure around word 'islands' in the sentence. The object of step 4, which looks beyond the immediate context of an island, is to detect moved parts of a discontinuous constituent.

This is a multi-pass parser. Dependents search for heads but not *vice versa*: heads do not search for dependents. Hellwig makes no claims for the validity of the parser as a psychological model; its motivation is purely implementational and part of the ongoing programme of research is devoted to parallelizing the algorithm.

INITIALIZATION: initialize two lists: Pointer_L and Term_L;
Term_L is an ordered list of DRL terms
corresponding to the words of the sentence;
Pointer_L is an ordered list of pointers
to these DRL terms;
C is a pointer;
C↑ is the term pointed to by C;
C↑:Slot is any valency slot in C↑;
X and Y are variables;
e is not an absolute end-of-list marker
initialize an empty stack: Stack.

1. **IF** C=e

THEN IF top(Stack) = Term_L

THEN IF length (Term_L) = 1

THEN succeed

ELSE fail

ELSE push(Stack,Term_L),
remove(Pointer_L,C),
append(Pointer_L,e),
C:=first(Pointer_L),
goto(2)

ELSE IF C↑ ⊔ C↑−1:Slot

THEN remove(Term_L,C↑),
remove(Pointer_L,C),
C:=first(Pointer_L),
goto(1)

ELSE IF C↑ ⊔ C↑+1:Slot

THEN remove(Term_L,C↑),
remove(Pointer_L,C),
C:=first(Pointer_L),
remove(Pointer_L,C),
append(Pointer_L,C),
C:=first(Pointer_L),
goto(1)

ELSE remove(Pointer_L,C),
append(Pointer_L,C),
C:=first(Pointer_L),
goto(1).

2. **IF** C=e

THEN remove(Pointer_L,C),
append(Pointer_L,C),
C:=first(Pointer_L),
goto(1)

ELSE IF C↑ ⊔ C↑−2:Slot
THEN remove(Term_L,C↑),
remove(Pointer_L,C),
C:=first(Pointer_L),
X:=last(Pointer_L),
remove(Pointer_L,X),
Y:=last(Pointer_L),
remove(Pointer_L,Y),
append(Pointer_L,X),
append(Pointer_L,Y),
goto(2)

ELSE IF C↑ ⊔ C↑+2:Slot
THEN remove(Term_L,C↑),
remove(Pointer_L,C),
C:=first(Pointer_L),
X:=C+2,
remove(Pointer_L,X),
append(Pointer_L,X),
goto(2).
ELSE goto(1)

Algorithm 5.1: Hellwig's dependency parsing algorithm

5.4 The well-formed substring table

One of the most interesting and innovative aspects of Hellwig's parser is his use of a well-formed substring table (WFST) to optimize processing in the parsing of sentences with ambiguity. WFST parsing has been developed in the context of PSG and has not been explored to any great extent in dependency-based systems. The normal conception of a WFST is of a structure with nodes and

edges. To begin with, there are as many edges as there are readings for the words in the input sentence. When a constituent is built an edge is inserted which spans all of the words which the constituent contains.

Hellwig's WFST is very like this except that his edges are labelled with DRL descriptions of the words spanned. These descriptions may contain slots. When a word becomes a filler for another word's slot, the two are unified and a new edge is inserted in the WFST spanning what was previously spanned by the two edges. Hellwig's WFST for the globally ambiguous sentence *Flying planes can be dangerous* (Hellwig 1988: 243) is shown in Figure 5.2.

However, the standard view of a WFST assumes that constituents are continuous. An edge serves to mark everything between its end-points as belonging to one constituent. The edge is labelled with the name of that constituent. This is not sufficient for Hellwig's parser, which advertises as one of its benefits the ability to parse discontinuous constituents. If a constituent is discontinuous, simply marking its left and right boundaries does not serve to identify its components since, by virtue of the discontinuity, some of the material between the endpoints will not belong to the constituent.

Hellwig's solution is to adopt a word-centred rather than a constituent-centred approach to WFST parsing. This he does by assigning a bit string to each word in the input sentence. Each bit string in an n-word sentence consists of one '1' and n–1 '0's. The ith word is represented by a bit string with the '1' in ith position. Before any attempt is made to establish a dependency relation between two words, their bit strings are added. If the addition involves any 'carry' operations (i.e. a 1 is added to a 1) then the dependency is prohibited even before the slot features have been checked. If no 'carry' operations are involved, the process may proceed. In this way a WFST can be built up for discontinuous constituents.

For example, the words of sentence (29) would be assigned the initial bit strings shown in (30) (trailing zeros in bit strings and features in DRL slots

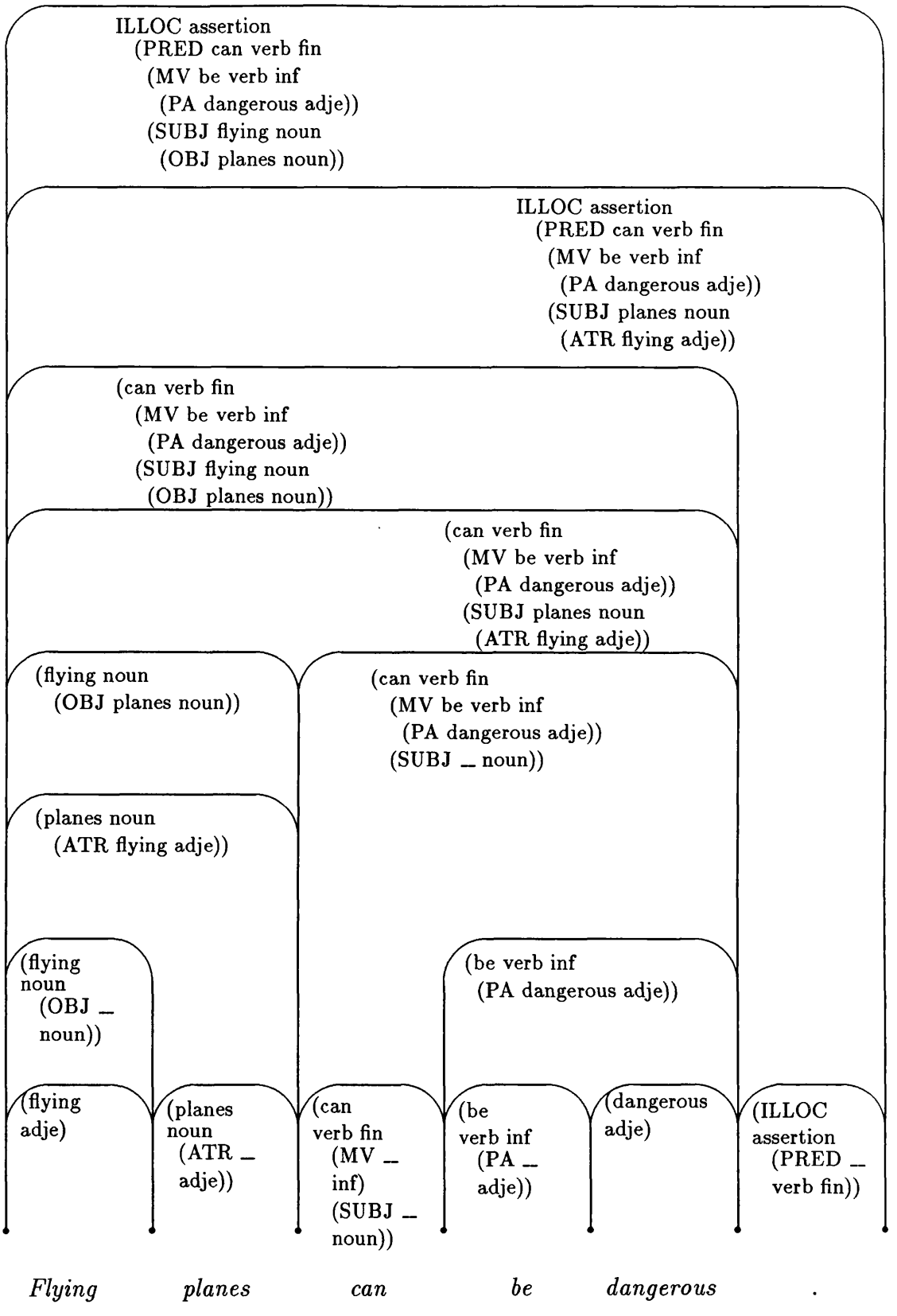


Figure 5.2: Hellwig's WFST for *Flying planes can be dangerous*

are suppressed for readability).

(29)

What did Danforth say to George?

(30)

```
BITSTRING    TREE
1            (what pron)
01           (do verb fin
                (SUBJECT: _ )
                (MAINVERB: _ ))
001          (Danforth noun)
0001         (say verb inf
                (DIRECTOBJECT: _ )
                (INDIRECTOBJECT: _ ))
00001        (to
                ( _ )
000001       (George noun)
0000001      (ILLOCUTION question
                (PREDICATE: _ )
```

In (29), *What* is the direct object of *say.* The discontinuous tree rooted in 'say' is represented unproblematically in Hellwig's WFST as shown in (31).

(31)

```
BITSTRING    TREE
100111       (say verb inf
                (DIRECTOBJECT: what )
                (INDIRECTOBJECT: to
                   (George)))
```

What Hellwig has done is to discard the notion of 'constituency' and replace it with the notion of 'consistency'.

What is missing from the PLAIN literature is a description of exactly how the WFST is used in the parsing algorithm to increase the efficiency of the parser. Hellwig consistently describes his system as a 'chart parser' thereby implying a more sophisticated control mechanism than is necessary in a simple WFST parser. The omission is particularly disappointing since Hellwig's system is, to the best of my knowledge, the only dependency parser to make use of a WFST in the management of ambiguity. We shall return to this topic in Section 12.6 below.

Table 5.1: main features of Hellwig's dependency parser

Search origin	bottom-up
Search manner	depth-first
Search order	left to right
Number of passes	at least two
Search focus	dependents seek heads
Ambiguity management	WFST (adjacency not enforced)

5.5 Summary

Hellwig uses a simple unification grammar expressed in terms of complex feature structures. His parser has a bottom-up island-driven control strategy which is claimed to be able to build discontinuous constituents without recourse to special registers or feature passing (although more information on the precise use of the `lim` feature is required before the system can be properly evaluated). Words look for heads; they never look for dependents. The parser's efficiency is increased by the use of a WFST which differs from standard WFST parsers in building dependency rather than constituency structures and in representing non-contiguous collections of dependents.

The main features of Hellwig's parser are summarized in Table 5.1.

Chapter 6

The Kielikone parser

6.1 Overview

In this chapter I examine the Kielikone dependency parser. Since June 1982 the Finnish National Fund for Research and Development ('SITRA') has sponsored a research project known as 'Kielikone' at the Helsinki University of Technology. The aim of the project is the development of a computer system for the automatic interpretation of written Finnish. The main application focus of the research is the design and implementation of a Finnish text interface to computer databases. However, the object is to produce an interface which is independent of any single database so that it can be ported to many applications. The overall structure of the interface system — which has recently come to be known as 'SUOMEX' — is described in Jäppinen et al. (1988a).

Sentence processing in the Kielikone system is achieved by four distinct modules.

1. A morphological analyser known as 'MORPHO' breaks words down into their component morphs (Jäppinen et al. 1983; Jäppinen and Ylilammi 1986). This is vital in an agglutinating language like Finnish since a full form lexicon would be much larger than for a language like English which has much less morphological variation. By 1987 the lexicon contained over 35000 lexical entries (i.e. stems) (Valkonen et al. 1987a).

2. A parser, known as 'ADP' (Augmented Dependency Parser), uncovers the dependency structure of sentences. It is this module which will be the focus of investigation in this chapter.

3. A logical analyser is responsible for constructing the propositional meaning of sentences and also for interpreting sentences in their dialogue context. Thus the module embraces both semantics and pragmatics. In early 1987 this module was referred to as 'DIALOG' (Jäppinen et al. 1987: preface); by 1988 its name seemed to have changed to 'AWARE' (Jäppinen et al. 1988a: 335).

4. The fourth module appears not to have a name. It serves as the buffer between the natural language understanding module and the database. Its task is to transform interpretations of Finnish sentences into sequences of formal database queries. In order to make this a general purpose portable interface, queries are couched in a database interlingua called 'UQL' (Universal Query Language). To interface the system to any specific database it is only necessary to write an interpreter to translate UQL queries into the format expected by the specific database, e.g. SQL.

Some of the dependency parsers covered in this thesis are described on the basis of just one or two papers or reports. With the Kielikone parser there is an abundance of documentation. A Kielikone bibliography published in 1987 lists 53 items, of which 14 are specifically concerned with parsing. This abundance of literature is obviously very welcome to the student of dependency parsing. However, it does introduce some problems of version control. During the lifetime of the project a number of changes in direction have been made and it is difficult to keep track of exactly which incarnation of the system is being described at any given point. As we have already seen, many of the components in the system have been given names. When new names appear it is not always clear whether (i) only the names have changed while the

components remain the same, (ii) the new names introduce new components to complement the existing components, or (iii) the new names introduce new components to supersede old components. This would all be self evident were it not for the fact that SUOMEX is a very complex system and most published papers can only discuss selected sub-parts of it. It is thus necessary to try to guess whether elements which are not mentioned have been left out for lack of space or because they have been quietly dropped from the system. The parser itself suffers from this problem since, as we shall see, its internal structure is also rather complex.

6.2 Evolution of the parser

In order to aid exposition, I shall plot the main milestones in the development of the parser before turning to a more detailed examination of the most recent version.

6.2.1 The earliest version: two way finite automata

The earliest descriptions of the parser appeared in 1984 (Nelimarkka et al. 1984a; Nelimarkka et al. 1984b). At that stage the developers of the parser were emphasizing three main points:

1. The grammar was based on the notion of *functional dependency.*
2. 'Constituents' were built *middle out.*
3. The parser built structure using *two-way finite automata.*

Functional dependency grammar

The parser builds dependency structures consisting of pairs of words in binary antisymmetric dependency relationship with each other. The words involved in dependency relationships are identified using a 'regent-dependent' nomenclature. Non-terminal phrase nodes or labels do not appear anywhere in the

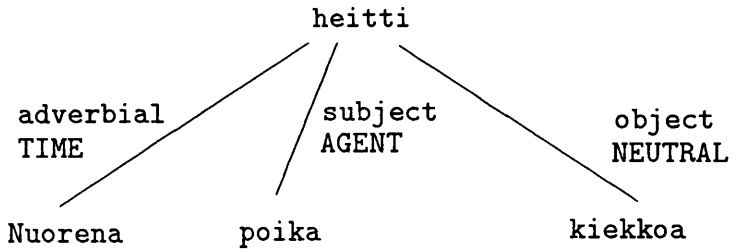


Figure 6.1: a functional dependency structure

system. However, the term 'constituent' is used consistently to refer to a word plus all of its (direct or indirect) dependents. It is even (confusingly) used to refer to a single word which has no dependents. The word on which all others depend (directly or indirectly) in a constituent is the 'head'. Different kinds of dependency are recognized and these are linked with the traditional syntactic functions (or relations) subject, object, adverbial, genitive attribute, etc. These, in turn, are associated with semantic interpretations such as AGENT, NEUTRAL, DIRECTIVE, etc.

For example, the sentence *Nuorena poika heitti kiekkoa* ('When he was young the boy used to throw the discus') is given a stemma analysis as shown in Figure 6.1 (example cited in Nelimarkka et al. 1984a: 169). This combination of dependency, syntactic function, and deep case is what is referred to by the term 'functional dependency grammar'.

Middle-out structure building

The parser is described as being strongly data driven, left-to-right, and bottom-up. It is also described as building a constituent from the middle outwards. This seems slightly inconsistent: left-to-right suggests one control strategy, middle-out suggests another. In fact, the parser is only left-to-right in the sense that it sees word 1 before it sees word 2. It may actually end up building constituents at the end of the sentence before it has built any at the beginning.

Overall, the strategy is very close to that of an island parser which starts constructing 'islands' as close to the beginning of the sentence as it can.

Suppose that the string the parser is operating on consists of constituents $C_1 - C_n$ (remember, a single word can be a constituent and, if the constituent consists of more than one word, only the head is visible externally). Middle-out control works as follows:

1. Try to recognize C_{i-1} as a dependent of C_i.
2. Try to recognize C_{i+1} as a dependent of C_i.
3. Shift the focus to C_{i-1} or C_{i+1}.

Notice that the parser only attempts to link *immediately* adjacent (i.e. neighbouring) constituents. If constituent A meets the dependency requirements of constituent B, then constituent A is 'absorbed' into constituent B and so disappears from sight of the parser. Constituent B now has a new neighbour and so the parser can attempt to establish a new dependency link between them.

The parser can be envisaged as consisting of a register holding the current constituent, plus two stacks, one storing the left context, the other storing the right context (see Figure 6.2, due to Lehtola et al. 1985).

The current constituent C either establishes a dependency link with L1 or R1, or it is pushed onto one stack and the current constituent register is filled from the top of the other stack. The parser is constructed so as always to search the immediate left context first.

Two-way finite automata

The grammar stores information concerning binary dependency relations and their corresponding functions. However, it is also necessary in this system to store information specifying what all constituents may contain. In other words, it is necessary to store for each word type a complete record of all its

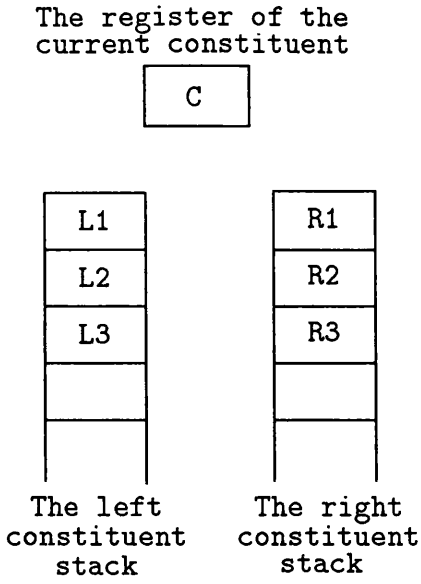


Figure 6.2: left and right context stacks

obligatory and optional dependents. This can then serve as a model for actual occurrences of that word type. For this task the system uses two-way finite automata.

A two-way finite automaton (Levelt 1974) consists of a set of states. One of these is distinguished as the start state and one or more are distinguished as final states. The states are linked by transition arcs between the states. Each arc recognizes a sentence element and moves the reading head either to the right or to the left in the input string. The automaton accepts an input string if it begins in the start state with the first word under the reading head and proceeds to a final state, leaving the reading head pointing to the right of the last word in the input string.

The standard idea of a two-way finite automaton is modified somewhat in the Kielikone system. Instead of recognizing words in the input string, each automaton recognizes functions like subject, object, etc. Each arc traversal also serves to build some structure, namely to insert a dependency relation between two neighbouring words. The dependency relation is labeled with

the name of the function specified by the arc traversed. States are divided into 'left' and 'right' states indicating the side of the current word on which dependents so marked will be found. Thus, *contra* Covington (1990b), relative position is expressed explicitly in the grammar of a free word order language.

It has been known for some time that any language recognized by a two-way automaton is regular (i.e. type 3, the most highly constrained set of languages in the Chomsky Hierarchy). This power is not sufficient for the requirements of natural language. To increase the recognition power, several automata are made to activate one another. They make use of three 'control' arcs which shift processing from the current word to one of its neighbours. These control operations are 'BuildPhraseOnRight', 'FindRegOnLeft', and 'FindRegOnRight'.

When an automaton has found all of the obligatory dependencies associated with a given word, the final action of the automaton is to mark the head '+phrase', thus indicating that the constituent is complete. Other, more specific, features may also be used, e.g. '±sentence', '±nominal', '±main'.

It is worth noting that automata 'know' nothing about when and why they were activated. This distributed control (or 'local control' as it is referred to by Kielikone researchers) ensures that parsing is strongly data driven. Careful ordering of function and control arcs in the automata is said to result in very little backtracking being necessary.

Automata are fairly complex objects in the Kielikone system. The only automaton to be described in the Kielikone literature can be found in Lehtola et al. (1985: 99).

6.2.2 A grammar representation language: DPL

It is not clear from the literature whether the representation language described in this section was developed concurrently with the components covered in the section above or whether it represents a subsequent step.

The language, 'DPL' (Dependency Parser Language), is a representation language developed as part of the Kielikone project (Lehtola et al. 1985;

Lehtola 1986). All functions, relations and automata were, at one time, expressed in this unified representation language.

Given that DPL abbreviates 'Dependency Parser Language', it seems somewhat incongruous that "the main object in DPL is a constituent" (Lehtola et al. 1985: 100). However, this can be read as meaning 'the main object in DPL is a word plus all its properties, including its dependents'. The grammar writer specifies an inventory of permitted property names and values. These can then be built into descriptions. A number of operators are available to relate objects to each other and to perform actions on objects, including the following:

= equality

:= replacement

:- insertion

<> mark the scope of an implicit disjunction

() mark the scope of an implicit conjunction

→ perform all operations on the right

⇒ terminate execution after first successful operation

The definition of Subject shown in Figure 6.3 should serve to illustrate what a DPL entry looks like. This example is taken from Lehtola et al. (1985: 102). I shall not discuss its detail here. The important point to note is that the grammar writer is forced to write a procedural grammar. It is generally acknowledged that procedural grammars — other than grammars for tiny fragments — are much harder to write, to understand, to modify and to port than declarative grammars so it could be argued that DPL is not the best representation on which to base a parser. Notice that the grammar writer is charged with the task of defining the automata in DPL as well as the task of defining the functions and relations in the grammar. Fairly minor modifications to the grammar could be expected to require a lot of hard work.

```
(FUNCTION:    Subject
              ( RecSubj -> (C := Subject))
)

(RELATION:    RecSubj
              ((C = Act < Ind Cond Pot Imper >) (D = -Sentence +Nominal)
                  -> (D = PersPron (PersonP R) (PersonN R)
                     ((D = Noun) (C = 3P) -> ((C = S) (D = SG))
                                             ((C = P) (D = PL))))
                     ((D = Part) (C = S 3P)
                         -> ((C = 'OLLA)
                                 => (C :- +Existence))
                            ((C = -Transitive +Existence))))
)
```

Figure 6.3: a DPL definition of Subject

Before moving on to examine the next development in the Kielikone project we must note a cryptic comment buried in one of the papers describing the DPL representation language:

> An automaton can refer up to three constituents to the right or left using indexed names: L1, L2, L3, R1, R2 or R3 (Lehtola et al. 1985: 101).

Everything else in the Kielikone literature seems to suggest that the only constituents in sight of the current word are its immediate left and right neighbours. The above comment seems to suggest that the parser really has three-cell lookahead and lookback buffers, rather like Marcus's deterministic PARSIFAL system (Marcus 1980) (which has a three-cell lookahead buffer). This would be a very important point if it were the case. However, since nothing else in the literature points in this direction we must simply place a question mark beside the above remark, and proceed.

6.2.3 Constraint based grammar: FUNDPL

As I have previously observed, DPL presented the grammar writer with a fairly unwieldy formalism. The grammar writer was required to work out complex control issues. This problem was acknowledged by the Kielikone team who

responded by designing a more user-friendly high-level representation language called 'FUNDPL' (FUNctional DPL).[1] FUNDPL is built on top of DPL so its functionality is exactly the same. The crucial difference is that the grammar writer is no longer required to worry about control issues (at least, not to the same extent). FUNDPL is described in Jäppinen et al. (1986).

FUNDPL is basically a constraint system. As such, it is claimed to be related to other constraint-based grammars such as LFG (Kaplan and Bresnan 1982), FUG (Kay 1985), and GPSG (Gazdar et al. 1985). In common with these systems FUNDPL allows the grammar to be written as a set of well-formedness constraints. Conceptually, the job of the parser is to search for an analysis of the sentence which does not violate any constraints. However, unlike these other systems, FUNDPL grammars are *not* unification grammars. FUNDPL is simply a high level interpreter which maps declarative FUNDPL structures onto procedural DPL structures. The main benefit of FUNDPL is that DPL, with all of its procedural complexity, is no longer visible to the grammar writer. It is no longer necessary to think in terms of two-way finite automata.

Functional schemata

FUNDPL constraint structures for the description of constituents are known as *schemata*. Each schema has four parts: pattern, structure, control, and assignment, as shown in Figure 6.4

A schema is triggered by matching the properties of a constituent with those in the **When** slot of the schema. (Presumably the slot is named to signify something like 'when this pattern is matched, use the schema'). The *structure* part of the schema lists optional and obligatory dependents for the head of the constituent. The **Order** slot specifies any ordering (concatenation) restrictions which may apply. For example, **Order** = <D1 D2 R> states that D1 must

[1]The pronunciation of this acronym is not known.

F_SCHEMA: name	
When = [properties]	*pattern*
Obligatory = (functions)	
Optional = (functions)	*structure*
Order = <conc.description>	
TryLeft = <functions>	
TryRight = <functions>	*control*
Down	
Up	
Assume = [properties]	*assignment*
Lift = function(attributes))	

Figure 6.4: the general form of functional schemata

precede D2 which in turn must precede the regent. Irrelevant intervening material is indicated by two consecutive dots (..). **Order** = <D1..R..D2> requires D1 to appear somewhere to the right of R and D2 to appear somewhere to the left of R. The **Order** slot may be empty. The *control* part of the schema consists of heuristic information to guide the parser's search order. This is stored in the **TryLeft** and **TryRight** slots. If a word's dependents are *usually*, though not necessarily always, found in particular locations, the heuristic information can cut down average search time considerably. **Down** and **Up** are used to change levels between matrix and subordinate sentences. Their use is not well documented. Presumably their purpose is to prevent constituents at one level from being confused with those at another level; it is not clear how they work and no examples are available. Clearly, the designers of FUNDPL are being somewhat optimistic when they say that their system "liberates a grammar writer from control anxieties" (Jäppinen et al. 1986). The **Assume** slot assigns new features (e.g. +Phrase) to the regent once the schema has been fully matched and bound. The **Lift** slot is like the **Assume** slot except that it copies features from a dependent to the regent. For example, '**Lift**=Subject(Case)' copies the Subject's case feature to the regent.

The example shown in Figure 6.5 appears in Jäppinen et al. (1986: 463). It is the functional schema for normal Finnish transitive verbs which allow unlimited adverbials on either side. The schema allows all ordering permutations

(**F_SCHEMA**: VPTrAct
When = [Verb Act Ind + Transitive]
Obligatory = (Subject Object)
Optional = (Adverbial*)
TryLeft = <Subject Object Adverbial>
TryRight = <Object Adverbial Subject>
Assume = [+Phrase +Sentence])

Figure 6.5: a schema for Finnish transitive verbs

```
((R = Verb Act
     < (< Ind Cond Imper Pot IIpartis > (PersonP D)(PersonN D)
        -Negative -Auxiliary)
       (Auxiliary IIpartis Nom -Negative)
         (Negative < (Imper Pr < (S 2P) Neg >)
                     (Cond Pr S 3P) (Pot Pr Neg)
                     (IIpartis Nom)> -Auxiliary)>)
 (D = PersPron Nom))...
```

Figure 6.6: the binary relation 'Subject'

among dependents but it 'prefers' SVO order.

Binary relations

Notice that functional schemata specify the possible components of a constituent. They do not contain any information detailing what might constitute a legitimate dependent of the regent. For example, the schema in Figure 6.5 records that a transitive verb requires a subject but it says nothing about what may legitimately serve as a subject. In the FUNDPL system, functional schemata — which are generalized descriptions of the structure of constituents — are completely distinct from *binary relations* which define all permitted dependency relations which may hold between pairs of words in Finnish sentences. Binary relations are boolean expressions which succeed if all conditions are met, otherwise they fail. Unfortunately, the literature offers only half a binary relation by way of example. This half, which is part of the binary relation 'Subject', is shown in Figure 6.6 (Valkonen et al. 1987b).

The regent R must be an active verb. Further restrictions appear within

```
(CATEGORY : SynCat
          < (Word)
                (Noun ! Word)
                  (Proper ! Noun)
                  (Common ! Noun)
                (Pronoun ! Word)
                  (PersPron ! Pronoun)
                  (DemPron ! Pronoun)
                  (IntPron ! Pronoun)
                  ...
```

Figure 6.7: the 'SynCat' category

the disjunctive angle brackets. '–' expresses negation. The dependent D must be a personal pronoun. The significance of round brackets is not clear. If the conditions for both R and D are satisfied then the value of the relation is 'True', i.e. a dependency relation can be established between them.

Type definitions

A FUNDPL grammar includes type definitions of three varieties: CATEGORIES, FEATURES, and PROPERTIES.

CATEGORY definitions set up hierarchical relations amongst names. This allows properties to be inherited automatically by lower individuals from higher individuals in the hierarchy. For example, a category SynCat, consisting of a word class hierarchy, would be defined as shown in Figure 6.7 (Valkonen et al. 1987b: 219).

The '!' symbol can be read as 'isa'.

FEATURE definitions record the names and possible values of features.

PROPERTIES are like features except that they can have default values. For example, the following property definition (from Valkonen et al. 1987b: 219) records the fact that 'Polar' can have two values, 'Pos' or 'Neg'. The value of 'Polar' is 'Pos' by default.

(32)

```
(PROPERTY: Polar < ( Pos ) Neg >)
```

Lexicon

The FUNDPL lexicon records idiosyncratic, non-inferrable features for words. Thus it consists of word:feature structure pairings.

This concludes my sketch of the evolution of the Kielikone parser. Inevitably, some features have not been covered. Some of these were left out because they were minor, ephemeral suggestions. Others were left out because the literature contains insufficient or confusing information. For example, Kettunen 1986 mentions a parser called 'DADA' (an acronym from the unlikely designation 'Dependency Analysis is Dependency Analysis'!) and describes it as being part of the Kielikone system. The parser is never heard of again so it is hard to tell whether it was a short-lived alternative to the older system or simply a confusion of names.

In the next section I explain how the FUNDPL components I have described fit together in the most recent version of the Kielikone parser.

6.3 The parser

The best texts describing the present state of the parser are Valkonen et al. (1987b) and Kettunen (1989). There is not full agreement between these papers — they even disagree about the name of the parser! Valkonnen et al. call the parser ADP and describe FUNDPL as a declarative high level language. Kettunen consistently refers to FUNDPL as a parser, even in the title of his paper *Evaluating FUNDPL, a dependency parser for Finnish.* However, since Kettunen's usage seems to be idiosyncratic I shall ignore it.

6.3.1 The grammar

The grammar accepted by the parser is written in FUNDPL. It consists of the four components described in the previous section, namely

1. Type definitions, consisting of definitions for categories, features and properties.

2. A lexicon for associating features with words. Recall that the SUOMEX system includes a morphological analyzer, MORPH (Jäppinen and Ylilammi 1986), which analyzes words into their component morphs. The role of the lexicon in the grammar is simply to add information which cannot be predicted from general principles.

3. Binary dependency relations which are boolean evaluation functions to determine whether the features of any two words are such as to allow them to enter a dependency relationship.

4. Functional schemata, consisting of definitions of the structure of constituents. These may be under-specified so, for example, relative word order may not be defined, thus allowing any ordering.

6.3.2 Blackboard-based control

The structure of the parsing system is represented by the diagram in Figure 6.8 which appears in Valkonen et al. (1987a: 700) and Valkonen et al. (1987b: 221).

The account of the system's structure offered by its designers proceeds as follows.

The system has two knowledge sources, a body of functional schemata and a body of binary relations (i.e. boolean expressions). These two knowledge sources do not communicate directly. Instead, they read from and write to a shared data structure known as a 'blackboard'. When a word becomes the current word its properties are matched against the triggering patterns of the functional schemata (i.e. the values of the **When** slots in the schemata). Only one match can be entertained at any one time. A matching schema is used to create an 'active environment' associated with the constituent to be built around the current word. This active environment is located on the blackboard and is monitored by the binary relations. These are used to indicate when the properties of a regent and a candidate dependent are such as to

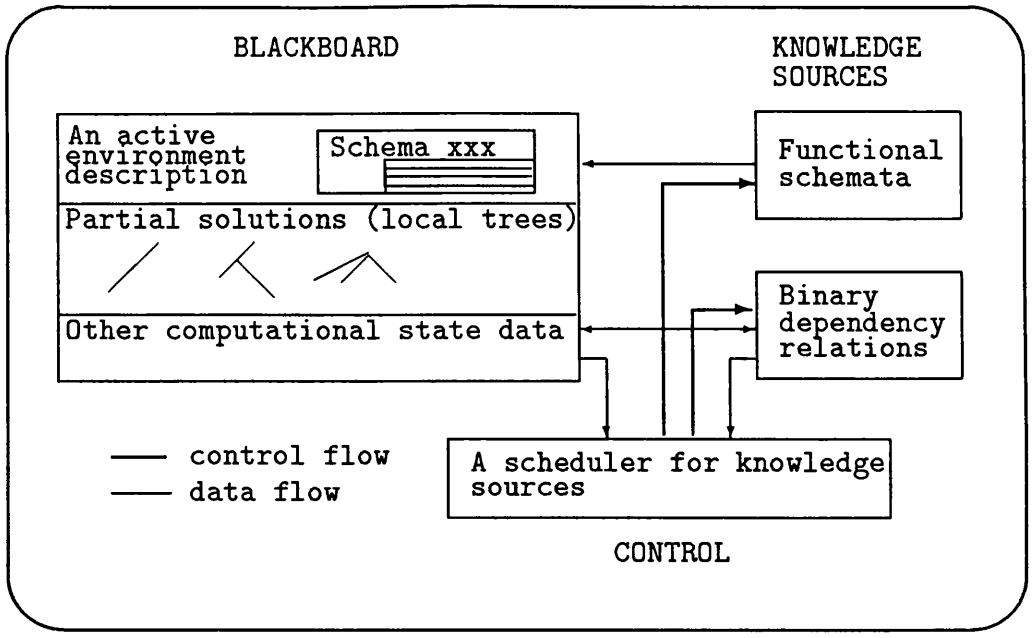


Figure 6.8: architecture of the Kielikone parser

allow a dependency link to be established. When the prevailing conditions allow linking, the partial dependency tree is built by "dependency function applications" (Valkonen et al. 1987b: 221). It is not clear what these are or where they fit in the above diagram. This process continues until all of the obligatory slots (and perhaps some optional slots) have been filled in the active environment. At this point the local partial dependency tree is complete and processing can shift to another constituent with another active environment, unless, of course, the constituent to be completed has a main verb (+Sentence) as head in which case the parse is complete.

The blackboard is a well known data structure in artificial intelligence (Hayes-Roth et al. 1983; Nii 1986). The principle behind blackboard systems is that several component processes (or *knowledge sources*) can collaborate in the construction of objects residing on the blackboard. The order in which objects are added to the blackboard is determined by the availability of information to the processes. Thus, a knowledge source can be thought of as

a demon watching the blackboard until something appears which that demon is able to process. The demon writes the resulting structure to the blackboard and returns to a semi-dormant monitoring state. In this way, different knowledge sources can collaborate to achieve some task.

An example of this kind of blackboard system is the HEARSAY-II speech understander (Erman et al. 1981) which used a blackboard to keep track of the sentence analysis being developed by several different knowledge sources.

Whether or not this degree of architectural sophistication is really necessary in a dependency parser is open to question. The motivation for using a blackboard is usually that it is necessary to apply several knowledge sources to each structure in order to generate a solution. In the Kielikone parser there are only two knowledge sources, namely the functional schemata and the binary relations. It is not even clear that these need to be separate knowledge sources. The division is not justified anywhere in the Kielikone literature and a number of other dependency parsers described in this thesis seem to work adequately without any such division of labour.

6.3.3 The parsing algorithm

In this section I describe the parsing algorithm. Before getting too close to the detail it is worth attending to the designers' high-level description of what their system does:

> In analysis two abstract levels exist. On the regent level (R-level) are those constituents which lack dependents to fill some required functional roles. On the dependent level (D-level) are those constituents which have become full phrases (marked by the feature +Phrase) and are therefore candidates for functional roles... The underlying abstract view is this. A word enters the parsing process via R-level. When all dependents of the constituent (the word) have been bound (from D-level), it descends to D-level. There it remains until it itself becomes bound as a dependent. Then it vanishes from sight (Jäppinen et al. 1986).

The parsing algorithm is defined by a two-way finite automaton. This is not to be confused with the two-way finite automata originally used by the grammar writer to define functional schemata and still built on the fly by the FUNDPL interpreter. The parsing algorithm embodied in the automaton consists of five main steps, namely:

1. One of the schemata associated with the current constituent is activated.
2. Search for left-side dependents for the current constituent.
3. The current constituent is waiting for the building of the right context.
4. Search for right-side dependents for the current constituent.
5. The schema associated with the current constituent has been fully matched and becomes inactive. The current constituent is now a completed (partial) dependency tree.

No more than one schema may be active at any one time, i.e. only one constituent may be at step 2 or step 4 in the automaton. However, any number of constituents may be at step 3. These are termed 'pending' constituents and are implemented as a PENDING stack. Parsing starts with the first word and proceeds to the right. A sentence is well-formed if the parsing process yields a single constituent in step 5.

We shall now consider each step in the algorithm in greater detail:

1. All constituents have heads, whether they consist of single words or complex dependency structures. A schema, whose **When** features match the head features of the constituent, is activated. It is not clear whether matching must be exact or more like unification, i.e. there is a match if there is no conflict. Move to step 2.
2. Left-side dependents are searched for on the basis of the dependency requirements stated in the active schema. There are two possible outcomes:

(a) There are no left neighbours or left neighbours are at step 3, pending. Go to step 3.

(b) The left neighbour is in step 5 (i.e. is a complete constituent). Binary relation tests are carried out to establish whether or not it is a suitable dependent. If it is then the left neighbour is subsumed in the current constituent which re-enters step 2 (now with a new left neighbour). If binary relations fail, the active schema enters step 3, pending.

3. There are two possibilities here:

   (a) There are no right neighbours. Go to step 5.

   (b) There are right neighbours. Push the current constituent on the PENDING stack and go to step 1 with the next constituent to the right (i.e. read in the next word).

4. Search for right-side dependents. If binary relation tests succeed then subsume each dependent in the current constituent. Return to step 3.

5. There are two possibilities:

   (a) If no constituents remain other than the current constituent then the sentence has been successfully parsed. If right-side constituents exist then go to step 1 with the next constituent as input (i.e. get next word from input). If neither of these succeed then go to step 4 and pop PENDING.

   (b) Failure.

The control strategy automaton is shown in Figure 6.9.

The above description has been constructed following published descriptions of the Kielikone parser as closely as possible. A PARS description of the Kielikone parsing algorithm is given below:

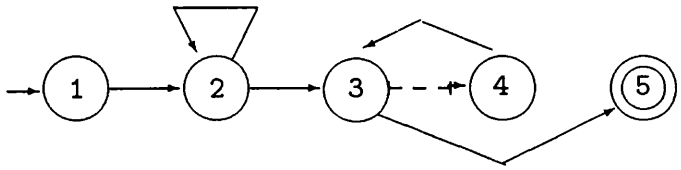


Figure 6.9: the Kielikone parser control strategy automaton

INITIALIZATION: read input words into a list;
C is the current word;
C:=1;
initialize an empty stack;
Result is the result variable;
'saturated(C)' is a condition which succeeds iff
C's valency requirements have been satisfied.

1. **IF** (C=1 v C–1=top(Stack))

   THEN goto(2)

   ELSE IF (saturated(C–1) & C → C–1)

   THEN record(C → C–1),
   remove(C–1),
   goto(1)

   ELSE C:=top(Stack),
   pop(Stack),
   goto(3).

2. **IF** (saturated(C) v C+1=e)

   THEN goto(4)

   ELSE push(C),
   C:=C+1,
   goto(1).

3. **IF** (saturated(C+1) & C → C+1)

   THEN record(C → C+1),
   remove(C+1),
   goto(2)

   ELSE fail.

```
4. IF (C+1=e & C-1=0 & empty(Stack))
   THEN Result:=C,
        succeed
   ELSE IF C+1=e
        THEN C:= top(Stack),
             pop(Stack),
             goto(3)
        ELSE C:=C+1,
             goto(1).
```

Algorithm 6.1: the Kielikone dependency parsing algorithm

The basic parsing strategy should be obvious. Each schema becomes active and continues to be active until either it builds a complete constituent or it goes to sleep to wait for the constituents it requires to be built. As the Kielikone parser is described, an active schema is just the data structure that happens to be being manipulated at the present moment. Active schemata are not, of themselves, either active or inactive: they are simply representations. They are interpreted as being active or inactive according to whether the parser is currently trying to satisfy the dependencies specified by them. The situation would be completely different, if each schema were actually a process rather than a representation. This would make for rather an interesting parser which would bear a family resemblance to a Word Expert Parser (Small 1983), a parser which consists of a set of interacting processes, each of which is an 'expert' on some word in the lexicon. This flavour of system is mentioned briefly in the closing remarks of Valkonen et al. (1987a: 702):

> We argue that our blackboard-based computational model also gives a good basis for parallel parsing. There should be an own processor for each word of the input sentence. The partial dependency trees would be built in parallel and sent to the main process that links them into a parse tree covering the whole sentence.

For a parallel Word Expert Parser see Devos et al. (1988).

6.3.4 Ambiguity

Ambiguities arise in the system due to indeterminacies of three distinct kinds: choice of analyses for homographic words, choice of schemata, and choice of dependency relations. In parsing, a record is kept of all choice points and exhaustive enumeration of all possible readings of a sentence is produced by chronological backtracking. This is not a computationally efficient approach to ambiguity since it can result in identical structures being built many times over.

6.3.5 Long distance dependencies

Under normal circumstances, dependency relations are established between immediately neighbouring constituents. However, this is not possible in the case of long-distance dependencies where, by definition, part of a constituent is moved out of its normal position into another, inaccessible, position.

> Long-distance dependency is caused by an element which has moved from the local environment of a regent to the local environment of another regent (Valkonen et al. 1987b: 220).

In order to deal with long-distance dependencies, a minor modification is made to the grammar and the parser. The modification to the grammar involves marking schemata which can become possible neighbours of moved items as having a special (optional) 'DistantMember' dependency function. This can act as a place-holder for the moved item which is said to be 'captured'. The schema from which the constituent can be moved is marked with a 'DISTANT' clause indicating which dependents could possibly be moved out of the immediate vicinity of the constituent. For example, a schema might contain the entry:

```
(DISTANT Object)
```

indicating that an object could be a possible candidate for movement.

A modification to the parser is also required. The parser is given an extra register. Any captured constituents are copied from the 'DistantMember' slot of the 'host' schema into the special purpose register. This register must be checked in addition to a constituent's immediate neighbours during the parsing process. If the item in the register is found to satisfy a dependency requirement of the current constituent, it can be copied from the register into the current constituent as a dependent. (I assume — although this is not stated explicitly — that the register is only checked if a dependency can not be satisfied by more conventional means). After initially being copied into the special register, the captured constituent is no longer visible in the constituent which captured it. This could be described as a 'swooping' analysis. The 'DistantMember' dummy dependency is similar to the 'Visitor' relation in Word Grammar (Hudson 1988b: 202ff; also 189 below). However, unlike WG, the Kielikone solution does not appear to handle 'island constraints' (Ross 1967). One such constraint stipulates that extraction out of a complex noun phrase (e.g. *the claim that Saddam is a wonderful host*) is prohibited. There is nothing in the Kielikone parser's treatment of movement to stop it from accepting a sentence with this kind of prohibited extraction, i.e. it would parse both of the sentences in (33).

(33)

a Nobody believes the claim that Saddam is a wonderful host.
b *What does nobody believe the claim that Saddam is?

'DistantMember' is directly analogous to the HOLD register in Augmented Transition Networks (Woods 1970) and is thus subject to the same kinds of criticisms. (It is an ad hoc device, it is descriptively inadequate, etc.)

6.3.6 Statistics and performance

The most recent available figures (Valkonen et al. 1987b: 225) report that the system contains 66 binary relations, 188 functional schemata and 1800 idiosyncratic lexical entries. The lexicon of the separate MORPHO morphological

analyzer contains 35000 entries.

It is claimed that a recent modification to the parser (discussed below) parses unambiguous sentences in linear time. This sounds impressive but is misleading. It is not unusual for dependency parsers to operate in linear time on unambiguous sentences (for example, my own parser described in Chapter 9 does so). It is also the case that there exists a class of ambiguous languages (which is hard to describe in intuitively comprehensible terms) which can be parsed in linear time by parsers based on context free grammars. (Some examples are given in Earley 1970). It is normal to cite *worst case* or possibly *average case* complexity rather than *best case* complexity in order to evaluate a parser. Unfortunately, these figures are not published for the Kielikone system.

6.3.7 Open questions

Theoretical status

Unlike some of the other dependency parsers reviewed in this thesis (e.g. the Lexicase and Word Grammar parsers, Chapters 8 and 9), the Kielikone parser is not based on a linguistically motivated theory. In spite of the fact that Finnish has fairly free word order, it does not have a tradition of DG scholarship as is the case with, for example, German and Russian. Indeed, Tarvainen (1977) is one of the few texts which makes any attempt at analysing Finnish syntax in terms of DG and this work is not mentioned in the (English) Kielikone literature.

There seems to be some uncertainty as to the status of the Kielikone parser. Obviously, it is an NLP system with a clear application in view, namely the design of a portable natural language interface to computer databases. However, from the early days of the project the designers have claimed that they were also developing a cognitive model (e.g. Nelimarkka et al. 1984a: 168; Lehtola et al. 1985: 106). Not everyone shares this view. For exam-

ple, Starosta and Nomura (1986: 127) describe the Kielikone parser as having "evolved from the computational rather than the linguistic direction". If the claim that the Kielikone parser is a cognitive model is to be taken seriously it must be backed up by argumentation and evidence. At the moment this is conspicuous by its absence.

Modularization

As they stand, the parser and the grammar are almost distinct — but not quite.

To begin with the grammar, the functional schemata contain **Up** and **Down** slots which can be interpreted as control statements. They also contain heuristic **TryLeft** and **TryRight** slots whose sole purpose is to reduce the amount of search required of the parser. Jäppinen et al. (1988b) have recently proposed an optimization of the parsing algorithm which clearly removes the boundary between grammar and parser. They do this by introducing an ordered set of constituent types to look for (in much the same manner as Starosta and Nomura 1986):

> The basic left-corner-up algorithm can be modified so that it hierarchically first builds nominal LGT's [Locally Governed Trees] without prepositional modifiers, then LGT's governed by prepositions and postpositions, then nominal LGT's with postpositional modifying nominal LGT's, and finally the LGT governed by the finite verb (Jäppinen et al. 1988b: 277).

Division of labour

One of the outstanding questions surrounding the Kielikone parser is why there is a distinction between functional schemata and binary relations. This might be restated more succinctly by asking why the notion of 'constituent' has been retained at all. In the present system, the binary relations are concerned with the kind of simple pairwise relations familiar from dependency grammar whereas the functional schemata are concerned with larger objects which can

be identified with constituents in the traditional sense. In fact, a schema acts just like an $\bar{X}$ immediate dominance (ID) rule.

In criticizing the Kielikone approach, Kettunen claims that:

> It seems evident that the lexicon should be working more actively in a dependency parser. In FUNDPL this is not the case. As such, FUNDPL is not modelling dependency grammar properly (Kettunen 1989).

This seems like a harsh criticism with which to conclude this examination of the Kielikone parser. However, the Kielikone researchers have left themselves open to criticism. Although they have been prolific in their output, it has consisted almost exclusively of descriptions of the systems they have built, and, as has been noted above, these have not always been readily interpretable. There has been hardly any real discussion of motives for choices or arguments against possible alternatives. Parsers are notoriously difficult to compare and evaluate. Bald performance figures are not very helpful. What is required is a clear statement of the decisions which the parser embodies and some strong arguments for these decisions.

6.4 Summary

The Kielikone parser works from left to right, bottom-up. With each input word it associates an active schema, i.e. a frame consisting of dependency slots and heuristic information. Search proceeds from heads to dependents in a single pass through the sentence.

The parser is based on a blackboard architecture. While the basic idea of the parser is fairly clear, my attempts to reconstruct the algorithm on the basis of published accounts have not met with complete success.

The main features of the Kielikone parser are summarized in Table 6.1.

Table 6.1: main features of the Kielikone dependency parser

Search origin	bottom-up
Search manner	depth-first
Search order	left to right
Number of passes	one
Search focus	heads seek dependents
Ambiguity management	chronological backtracking (heuristics guide search)

Chapter 7

The DLT MT system

7.1 Overview

In this chapter I examine the Distributed Language Translation (DLT) systems produced by Buro voor Systeemontwikkeling ('BSO/Research'[1]).

I begin with an overview of the DLT system. In Section 7.2 I consider in more detail the DLT DG formalism. In Section 7.3 I review the approach to parsing adopted in the first system prototype. In Section 7.4 I consider the more radical solution suggested for the second prototype: a probabilistic dependency parser.

The DLT project is a large MT project jointly funded by BSO/Research and the Dutch Ministry of Economic Affairs. It began in late 1984 and, so far, 50 person-years have been invested in it. The aim of the project is to construct a semi-automatic MT system. The precise meaning of the designation 'semi-automatic' will become clear shortly. Unlike some of the other projects described here, there is an abundance of published material describing the DLT system, including a six-volume book series published by Foris and devoted entirely to DLT. For present purposes the most interesting of these are Schubert (1987) and Maxwell and Schubert (1989).

An important design consideration was the need to give the system a powerful language-neutral inference engine which could be simply customized for any language pair. The effort involved in constructing an MT system is much

[1]Since 1 July 1990 BSO/Research has been known as 'BSO/Language Systems'.

too great to risk having to re-build the whole system every time a new language is added. The design adopted in DLT 'distributes' the translation task into two sub-tasks. Firstly, the source language is translated into an intermediate representation. Secondly, the intermediate representation is translated into the target language. This is not obviously a simplification since where there might have been a single language pair, there are now two. The rationale for this approach is that all that is required in order to add a new language to the system is to write a sub-system for translating between that language and the intermediate representation. Once this has been done, it is possible to translate from the newly added language to all of the other languages in the system without further effort. Thus, if there are ten languages in the system and a new language is to be added, this necessitates the development of a translator for one language pair instead of ten language pairs. The intermediate representation used in the DLT system is a slightly modified version of Esperanto(!). In the early prototypes English is the source language and French is the target.

Translation is semi-automatic in DLT in the sense that the system can seek clarification from the user when necessary. When there are no difficulties, the system can translate from source to intermediate to target as though operating in batch mode. When a problem arises in translating from source language to intermediate representation, the system can query the user (in the source language). For example, if a source sentence is ambiguous, the system is able to resolve the ambiguity by asking the user to select amongst alternative readings.

The syntactic framework used in the DLT system is a version of DG. I shall describe it in more detail in the next section. DLT is controversial in its failure to construct explicit meaning representations for the sentences to be translated. Most MT systems first construct a semantic analysis of the source sentence and then use it to generate a sentence in the target language. DLT workers have argued that this sort of content-oriented approach is a kind of 'analytic overkill'. In trying to make the semantics explicit, a lot of problems

are raised which then have to be solved. Instead, they argue for an approach to translation which focuses on *form* rather than content. Schubert writes:

> There are a good deal of form correspondences, short cuts from form to form, which can and should be used. These correspondences are mostly not found in the directly visible syntactic form of texts, but at the next level of abstraction, the level of syntactic functions that are inferrable from syntactic form (Schubert 1987: 202).

In order to effect the mapping from syntactic structures of one language to syntactic structures of another language, a higher-level, dual language 'contrastive syntax' is required. The name by which this contrastive syntax is known is *metataxis*, from Tesnière's term 'métataxe'.[2]

> Metataxis...is a process which starts with syntactically analysed source language texts as the input and results in a synthesis of syntactically correct texts in a target language (Schubert 1987: 125).

It is claimed that a metataxis approach to MT does not make *no* use of semantics, but rather that the semantic information is used *implicitly*.

> In a metataxis-oriented semantic transfer process, it is possible to keep deep cases implicit and use semantic relators that are rather straightforwardly inferrable from syntactic functions (op. cit.: 203).

I shall not investigate this claim here. (For more information see Sadler 1989c.) Instead, I shall focus on the way in which DG is used to represent sentence structure and the way in which that structure is built by a parser.

The DLT system is summarized in Figure 7.1, which is based on a diagram in Witkam (1989: 142).

[2] "La traduction d'une langue à l'autre oblige à faire appel à une structure différente. Nous donnerons à ce changement structural le nom de **métataxe**" (Tesnière 1959: 283).

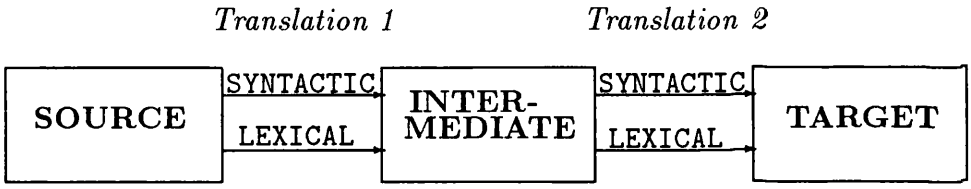


Figure 7.1: the Distributed Language Translation system

7.2 Dependency grammar in DLT

Although the DLT system has been well-publicized, my discussion of the version of DG on which it is based will be hampered by the fact that I have not been able to find any published account of the form of dependency rules adopted. The remarks in this section will accordingly be confined to a discussion of general constraints on well-formed sentences.

Many of the constraints on well-formedness are expressed in terms of tree geometry. In DLT, dependency structures are required to be 'true trees' rather than arbitrary graphs. That is, they must be rooted, directed, acyclic, and non-convergent.

Rooted The root of the tree represents the single independent element to which all other words in the sentence must be subordinate.

Directed The directedness of the arcs indicates the direction of the dependency relation holding between heads and dependents.

Acyclic The fact that the tree must be acyclic precludes the possibility of interdependence. Word A can not be head of word B in respect of one dependency relation and dependent of word B in respect of another dependency relation as this would lead to the presence of a cycle in the tree.

Non-convergent Links in the tree may never converge on a node. The effect of this is to prevent a word from depending on more than one other

> word or from depending on a single word by virtue of more than one dependency relation.

So far, this definition of well-formed dependency structures is entirely standard. Where it differs from the conventional model is in making no use of a projectivity or adjacency constraint. In terms of tree geometry, this would lead to crossing arcs, were it not for the fact that surface word order is not preserved in DLT dependency trees.[3]

> Dependency syntax does not rely on the contiguity principle. Word order may well play a role in syntactic form, but as soon as a word by means of its syntactic form has been assigned a dependency type label, syntactic form has fulfilled its function and need not be rendered in the tree. Dependency trees thus do not represent word order. They are not projective, at least not in the present model (op. cit.: 64).

The DLT dependency grammar de-couples word-order from dependency. This is illustrated in Figure 7.2 which shows the analysis for the sentence *Whom did you say it was given to?* (op. cit.: 103). (DLT dependency trees are usually represented as Tesnièrian stemmas. Arcs are labelled with the name of the type of dependency relation involved, although I have omitted labels here for the sake of readability.) Reading from right to left, notice that *you* precedes *did* (unlike in the sentence), and *whom* is in object position in the embedded sentence, rather than in its 'moved' sentence-initial position.

The arcs in a DLT tree represent dependency relations but what do the nodes represent? The simple answer is that most of the time they represent words, where 'word' is defined crudely in terms of a string of characters bounded by space characters. A node is never allowed to represent more than one word. Nodes are even prohibited from representing frozen multi-word foreign language borrowings such as *ipso facto.*

Although nodes signifying more than a single word are not allowed, a case is made for allowing nodes to signify *less* than one word, i.e. a morpheme. The

[3]Except in the form of features indicating the word's position in the input string.

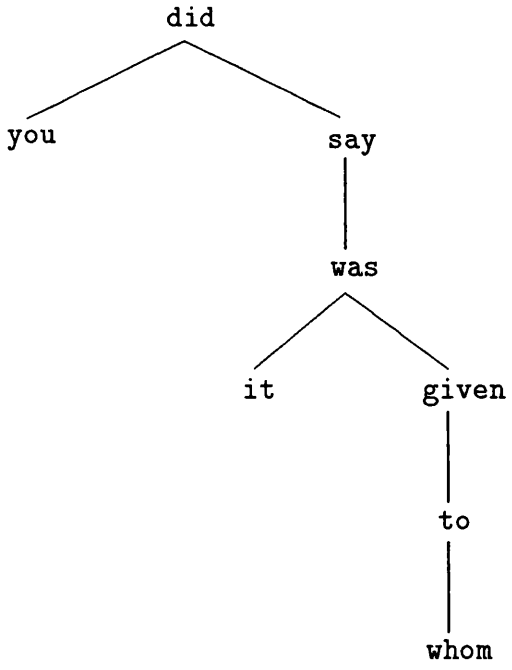


Figure 7.2: dependency analysis of the sentence *Whom did you say it was given to?*

arguments hinge around phenomena such as English clitics (*can't = can not*) and possessives (*Elizabeth's, the Queen of England's*) and the class of German verbs which combine a root with a participle in a single word in some contexts but which separate them into two words in other contexts. Thus, the root and the participle must be identified by different nodes in the dependency tree.

A more accurate characterization of the restriction on nodes is that they may only be used to represent morphemes, or morpheme strings smaller than or coextensive with the words in which they appear. It is necessary to allow morpheme strings to be represented by nodes since it would not be helpful to recognize a root word and its inflectional affix as separate nodes in the tree.

Things are not quite as simple as this, since the DLT grammar recognizes punctuation symbols as having a place in the structural analysis of sentences. For example, the period is used to mark the end of a sentence (van Zuijlen 1990) and the comma is used as a conjunction in coordinate struc-

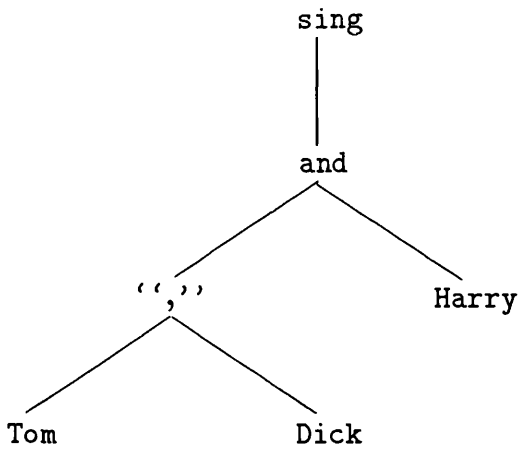


Figure 7.3: the use of *comma* in coordinate structure analyses

tures, such as the one shown in Figure 7.3 (Schubert 1987: 114ff; cf. Hellwig's use of punctuation in DG described on page 93 above).

7.3 An ATN for parsing dependencies

A number of parsing approaches have been considered in connection with the DLT project, most of them modifications of parsing techniques well tried with PSGs. In this section I shall briefly mention three of these — augmented PSG (APSG), definite clause grammar (DCG), and augmented transition network (ATN) grammar — which were briefly investigated during the development of the first DLT prototype.

In the early stages of the DLT project two parsers were developed for a subset of English in order to compare their computational efficiency. These were based on APSGs (Winograd 1983: 377ff) and ATNs (Woods 1970; Woods 1987). It appears that the ATN grammar performed best. I shall discuss it further below.

Schubert (1987: 213) argues that far from being tied to PSG, APSG is a general-purpose formalism for the description of trees which is "suited for dependency parsing as well." The APSG-based parser was imple-

mented in a parsing environment developed at the University of Amsterdam (van der Korst 1988). However, it stretches the meaning of 'dependency parser' somewhat to designate the APSG parser thus. Rather, it is a PSG parser which is able to map constituent structure onto dependency structure as it goes along. Its input is a PSG, not a DG. According to Korst the grammar contains 49 non-terminal categories and 27 lexical/punctuation categories (op. cit.: 6–7). I shall not consider the APSG parser any further here.

Schubert argues that DCGs (Pereira and Warren 1981) are not inherently inappropriate for expressing (or parsing) dependency relations. He continues:

> I am not aware of an implementation of DCGs involving dependency syntax, at least not for a complete syntax of a language. Within the DLT machine translation project, a small word parser has been implemented (by Job van Zuijlen) which builds up dependency trees for morphemes of complex Esperanto words (Schubert 1987: 214).

To the best of my knowledge no further research has been done towards developing a dependency version of DCG. Van Zuijlen's DCG morphological analyzer is reported in van Zuijlen (1986a, 1986b) .[4]

Turning to the ATN-version of DG, we find slightly more details in the literature. In fact, an ATN was used in the first DLT prototype which was completed in 1988. Schubert writes:

> For the DLT machine translation system, Witkam (1983: IV-87ff) designed an ATN for Esperanto, which is basically constituency-based and for which he had constituency trees in mind (Witkam 1983: IV-72ff). When dependency syntax was chosen for the DLT system, it was easy to equip this same ATN with tree-building actions for dependency trees (Schubert 1986: 11ff, 99ff). No rearrangements whatsoever were required in the ATN in order to shift from assumed constituency trees to dependency trees (op. cit.: 213).

[4] A very simple DCG for parsing sentences and constructing dependency trees can be found in the file `dcg.pl` in Appendix A.3. The file also includes a predicate `dcg_generate` which generates all strings and trees allowed by the grammar. The program in `map_to_dcg.pl` (also in Appendix A.3) can be used to map an arbitrary Gaifman grammar into an equivalent DCG.

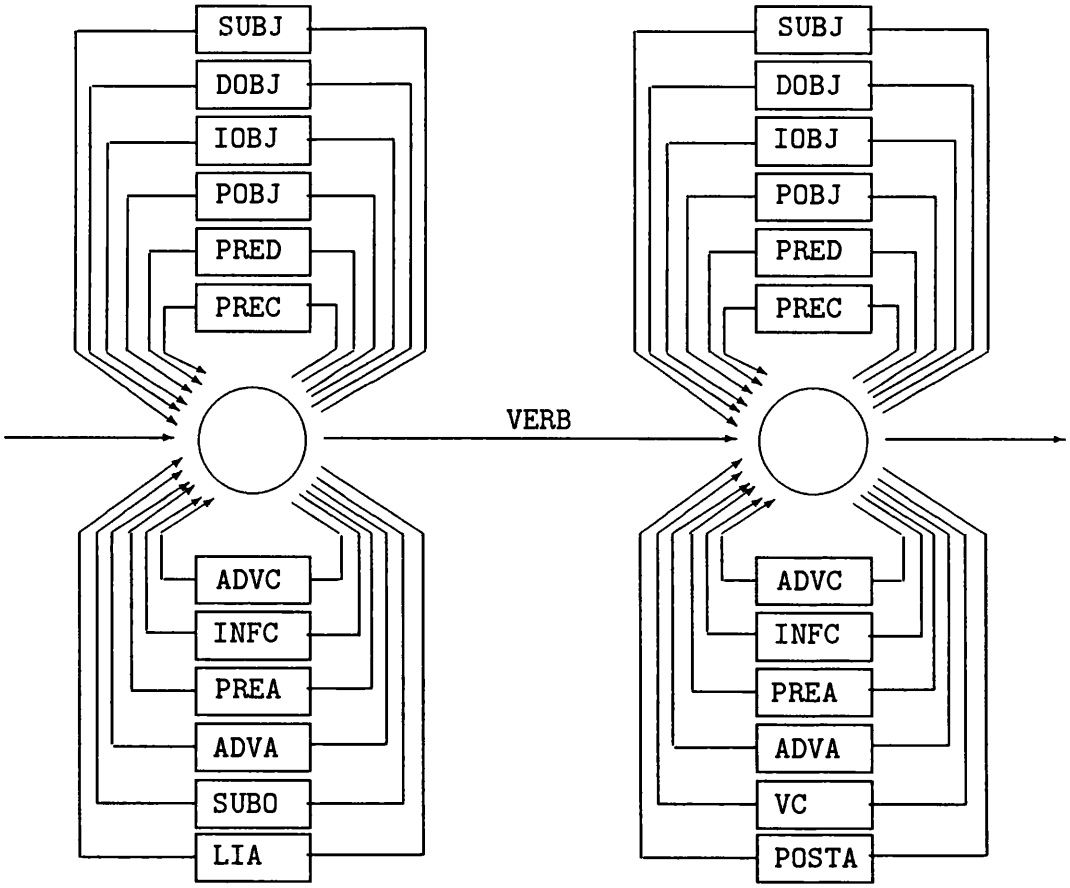


Figure 7.4: an ATN for parsing Danish sentences

ATNs are very simple and effective for parsing languages with an adjacency constraint (i.e. contiguous constituents) in terms of DG. The example network in Figure 7.4 is taken from Schubert (1987: 219). It shows the top level network for describing the structure of simple Danish sentences. Labelled boxes denote named networks; un-boxed labels on arcs indicate words to be consumed. Notice that there is considerable scope for variation of word order amongst the dependents of the verb. Registers are used to ensure that a verb has the correct number of dependents, e.g. that a verb has exactly one subject (as opposed to either any number of subjects or one before, and one after the verb). Figure 7.5 shows the separate SUBJECT network.

This dependency parser implements a top-down, left corner parse strategy.

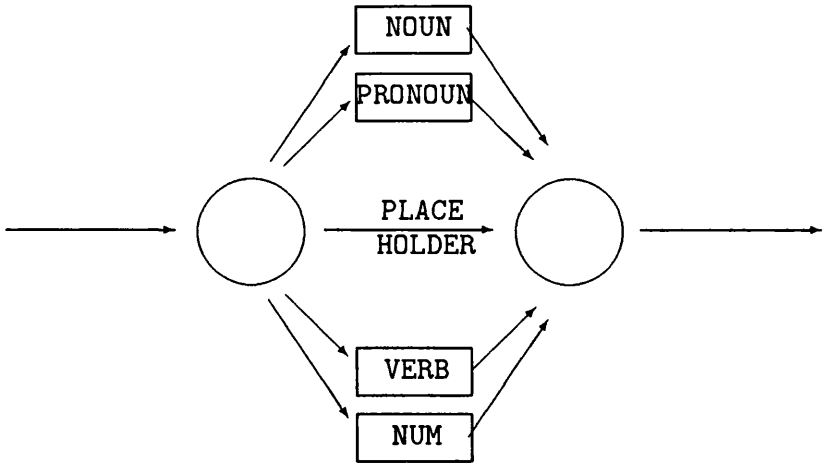


Figure 7.5: an ATN for parsing Danish subjects

ATNs impose an explicit search ordering, although in this case the relative order of the verb's dependents is fairly free. It could be argued that this works against one of DG's greatest assets, namely its orientation to *relationships* amongst words, rather than *sequencing* of words, which is what ATNs orient to.

As is normally the case with ATNs, the grammar and the parser are conflated. In fact, this is a *procedural grammar*. In line with the prevailing view in computer science and computational linguistics, I endorse the view that a clean separation should be maintained between grammars and parsers for reasons of clarity and modifiability (e.g. see Gazdar and Mellish 1989: 95ff). Presumably the same conclusions have been reached by the DLT team since they have now abandoned the use of ATNs.

7.4 A probabilistic dependency parser

For the second prototype of the DLT system, a completely different approach to parsing has been adopted. In the earlier prototype, fairly standard rule-governed parsers were tried. For the second prototype, experiments are being carried out with probabilistic parsing methods.

Probabilities can be incorporated into grammars in at least two ways. First, grammar rules can be augmented with probabilities reflecting the probability of each rule actually being used in a context in which it could be used. For example, the following notation could be used to indicate that the rule n(det,*) is appropriate for 60% of all nouns and the rule n(det,adj,*) for 20% of all nouns.

$$Pr(n(det, *)) = 0.6$$

$$Pr(n(det, adj, *)) = 0.2$$

This information can be used heuristically during parsing so that the rule with highest probability is tried first. Alternatively, all possible rules can be tried and all possible analyses built for a sentence. The analysis with the highest probability (calculated from the joint probabilities of all the rules used) is selected. In this way probabilities are used to choose amongst analyses in a language whose boundaries are fixed.

The second way in which probabilities can be built into a grammar dispenses with the dichotomy between well-formedness and ill-formedness, replacing it instead with a grammaticality continuum ranging from fairly ill-formed constructions through very well-formed constructions. In this approach the core rules of the grammar may be assigned probabilities in the fashion shown above. Additionally, all other rules possible within the logic of the grammatical framework may be allowed with very low probability. For example, the APRIL ('Annealing Parser for Realistic Input Language'; Haigh et al. 1988) and RAP ('Realistic Annealing Parser'; Atwell et al. 1989) projects use the technique of simulated annealing to reduce the amount of search required in order to parse with a grammar which does not rule out any structural possibilities *a priori*, instead assigning very low probabilities to all tree configurations not attested in the corpus. The object of this approach is to ensure that an analysis of

some kind is produced for every sentence, including those which conventional parsers would simply reject as ungrammatical.

It is normal for the probabilities attached to rules to be derived from empirical studies of text corpora. A corpus is first parsed and the analyses verified. The frequency of application of each rule is counted and then used to compute the probability of each rule. These probabilities are then projected from the 'training' corpus to the rest of the language. A rationale for allowing all logically possible rules with very low probability is that no training corpus will ever be large enough to furnish examples of the use of all rules of a natural language. By allowing every logically possible rule with very low probability it may be possible to make a parser robust enough to produce reasonable analyses, even for structures not attested in the training corpus.

As far as I am aware, Job van Zuijlen of BSO/Research is the first person to implement a probabilistic dependency parser. While he has investigated the theoretical possibilities of using simulated annealing in dependency parsers (van Zuijlen 1989a, 1989b), he has in practice adopted a more straightforward approach in the probabilistic dependency parser he has actually implemented (van Zuijlen 1990). First of all it was necessary to obtain a syntactically analyzed corpus in order to compile a set of rule probabilities. The *Bilingual Knowledge Bank* (BKB) is a corpus-based knowledge source which has come to be regarded as the heart of the DLT system (Sadler 1989a; Sadler 1989b). Put simply, it consists of a fully analyzed text in one language and the same text fully analyzed in another language.[5] This can then be treated as a resource for working out correspondences between the languages. Since the analysis of a language in the BKB includes preferred (hand-constructed) parse trees, it can be used to generate rules and associated probabilities of occurrence. (For the purposes of probabilistic parsing the fact that it is a *bilingual* knowledge

[5]According to van Zuijlen (personal communication) a simple rule-based dependency parser and a graphical tree editor were used to assist the human analyzer. I have no further information on the rule-based parser.

Table 7.1: different dependency links retrieved from the BKB

Word	Links
you	17
can	10
remove	4
the	9
document	58
from	16
drawer	37
	—
	151

base is of no interest: only one language is examined). For his first probabilistic parsing experiment (January to April 1990), van Zuijlen used a BKB tree bank consisting of 1400 dependency trees, representing some 22000 words from a software manual. (This is far too small a tree bank to have any significance outside of an exploratory experiment.) Corpus-based probabilistic parsing proceeds in four stages which are identified as Retrieval, Construction, Generation, and Evaluation.

Retrieval For each word in the input sentence, the corpus is searched. All of the occurrences of the word in the corpus are identified and a record is kept of all the different pairwise dependency relations in which the word-instances in the corpus participate. For example, the number of different dependency links retrieved for the input sentence *You can remove the document from the drawer* is shown in Table 7.1 (all examples from van Zuijlen 1990).

In addition to the information regarding the separate dependency links which point towards and away from the word instances, a tally is also kept of the patterning of these links with each other. Thus, a record of the individual links and the collective patterns is assembled.

Construction A network is constructed by finding pairs of links which 'fit together'. Intuitively, these links are descriptions of the same relation from

different perspectives, the head perspective and the governor perspective. More formally, a link can be added in the network if:

1. the governor label of the head link corresponds to the dependent label of the governor link,
2. the dependent link should be present in one or more of the dependency patterns of the governor, and
3. the position of the governor should agree with the direction of the dependent link.

The network produced for the test sentence *You can remove the document from the drawer* is represented in Figure 7.6. Dependency links are portrayed as connected rectangles. Solid rectangles identify dependents, dashed rectangles identify heads. The arrow points from dependent to head. Note that of the original 151 links found in the corpus, only 19 have fulfilled the construction conditions for inclusion in the network.

Generation In the generation phase the network is processed to remove links which do not form part of any possible coherent parse tree which has a single root to which all other words are subordinate. The removal of impossible links from the network in Figure 7.6 leaves 13 links remaining in the network. (These generate four different trees.)

Van Zuijlen has developed a method for representing multiple dependency trees in a single graph with structure-sharing (van Zuijlen 1988). However, its complexity is such that it can not be described here.

Evaluation Associated with each link in the network is a pair of numerical values. The *weight* of a link is an indication of how well a dependent fits in the dependency pattern of its governor, taking the governor's other dependents into account. The *suitability* of a link is an indication of how well a particular word is suited to having a specific function with respect to a particular governor. The

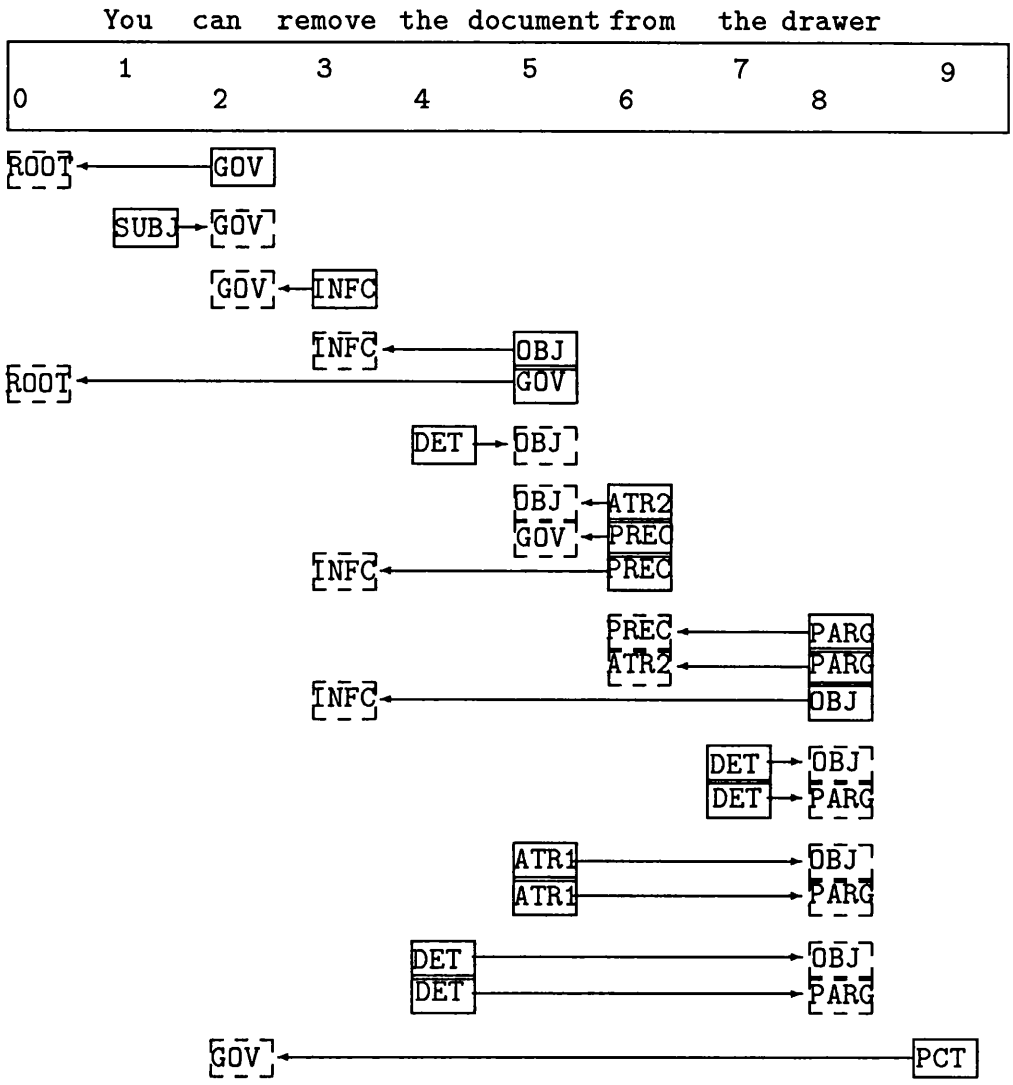


Figure 7.6: a dependency link network for the sentence *You can remove the document from the drawer*

weight and the suitability measures are merged in an adjustable proportion to yield the *quality* of the analysis represented by a given tree. In this way the alternative readings for the sentence can be compared and a 'best analysis' can be selected. I shall not explore the mathematics of the 'best analysis' selection method here.[6]

This parser represents an interesting innovation in both the fields of dependency parsing and probabilistic parsing. The association of probabilities with pairwise dependencies is, to the best of my knowledge, without precedent. It will be very interesting to watch this research develop and to see what the performance of the parser turns out to be when it has a reasonably large corpus to operate on. In the meantime judgement must be reserved on it until more results become available. Because of the extent to which this parser differs from the others in this thesis, detailed comparisons are difficult to make. I shall refrain from presenting a more formal PARS version of the algorithm or a worked example.

7.5 Summary

This chapter has presented an overview of the Distributed Language Translation MT project which is based on the idea of metataxis or contrastive syntax. I have shown how the functional structures represented by dependency trees provide a starting point for the process of metataxis. I have briefly noted the existence of small experimental quasi-dependency parsers based on augmented PSG and definite clause grammar. I have looked in more detail at a dependency parser which is no more than a slight modification to a conventional augmented transition network. This implements a top-down, left-to-right parsing strategy. Probabilities are used to decide the 'best' analysis when more than one is possible. However, the binary distinction between well-formedness and

[6] van Zuijlen (personal communication) says "In future work I hope to include incremental evaluation in order to control the size of the solution space during parsing" (original emphasis).

Table 7.2: main features of the DLT ATN dependency parser

Search origin	top-down
Search manner	depth-first
Search order	left to right
Number of passes	one
Search focus	network navigation
Ambiguity management	first parse only

Table 7.3: main features of the DLT probabilistic dependency parser

Search origin	bottom-up
Search manner	breadth-first
Search order	unspecified, unimportant
Number of passes	one
Search focus	heads and dependents seek each other simultaneously
Ambiguity management	highest-scoring parse selected

ill-formedness is strictly maintained.

The main features of the DLT ATN dependency parser are summarized in Table 7.2.

The latest parser to be developed in the project is much more radical, being based on the use of rules and probabilities generated 'on the fly' from a hand-analyzed corpus. The parser mixes bottom-up and top-down search: the actual words of the sentence are used to construct a grammar which thereafter guides search. Direction of processing is not crucial to the parser's control strategy (i.e. there is nothing inherently left-to-right or right-to-left about it). Rather, the parser begins by constructing as many minimal islands (i.e. word pairs) as it can and then rules out those which are not consistent with a coherent analysis or with what is known about the co-occurrence of dependency links.

The main features of the DLT probabilistic parser are summarized in Table 7.3.

Chapter 8

Lexicase parsers

8.1 Overview

Lexicase (Starosta 1988) is a grammatical theory developed by Stanley Starosta and his graduate students at the University of Hawaii over the last two decades. It is unique in contemporary linguistic theory for a number of reasons. First, it is old. The version of the theory in use today can be traced back to a class handout produced by Starosta in 1970 (Starosta 1970). To this a number of papers were soon added (e.g. Starosta 1971a, 1971b). No other theory of natural language mentioned in this thesis has remained so stable for such a long time.[1] Second, the theory has been widely field-tested. Lexicase grammars have been written for significant parts of around fifty different languages including many so-called 'exotic' (i.e. not Indo-European) languages. Apparently the theory's longevity does not stem from the sort of disregard for the hard facts of language of which some theories are occasionally accused. A third fact which distinguishes Lexicase from its rivals is that the theory has been all but ignored in the linguistic mainstream. On first inspection it seems strange that a theory which has been in existence for so long and which can draw on such an impressive body of descriptive material should receive so little critical attention. If the theory were worthless it ought to have been exposed as such; if it were outstanding it ought to have been

[1]This may be interpreted positively as evidence of the theory's proximity to the truth, or negatively as evidence of the fact that the theory has not been subject to the critical attention of the wider linguistics community.

praised. Neither of these things has happened to any great extent. Instead, it has been largely ignored. This may be due in part to the fact that the first book-length introduction to Lexicase theory only became available fairly recently (Starosta 1988). (At present the Lexicase literature runs to some 130 items.) The fact that this introductory volume has received some positive reviews (e.g. Blake 1989; Fraser 1989b; Miller 1990) may signal the awakening of interest in Lexicase (but see Turner 1990 for a searing attack on the same volume). Certainly, some of the main features which have distinguished Lexicase from other theories for most of its existence — its lexicalism, its recognition of head/dependent asymmetries, its extensive use of features — now form part of the tool chest of mainstream linguistics.

I shall not attempt to evaluate Lexicase theory here. Rather, I shall sketch the main points of the theory and examine two parsing algorithms developed for use with Lexicase grammars. Section 8.2 provides an overview of Lexicase theory. Section 8.3 describes the two Lexicase parsing algorithms.

8.2 Lexicase theory

Starosta describes Lexicase as a "panlexicalist monostratal dependency variety of generative localistic case grammar" (Starosta 1988: 1). It is *panlexicalist* in the sense of Hudson (1981a), i.e. the rules of the grammar are lexical rules, expressing relations among lexical items and features within lexical entries. Larger structures are seen as sequences of words linked by dependency relations. Lexicase is *monostratal* in that it accounts for the systematic relationships among words in sentences by means of lexical rules rather than syntactic transformations. The grammar refers to only one level of representation — the surface level. This is a feature which Lexicase shares with most dependency-based theories of language (for notable exceptions see Robinson 1970; Anderson 1977; Sgall et al. 1986; Mel'čuk 1988). Dependency in Lexicase will be described in more detail in the next section. Lexicase is

generative in the traditional Chomskyan sense — the rules and representations are expressed formally and explicitly and are concerned with a speaker-hearer's linguistic competence. Lexicase is a *case* grammar in the Fillmorean tradition (Fillmore 1968); every nominal constituent is analysed as bearing a syntactic-semantic relation to its regent. However, it has evolved away from mainstream case approaches in a number of respects. It is *localistic* (Hjelmslev 1935; Hjelmslev 1937; Anderson 1971), that is, it places strong emphasis on the use of spatially oriented semantic features. Whereas most case grammars are primarily concerned with situations and 'deep' analyses (e.g. Fillmore 1968; Schank 1975), Lexicase tends towards identifying case relations with syntactic relations (in this it accords with Anderson's case grammar (Anderson 1971)). Other distinctive features of case in Lexicase are the feature-based formalization and the requirement that every verb contain a Patient in its case frame (the so-called *Patient Centrality hypothesis* (Starosta 1978)).

8.2.1 Dependency in Lexicase

Starosta presents his dependency system as a highly constrained version of $\bar{X}$ theory. However, he introduces a number of constraints on the form of his $\bar{X}$ grammar, namely:

1. the lexical leaf constraint;
2. the optionality constraint;
3. the one-bar constraint;
4. the sisterhead constraint; and
5. the features on lexical items constraint.

Before examining these constraints, it is worth noting that very few discussions of $\bar{X}$ theory make clear exactly how constrained an $\bar{X}$ system needs to be. There are, of course, many possible instantiations of $\bar{X}$ grammar (Kornai and Pullum 1990), only one of which could be said to be equivalent to Starosta's DG.

The lexical leaf constraint

The lexical leaf constraint ensures that all terminal nodes are words. Throughout the years that PSG has been used by linguists, terminal nodes have been used to represent a number of different things besides words, for example morphemes and dummy symbols. GB theory (Chomsky 1981) allows empty categories such as *PRO* and *t* and sub-lexical morphemes such as *Tense* and *AGR*. Amongst dependency grammarians the same sort of non-word nodes have been introduced into dependency trees. For example, Robinson proposes a sub-lexical *T* (tense) morpheme (Robinson 1970) and Anderson advocates a phonetically null Ø node (Anderson 1971: 43).

It is hard to over-emphasize the importance of the lexical leaf constraint in Lexicase. It makes explicit the distinction between morphology and syntax: the associated claim is that the morphological structure of words is irrelevant to syntax. It rules out 'empty category' analyses and the possibility of handling 'movement' by associating moved items with 'gaps'. Starosta sums up the effect of this constraint as follows:

> The Lexicase representation thus sticks quite close to the lexical ground, accepting as possible grammatical statements only those which can be predicated of the actual strings of lexical items which constitute the atoms of the sentence. This constraint plus [the other constraints] limit the class of possible grammars by excluding otherwise plausible analyses and deciding on equally plausible analyses formulatable within the constrained Lexicase framework (Starosta 1988: 13).

The analysis in Figure 8.1 is prohibited in Lexicase. The lexical leaf constraint requires this sentence to be analysed in a tree structure with exactly three leaf nodes corresponding to the three words in the sentence; the structure in Figure 8.2 is closer to the Lexicase analysis. We shall see the actual form of a Lexicase tree for this sentence once we have examined the rest of the constraints.

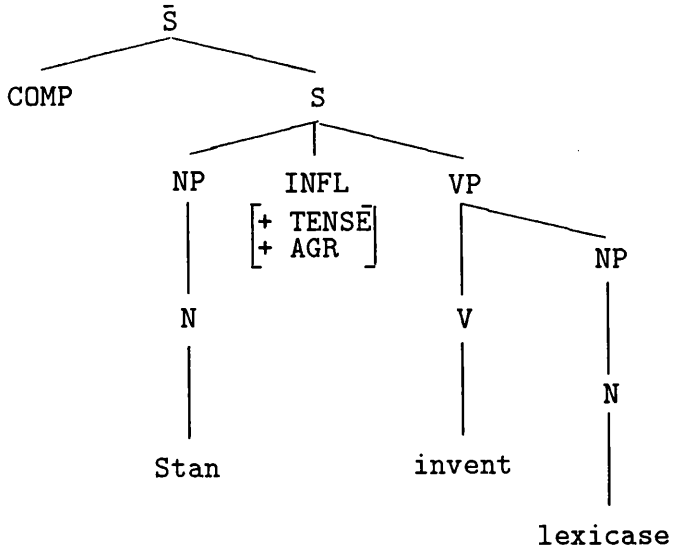


Figure 8.1: a syntactic structure with empty nodes

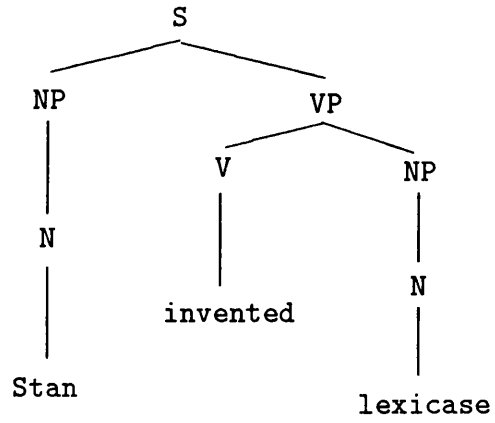


Figure 8.2: a syntactic structure without empty nodes

The optionality constraint

The optionality constraint states that every non-head daughter in a rule is optional. This is the standard understanding of 'optionality' as embodied in $\bar{X}$ PSG (Emonds 1976: 16; Jackendoff 1977: 36; Kornai and Pullum 1990). Notice that this does not exclude the possibility of a phrase containing more than one obligatory element; indeed, this is the normal case in exocentric constructions such as prepositional phrases. Starosta argues that "unlike conventional versions of dependency grammar... Lexicase does not require that every construction have a single head" (Starosta 1988: 12). This is misleading: conventional versions of DG do not require that every *construction* have a single head; rather, they require that every *dependent* have a single head. The notion 'head of a construction' is at best derivative in many dependency theories . However, Lexicase retains the idea of the construction or phrase (although it is not clear what work it does. Most constructions are endocentric and have a single head. The rest are exocentric and contain at least two *coheads*, exactly one of which is the lexical head and the rest of which are phrasal heads. Two kinds of exocentric construction are recognized, namely prepositional phrases and coordinate constructions. In prepositional phrases the preposition is the lexical head and the noun is the phrasal head. In coordinate constructions the lexical head is the conjunction and the conjuncts are each phrasal heads.[2]

It is important to understand Starosta's use of the terms 'endocentric', 'exocentric', and 'head'. In Bloomfield's seminal discussion of the endocentric/exocentric distinction (Bloomfield 1933: 194–7) his definitions rested upon the substitutability of one word in a construction for the construction as a whole. The distribution of *poor John* and *John* is identical so *John* is the head of an endocentric construction. Neither *in* nor *Wales* has the same distribution as *in Wales* so the construction is exocentric. By Bloomfield's definition, coordinate constructions are endocentric since *fish*, *chips*, and *fish and*

[2]Conjuncts may, of course, be realized as single lexical items.

chips all have the same distribution. However, by Starosta's optionality-based conjunction-as-head definition, coordinate constructions are exocentric. It is clear that when Starosta refers to a 'head' he is referring to a relationship which holds between a word (or words) and a whole construction. He reserves the term 'regent' to describe a word in relationship with a dependent word.[3] For example, in the sentence *I saw big bad John*, *John* is the regent of *big*; it is also the regent of *bad*; but it is the head of the whole phrase *big bad John*.

The one-bar constraint

The one-bar constraint states that "each and every construction (including the sentence) has at least one immediate lexical head, and every terminal node is the head of its own construction" (Starosta 1988: 14). This has the effect of guaranteeing that only single-bar phrases are possible nodes in a Lexicase tree. Every terminal node has its own one-bar projection and every non-terminal node is an $\bar{X}$ which is a maximal projection of its head X.

The most important consequence of the one-bar constraint is that it is no longer possible to analyse a sentence as consisting of an NP followed by a VP. Rather, a sentence is analysed as a $\bar{V}$ and the subject can be analysed as both a sister and a dependent of the main verb. Starosta argues that the absence of a VP removes the need to introduce an abstract *INFL* node to do to the subject what the verb would have done if it were the head of the sentence.

The effect of the one-bar constraint on the sentence shown in Figures 8.1 and 8.2 is to reduce and simplify the range of possible structural analyses. The overall shape of the tree thus constrained would be similar to that shown in Figure 8.3.

[3]Starosta (personal communication) names the Kielikone project (described in Chapter 6) as his source for this usage.

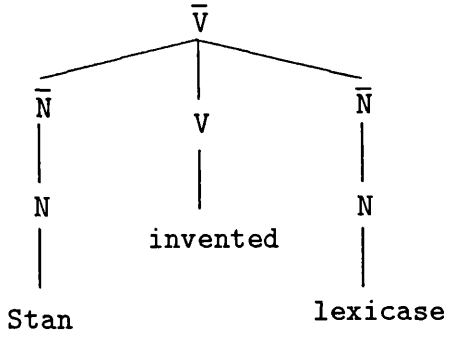


Figure 8.3: a syntactic structure constrained by the one-bar constraint

The sisterhead constraint

The sisterhead constraint states that "lexical items are subcategorized only by their dependent sisters" (Starosta 1988: 20). In other words, all grammatical relationships are statable in terms of regent-dependent pairings. Any word which depends directly or indirectly on X is said to be in the syntactic domain of X.

The relationship between regents and dependents is antisymmetric; regents are subcategorized by their dependents but dependents can not impose constraints on their regents. For example, a dependent could not require its regent to precede it.

The features on lexical items constraint

The 'features on lexical items constraint' states that "features are marked only on lexical items, not on non-terminal nodes" (Starosta 1988: 23). This constraint is the final step from a standard $\bar{X}$ grammar to a DG. If only the lexical items carry features, and lexical items are subcategorized by their dependent sisters, then clearly all the $\bar{X}$ structure is doing is relating lexical items pairwise. This can be clarified by simplifying the Lexicase tree representation further. Since every node in a tree is a one-bar projection of its head lexical item, node labels are predictable and therefore redundant. Thus the analysis

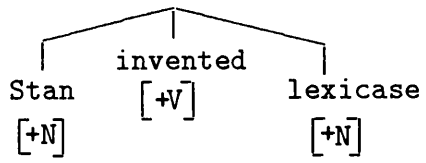


Figure 8.4: a Lexicase syntactic structure

of the sentence *Stan invented Lexicase* can be represented finally as shown in Figure 8.4.

This looks remarkably like a traditional dependency stemma except for the presence of a feature matrix attached to each word. In Figure 8.4 only the word class features have been shown. However, several different kinds of feature may appear in the lexical entry for a word. These are described in the next section.

8.2.2 Lexical entries in Lexicase

Associated with each word in the lexicon is a bundle of features. Features can be divided into contextual and non-contextual features. Non-contextual features are binary; a lexical item either has or has not got some property. The presence of property P is identified thus: [+P], its necessary absence thus: [–P]. Contextual features determine which words are dependent on which other words. They can be viewed as well-formedness conditions on the dependency trees associated with the words in a sentence. Contextual features can be positive, negative, or implicational. The following exemplify some uses to which features can be put:

(34)

a [+Det]
b [–fint]
c [–[+Det]]
d [+[+N]_]
e [+_[+N]]
f [⊃_[+N]]

Example (34a) is a positive non-contextual feature indicating that the word bearing the feature is a determiner. Example (34b) is a negative non-contextual feature indicating that the word bearing the feature is not finite. Example (34c) is a negative contextual feature indicating that the word bearing the feature does not have a dependent determiner (relative position unstated). Example (34d) is a positive contextual feature indicating that the word bearing the feature requires a preceding dependent noun. Example (34e) is a positive contextual feature indicating that the word bearing the feature requires a following dependent noun. Example (34f) is an implicational contextual feature indicating that the word bearing the feature is *expected* to have a following dependent noun. Under certain circumstances the expected word may be absent (for example, in the case of 'moved' wh-words). Double contextual features are prohibited. That is, a contextual feature may not be included within another contextual feature. An exhaustive listing of the formal properties of lexicase features can be found in Starosta (1988: 57).

If a Lexicase grammar were to consist solely of a number of lexical entries consisting of contextual and non-contextual features, then no useful generalizations would be made. However, Starosta takes the traditional view that a grammar should consist of a set of generalizations and a lexicon should be a repository for exceptions. It just happens that all grammatical rules in a Lexicase grammar are generalizations about lexical items. Accordingly, he sets up rules which are responsible for inserting all predictable features into lexical entries. These rules he divides into redundancy rules, subcategorization rules,, inflectional redundancy rules, morphological rules, derivation rules, semantic

interpretation rules, and phonological rules. This classification is purely a descriptive convenience. Each type of rule has the same basic operation: if a set of conditions is met by a word (the left hand side of the rule) then a set of features is added to the feature matrix of that word.[4]

We shall briefly consider the range of features utilized within Lexicase. There are five basic types:

1. syntactic category features;
2. inflectional features;
3. semantic features;
4. case relations; and
5. case forms.

Syntactic category features

Syntactic categories are atomic. They can not be defined, for example, as [+N,+V]. Major syntactic categories are drawn from a very small inventory which contains the following items: noun (N), verb (V), adverb (Adv), preposition or postposition (P), sentence particle (SPart), adjective (Adj), determiner (Det), and conjunction (Cnjn). These major categories are divided into distributional subcategories (e.g. subcategories of N include pronoun and proper noun) and this subclassification is indicated by the addition of extra features (e.g. [+prnn], [+prpn]).

As will become clear in the following discussion, syntactic category features play a very important part in the functioning of a Lexicase parser.

Inflectional features

Inflectional features correspond to the traditional inflectional categories of person, number, gender, case, tense, etc. These features have a central role to play in agreement so they are also important in parsing.

[4]Starosta's most recent work seems to suggest that there may be some slight formal differences amongst rule types (Starosta forthcoming).

Semantic features

Semantic features serve to distinguish words from each other. It is assumed that the grammar contains enough semantic features to distinguish every lexical item from every other (non-synonymous) item in respect of at least one distinctive semantic feature. In parsing, semantic features have the character of selectional restrictions. These restrictions are implicational rather than absolute. Thus, the verb *drink* might expect an object marked [+dkbl] (drinkable) but in the absence of such an object a metaphorical reading would be forced. This seems very close to the position adopted in Wilks' *Preference Semantics* (Wilks 1975).

Case relations

Lexicase assumes five 'deep' case relations, namely AGENT, PATIENT, LOCUS, CORRESPONDENT and MEANS. The *Patient Centrality Hypothesis* (Starosta 1978, 1988: 128ff) asserts that there is a PATIENT in the case frame of every verb, i.e. every sentence contains a PATIENT. The inventory of case relations is kept to only five since many of the distinctions typically made by case relations in other Fillmorean systems are made by the semantic features in Lexicase. Starosta and Nomura cryptically claim that:

> The...reduced non-redundant case relation inventory improves the efficiency of case related parsing procedures...It is necessary to refer to case relations in parsing structures containing multi-argument predicates, in accounting for anaphora and semantic scope phenomena and text coherence, and of course in translation (Starosta and Nomura 1986: 128).

Unfortunately there appear to be no published accounts of how these case relations should be used in the parser.

Case forms

Unlike the other features, case forms are not atomic. Rather, they are configurations of surface case markers such as case inflections, word order, pre- and post-positions, relator nouns, etc, which function to mark the presence of case relations. They are grouped together according to which case relations they identify and on the basis of shared localistic features. Case forms are composed of grammatical features such as 'nominative' or 'ergative' and localistic features such as 'source', 'goal', 'terminus', 'surface', 'association', etc. Starosta and Nomura claim that case forms are used by the parser to recognize the presence of particular case relations. They state that

> this means that in parsing, such information is obtainable directly by simply accessing the lexical entries of the case-markers rather than by more complex inference procedures needed to identify the presence of the more usual Fillmore-type case relations (ibid.).

Once again, this must be taken on trust as no documented examples are available.

At the conclusion of this overview of Lexicase, it may be observed that the theory makes use of dependency, although the variety of dependency adopted is defined in terms of a very highly constrained $\bar{X}$ system. It also makes use of many different kinds of features, representing many different things. A considerable number of pages could be devoted to exploring Lexicase's status as a case grammar but this would lead away from my primary objective of investigating dependency parsing. Starosta and his colleagues have yet to publish a detailed explanation of the place of case relations and forms in parsing so I shall not second-guess what might be intended. A more detailed and critical discussion of case in Lexicase can be found in *Valency and Case in Computational Linguistics* (Somers 1987), although many of the points made therein are disputed in Starosta's review of that monograph (Starosta 1990).

The next section investigates how some of the featural constraints of Lexicase are employed in parsing.

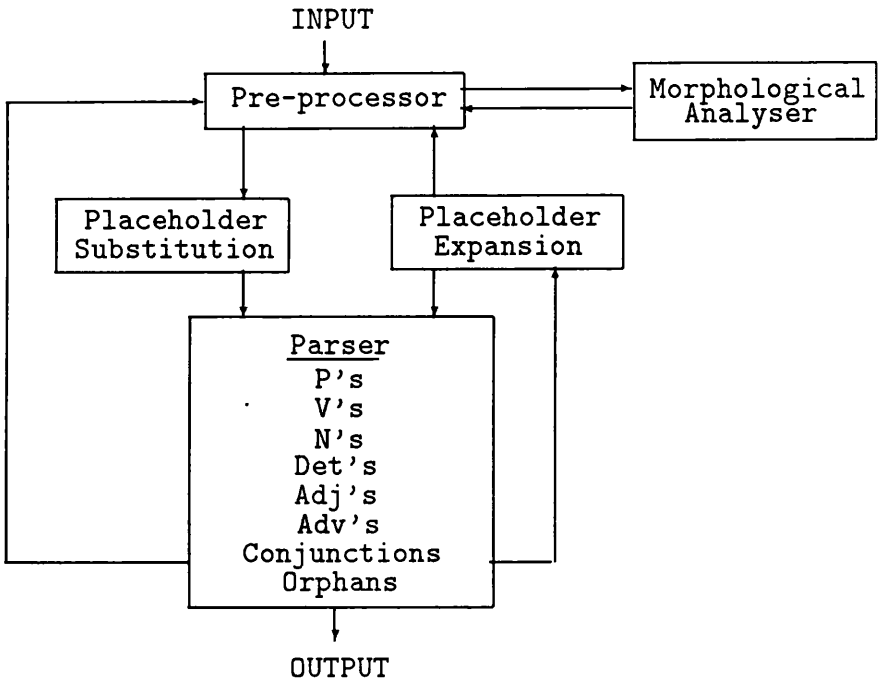


Figure 8.5: components of Starosta & Nomura's Lexicase parser

8.3 Lexicase parsing

In this section I examine two Lexicase parsers. The first, and better documented parser, was developed by Stanley Starosta and Hirosato Nomura (NTT Research Labs, Tokyo) and reported in *COLING '86*. The second is the product of Francis Lindsey Jr., a graduate student at the University of Hawaii. It is described in a short technical report.

8.3.1 Starosta and Nomura's parser

The principle reference for Starosta and Nomura's parser is Starosta and Nomura (1986).

Components

The overall architecture of the parser is shown in Figure 8.5.

The pre-processor The pre-processor replaces each word in the input sentence with a feature matrix, fully specified for all contextual and non-contextual syntactic and semantic features. If a word form in the input sentence could correspond to more than one feature matrix then the word is replaced with a 'cluster', a list of all the possible feature matrices. The output of the pre-processor is a string of feature matrices and clusters of feature matrices corresponding to the words of the input sentence.

Morphological analyzer The pre-processor is a basic look-up system which finds a word in the input sentence and looks it up in the grammar-lexicon. If the word can not be found then the morphological analyzer checks to see if the form matches any known stem-affix pattern. If a match is found, further searches are carried out with the stem to see if any other affixes produce homographic words. Once again, all of the possible feature matrices are stored together in a cluster.

Placeholder substitution Every cluster of feature matrices is temporarily replaced by a 'placeholder' which consists of the intersection of all feature matrices. If the only thing the feature matrices have in common is the word form then that is all the placeholder will consist of. The object of placeholder substitution is to minimize the amount of processing which has to be done. A parse can be produced for the unambiguous parts of the sentence and then, when it becomes necessary to try to integrate different readings for the ambiguous parts, the placeholder can be expanded and different possibilities tried without any need to reprocess common parts of the input.

Parser The parser uses the positive contextual syntactic features of head lexical items to search for dependents. These dependents must satisfy the criteria of the contextual features and they must be accessible. According to the definition of Lexicase, dependency relations (branches in a tree) are

not allowed to cross, i.e. Lexicase has an explicit adjacency constraint.[5] As soon as a potential link between words is established, the negative contextual features of the words are checked. If they are violated, the dependency link is discarded immediately. After each word pair has been linked by means of positive contextual features and checked and passed by negative contextual features, the implicational semantic contextual features (selectional restrictions) are checked. If the link violates the implicational features the analysis is not abandoned but it is marked as semantically anomalous.

Placeholder expansion Each string that contains a placeholder is expanded into separate structures by replacing the placeholder clusters with subclusters of items sharing more features in common. The resulting strings are passed through the parser once more to add links that become possible as the new clusters and entries become accessible. This process of placeholder expansion is repeated until all placeholders are eventually resolved into their original constituent entries. This ensures that all possible readings are obtained for a sentence without any sequences of words having to be reparsed.

Parsing algorithm

Clearly this is a multi-pass system. Pre-processing constitutes the first pass, morphological analysis the second, placeholder substitution the third and then some number of iterations through the parser/placeholder expansion cycle. In principle, there is no reason why pre-processing, morphological analysis and placeholder substitution should not take place incrementally from left to right. However, this would not buy anything extra since the parser's input is required to be a string of feature matrices and placeholders corresponding to the whole sentence.

The parser/placeholder expansion process is necessarily cyclic since the

[5]Since Lexicase is defined as a highly constrained $\bar{X}$ grammar, the adjacency constraint is basic and non-negotiable. In DGs which do not owe a debt to $\bar{X}$ grammar, the adjacency constraint is an optional extra. It can be used, not used, modified, or whatever.

effect of the interacting components is to maximize generalizations about the sentence and to proceed, iteratively, to all possible specific analyses. The process produces a maximally general analysis for the whole sentence, then it copies the analysis and adds different, more specific, details to each copy and then repeats the process for each copy. The process runs to completion for each candidate sentence. The effect of the parser/placeholder expansion cycle is similar to that of a chart parser in that it only builds structure once, no matter how many times it is used. However, this system lacks the elegance and simplicity of a chart parser's single pass through a word string. Even if there were some way for the Lexicase parser to construct a chart-like structure in a single pass in order to manage ambiguity, the parser is still required to pass through the word string many times for other reasons.

The parser sweeps through the word string *eight* times during each iteration of the parser/placeholder expansion cycle. This is because it tries to spot particular kinds of word on each pass. The passes are ordered as follows:

1. **Prepositions.** The parser attempts to link each preposition with an accessible N, V, or P by means of contextual features. The object of this pass is to link P's with their dependents to form PP's which delimit closed domains whose internal non-head constituents are then inaccessible to external heads or dependents. Subsequent passes may search inside or outside these phrasal domains but they need never consider any links between internal and external items. Recall that PP's are considered to be exocentric and that P's and N's have the status of *coheads.* When a P and an N are linked to form a PP, their non-contextual features combine to form a *virtual matrix* for that phrase. The features of both coheads thus become available to subcategorize the head of the phrase in which the PP is located.

2. **Verbs.** Verbs are linked to their dependents next to form 'sentences'. Once again, this has the effect of delimiting domains within which sub-

sequent linking may take place.

3. **Nouns.** Nouns are next to be linked to their dependents.

4. **Determiners.** Determiners are linked with accessible nouns next. It is not entirely clear why this phase exists in the parser since all determiners are dependents of nouns in Lexicase, so step 3 should already have linked them to their regent nouns. It must be assumed that what is going on is that determiners select their heads rather than *vice versa.* This is in direct contradiction of Starosta and Nomura's description of the operation of their parser: "Based on the positive contextual features of head lexical items, the heads are linked to eligible and accessible dependent items" (Starosta and Nomura 1986: 131). Whatever the status of steps 3 and 4 might be, their desired effect is obvious: in English determiners mark the left boundary of NP's and so linking them to their head nouns has the effect of closing off domains of government.

5. **Adjectives.** Link each adjective with an adjacent noun. The same points apply here as in step 4.

6. **Adverbs.** Link each adverb with a head verb or adjective. Once again the objections of steps 4 and 5 apply.

7. **Conjunctions.** Link each conjunction with one or more major constituents. Since most of the constituents will already have been discovered, the number of linking choices should be extremely limited. Since coordinate constructions are exocentric, the non-contextual features combine to form a virtual matrix for the whole construction.

8. **Orphanage.** Link all remaining free nouns, determiners, adjectives, adverbs, prepositions and verbs with an accessible head. All unattached lexical items will be found embedded inside other constructions and the attachment possibilities will be extremely limited. The exceptions are adverbs and PP's which tend to have more possible attachments available

to them.

Each of these passes through the sentence could take place in any direction but it makes sense to proceed from head to dependent. Therefore, passes could be expected to proceed from left-to-right in head first languages and from right-to-left in head second languages.

The presence of apparent contradictions in the published description of the parser, coupled with the general lack of published fine-grained detail, rule out the possibility of a more explicit PARS description of Starosta and Nomura's algorithm.

Given the algorithm described here, it would not be surprising to find that parsing a relatively short sentence involved something of the order of 100 passes through the sentence! No performance figures are supplied for the parser since it has never been implemented (although this is not made clear in any published description). The fact that multiple passes are required need not hav a negative effect on the efficiency of the parser, since the number of passes is fixed (i.e. independent of input length). However, the fact that the same string has to be processed time and again does beg several questions about the exact nature of the data structures used and the information represented. For example, if a subtree has been constructed somewhere in a string, does anything prevent the algorithm looking (pointlessly) at the corresponding substring in subsequent passes? Unfortunately, answers to important questions of this kind are not supplied in any published accounts.

A fundamental problem with this algorithm is that it does not maintain a distinction between grammar and parser. By building searches for specific kinds of lexical items into the parser, Starosta and Nomura have built in the assumptions (i) that all languages make use of the same inventory of word classes and (ii) that the appropriate order in which to analyse them remains constant between languages. The fact that the parser refers explicitly to things called 'nouns' and 'verbs' ensures that it will fail to work if it is presented with

a perfectly good grammar which happens to use different word class labels (such as 'a' and 'b') to identify nouns and verbs. In practice this objection could be mitigated if the algorithm were re-designed so that the parse order (e.g. P, V, N, Det, Adj, Adv, Conj, Orphans) was defined in a separate declarative database rather than being hard-wired into the algorithm. The parser would be invoked with two arguments: a pointer to the grammar to use and a pointer to the parse order definition to use. Any inconsistency between these two would lead to the result of the parse being not 'succeed' or 'fail', but 'error'.

8.3.2 Lindsey's parser

Lindsey's parser is simpler than Starosta and Nomura's but unfortunately even less information is available describing it. All of the information in this section has been gleaned from Lindsay (1987).

Lindsey's parsing system — known as 'FLX' — was written in Common LISP and runs on an HP-9000 Series 300 Bobcat workstation. It is based on Lexicase. The system consists of two primary components, the lexicon builder and the parser.

The lexicon builder

The lexicon builder takes as its input a Lexicase lexicon and a set of Lexicase rules. It uses the rules (which are, of course, statements of predictable information about lexical entries) to expand out the lexical entries to produce a fully specified, full form lexicon. In the case of homographic entries, a master entry containing as its matrix the intersection of the features shared by the matrices of all the words with the same form is created. A master entry would have the form shown in Figure 8.6.

```
(word (category shared features)
      (distribution shared features)
      (other shared features)
          (word1
                 (category features specific to word1)
                 (distribution features specific to word1)
                 (other features specific to word 1)
          (word2
                 (category features specific to word2)
                 (distribution features specific to word2)
                 (other features specific to word 2))))
```

Figure 8.6: a master entry showing the intersection of the feature sets of two homographic words

The parser

Once again, the parser examines the contextual features of a head word in order to establish dependencies. The parser assumes that each word is the head of a phrase and that a phrase is complete when all dependents of a word have been found. The parser proceeds as follows (quoted directly from Lindsey 1987: 3):

1. Find the entries for each word of the input sentence in the lexicon.
2. Eliminate from consideration all words which because of their position or word class could not be sisters of a head.
3. Determine which words must be sisters of a head because of the distributional requirements of the head.
4. For all words not unambiguously assigned as sisters to some head by the above steps, determine possible alternative assignments. This step provides for multiple parses.
5. Unpack master entries and determine which specific homonym successfully satisfies all distributional restrictions. This is done from top down, examining the parses given by step 4.
6. Print out those parses in which all words of the input sentence fit into one hierarchical structure.

Steps 1 through 4 set up a list of potentially successful parses which can then be examined as the basis of alternative sentence readings once the master entries have been unpacked. Thus, the algorithm allows the simple parts of the parse to be constructed and then reused in successive attempts to integrate alternative readings for words.

Unfortunately, the parsing algorithm is described in terms which are too terse to be really informative. The words "determine which words must be sisters of a head because of the distributional requirements of the head" are tantalizing in what they withhold rather than in what they tell. Lindsey's examples do not shed light on this process. However, it is clear that the parser is distinct from the grammar in this system. Lindsey writes:

> This complete parsing program is designed as a modular rule application system. The lexicon builder, given the appropriate minimally specified lexicon and Lexicase rules, may be used to create fully specified lexicons for any language... The parser is also a flexible program since it is nothing more than a program to determine possible (one-bar) dependency relationships between items in an input string on the basis of features associated with those items (Lindsey 1987: 4–5).

Thus, Lindsey's system is more flexible than Starosta and Nomura's but insufficient documentation is available to make a more informed comparison. It is not clear, for example, whether or not there is any loss of analytic accuracy on the part of the simpler system.

Insufficient information is available to construct a PARS description of Lindsey's algorithm.

8.4 Summary

In this chapter I have briefly reviewed the theory of Lexicase and two parsers which are based on it. It is clear that the theory is much better developed than the parsers based on the theory. The parsers do not make use of the full range of Lexicase resources, such as case relations and case forms.

Table 8.1: main features of Starosta and Nomura's Lexicase parser

Search origin	top-down
Search manner	breadth-first
Search order	left to right
Number of passes	at least eight
Search focus	heads seek dependents
Ambiguity management	packing/unpacking

Table 8.2: main features of Lindsey's Lexicase parser

Search origin	unspecified
Search manner	breadth-first
Search order	unspecified
Number of passes	multi-pass
Search focus	heads seek dependents
Ambiguity management	packing/unpacking

Starosta and Nomura's parser searches top-down for dependents for different classes of word on each of several passes. Unambiguous parts of the sentence are built first and then these 'common' parts are copied to different parse trees, one for each possible reading of the sentence. The main features of Starosta and Nomura's parser are summarized in Table 8.1.

Very few details are available for Lindsey's parser. It seems to share a number of properties with Starosta and Nomura's parser. For example, heads seek dependents, and ambiguity is managed by packing ambiguous words into clusters which can later be unpacked and tried in different parse trees. The main (known) features of Lindsey's parser are summarized in Table 8.2.

Chapter 9

Word Grammar parsers

9.1 Overview

In 1976 Richard Hudson published a monograph introducing his theory of 'Daughter Dependency Grammar' (DDG) (Hudson 1976). This publication was notable for two principal reasons. First, it argued that transformations were unnecessary in syntax — a heretical position in the linguistic climate of the day. Second, it argued that dependency as well as constituency should have a place in syntactic theory.

By the end of the 1970's Hudson was arguing against what he perceived to be an under-motivated and artificial distinction between grammar and lexicon in linguistic theories. Instead, he argued that all grammatical and lexical (and semantic) information should be stored in a single homogeneous representation within a single component — the so-called 'pan-lexicon' — which could be viewed as a body of facts about words (Hudson 1981a). Around this time he also published an important paper in *Linguistics* (Hudson 1980a) arguing that while dependency is necessary in syntax, constituency is not. Clearly, these two positions — the 'pan-lexicalist' and 'dependency only' positions — are compatible. In fact, the first implies the second since a collection of facts about words could not include facts about supra-word constituents. The second implies the first since a grammar without constituents leaves the word as the largest unit of analysis.

These ideas were molded into a coherent theory which came to be known as Word Grammar (WG) (Hudson 1983). The first monograph-length description of the theory appeared as Hudson (1984). Since the publication of that text there has been a major revision of the WG notation and a succession of papers describing WG treatments of various 'test case' constructions such as coordination (Hudson 1988a), extraction (Hudson 1988b), gapping (Hudson 1989b), and passives (Hudson 1989a). The state-of-the-art in WG is detailed in a recent monograph (Hudson 1990), which includes a grammar of a substantial fragment of English.

Section 9.2 introduces WG theory. Section 9.3 provides an overview of WG parsing, and presents WG parsers developed by myself and by Richard Hudson.

9.2 Word Grammar theory

9.2.1 Facts about words

A WG consists of a body of facts about words. In this section I describe the form that these facts take and the information they contain.

First of all, it is worth pointing out that 'word' in the context of WG includes any word-length unit, however specific or general. Thus, the first word of this sentence, the lexeme 'PLIMSOLL', the word-type 'noun' and the relation 'subject' are all words.

Each lexical entry is essentially a complex feature structure. As such it could be represented in a standard DAG format such as the one provided by PATR-II (Shieber 1986). However, Hudson has evolved his own metalanguage which has a quasi-English syntax and is often simpler to read than more familiar DAG structures.

A lexical entry is viewed as a collection of propositions. Each proposition has the general format

Argument1 Predicate Argument2

The predicate is placed in infix position rather than the more familiar prefix position

Predicate(Argument1, Argument2)

for the sake of readability. The chosen ordering is congruent with the normaal SVO order of English predicate-argument structure. However, nothing restts on the predicate-argument order of the notation. Any ordering would do sco long as it was used consistently.

Five predicates appear in WG propositions.[1] These predicates are thie following:

1. is
2. has
3. precedes
4. follows
5. isa

The predicate is

The is predicate is used to express identity between arguments. Thus

X is Y

identifies X and Y as being alternative names for the same object. An objecct can be identified in more than one way because of the facility for *relativve naming* in WG. In the sentence *Ollie obeyed Ronnie* shown in Figure 9.1, *Olliie* could be described either as 'word 1' or as 'the subject of word 2'.

Relative names are expressed in the form

(Name1 of Name2)

[1]It is possible to define a WG system which has only one predicate and which makes thhe necessary distinctions in terms of features (see Section 9.2.3 below). However, for the sakke of clarity of presentation I shall work with the five predicate system here.

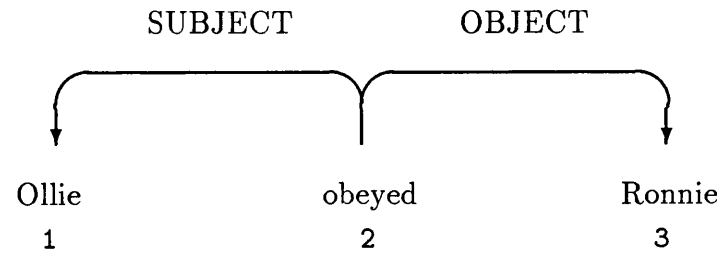


Figure 9.1: dependency structure of *Ollie obeyed Ronnie*

Where 'Name1' must be the atomic name of a relational concept (such as 'subject' or 'agent') and 'Name2' may be either the atomic name of a non-relational concept (such as 'noun' or 'word2') or another relative name. Thus, the following are both possible:

(35)

a (subject of word2)
b (agent of (referent of word2))

The identity predicate is can be used to unify sets of propositions (alternatively conceived of as feature structures) associated with labellings in the system. In this way categorial, functional, and semantic information can be combined in the property structure of a given word instance.

The predicate has

The has predicate is used for two main purposes. First, it is used to assign features to words. For example,

(36)

noun has (a number)

It should be obvious that values can be assigned to features using is propositions:

(37)

(number of wordN) is singular

The second use of the has proposition is in specifying the dependency requirements of a word. For example,

(38)

finite verb has (a subject)

Here the use of a quasi-English formalism is slightly misleading. The use of the predicate has in (38) does not express the fact that some particular finite verb is in possession of a subject. Rather, it expresses the fact that the prototypical finite verb has a subject.[2] Thus, it could be read as follows: 'A finite verb typically has a subject (slot)'.

WG has a mechanism for distinguishing optional and obligatory dependents, as well as for signaling a number of more subtle distinctions. The general format of 'slot' propositions is:

A has (Q B)

where A is some named entity, B is the name of a slot (e.g. 'subject') and Q is a 'quantitator'. A quantitator (Hudson's term) specifies the number of slots of the variety specified by B. To date, most of Hudson's writings have made use of the following set of quantitators:

(39)	a	a X	= one X required
	b	ano X	= at most one X allowed
	c	mano X	= any number of Xs allowed
	d	many X	= two or more Xs allowed
	e	mony X	= one or more Xs allowed
	f	no X	= X prohibited

The utility of these should be fairly obvious. (39a) is used when exactly one filler is required, as in the case of subjects. (39b) applies when a slot is optional but can never have more than one filler. For example, a noun can optionally have a dependent relative pronoun. (39c) is the least constrained — any number of fillers will suffice. For example, a noun can be modified by any number of adjectives. (39d) is used when at least two fillers are required. The principle

[2]Hudson intends his theory to be based on the notion of 'prototypes' (for useful introductions see Lakoff 1985 and Taylor 1989).

use for this is coordinate constructions where a conjunction must conjoin at least two conjuncts. (39e) is used when at least one filler of the specified type is required. For example, a whole has mony parts. (39f) is a simple prohibition stating that a word can not have a slot of some stated kind. In general, a WG grammar follows the *closed world hypothesis*, i.e. anything which is not explicitly allowed is considered to be implicitly forbidden. However, there are cases when explicit prohibitions are required, as we shall discover in section 9.2.2.

In recent presentations of the theory, Hudson has adopted an alternative, more flexible form of quantitator (Hudson 1990: 23–4). The new kind of quantitator is structured rather than atomic. Its structure is [*i*–*j*] where *i* and *j* are integers. *i* indicates the minimum number of fillers for the slot and *j* indicates the maximum number of fillers for the slot. Equivalences between the old and new systems are given in (40). I shall use the old system in all examples.

(40)

a	a X	=	[1–1]
b	ano X	=	[0–1]
c	mano X	=	[0–_]
d	many X	=	[2–_]
e	mony X	=	[1–_]
f	no X	=	[0–0]

The question of whether quantitators have the effect of creating multiple slots with identical properties or allowing single slots to have multiple fillers has to be worked out for any implementation but it has no theoretical importance.

Constraints can be placed on the range of potential slot-fillers by means of identity (is) propositions. For example,

(41)

a (subject of verb) is (a noun)
b (pre-adjunct of noun) is (a adjective)
c (comp of preposition) is (a noun)

In these examples, the second argument has the form *(a X)*. This use of *a* should not be confused with the use shown in (39a) and (40a). This version is simply used to distinguish the general case *X* from an instance of the general

case *(a X)*. The two versions appear in complementary distribution: the quantitator only appears in has propositions; the instance marker only appearrs in is propositions.

The predicates precedes and follows

precedes and follows are used to express relative linear orderings. For example:

(42)

a (subject of word2) precedes word2
b (object of word2) follows word2

Only one of these predicates is required to express linearization constraiints. For example, (43) shows the same facts as (42) but uses only one predicatte.

(43)

a (subject of word2) precedes word2
b word2 precedes (object of word2)

Redundancy is allowed to aid readability. There is no reason why an implementation should have to include both predicates. See Section 9.2.3 for furtther examples of the use of positional constraints.

The predicate isa

The isa predicate is used to relate entities to more general entities. For example:

(44)

a APPLE isa common-noun
b common-noun isa noun
c noun isa word

I say that the isa predicate is used to relate entities to entities, rather than entities to classes because WG assumes that the isa relation is a relation of instances to prototypes rather than a relation of members to classes. Unlike the is relation, the isa relation is antisymmetric.

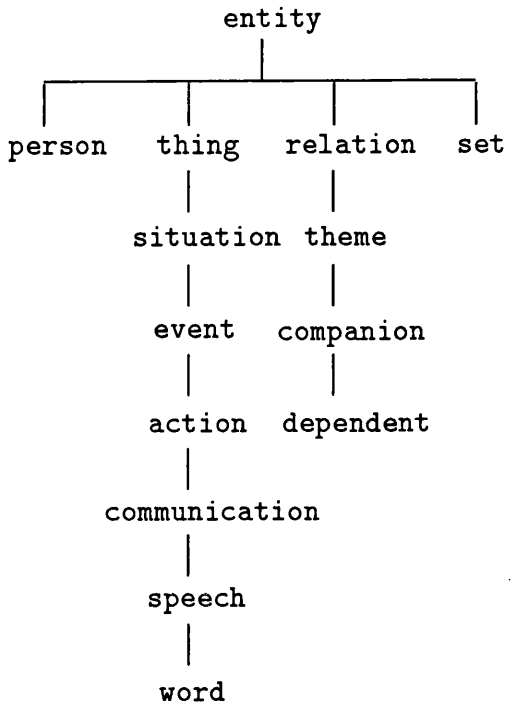


Figure 9.2: part of the WG ontological hierarchy

So far, I have described the kinds of predicates which can appear in propositions. I have presented propositions as devices for expressing facts about words. As it stands, this system has no mechanism for making or using generalizations. An adequate grammar must consist of more than a list of entries specifying all the properties of every word. It must make generalizations over collections of words. In the next section I describe how the isa predicate is used to make generalizations by allowing the properties of general cases to be transferred to specific instances.

9.2.2 Generalizations about words

All entities in a WG are thought of as belonging to a single, vast ontological hierarchy. Entities in the hierarchy are related by isa relations. Part of the top of the hierarchy is shown in Figure 9.2 (from Hudson 1990: 76).

The connections between lower and higher concepts in the hierarchy rep-

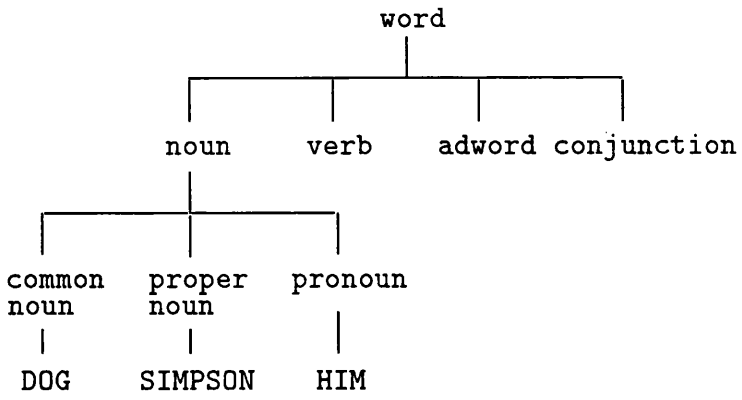


Figure 9.3: part of the WG word type hierarchy

resent isa relations. The hierarchy includes non-linguistic, as well as linguistic entities. The details need not concern us here. (For more information on the kinds of knowledge which a WG hierarchy represents see Hudson 1985a; Hudson 1986b; Hudson 1990: chapter 4). One part of the hierarchy of immediate relevance to this discussion is that part which comes below the 'word' entity and which could be described as the 'word type hierarchy'. Part of the word type hierarchy is shown in Figure 9.3.

The purpose of this hierarchical organization is to facilitate generalization. Any property which is shared by most or all common nouns is stored in relation to the 'common-noun' node in the hierarchy rather than at the level of 'DOG' or any other specific common noun. Any such property is said to be 'inheritable' by the lower node from the higher node. The simplest version of inheritance can be defined as follows (where P is any proposition):

(45)

IF X isa Y,
P is true of Y

THEN P is true of X

Most generalizations which can be made about language have got exceptions. Exceptions can be accommodated within the inheritance framework by stating the exceptional properties in relation to the highest node for which they hold

true. The property inheritance principle is then revised as follows:

(46)

IF X isa Y,
P is true of Y,
not: not: (P is true of X)[a]

THEN X has (Q P)

[a] 'It is not the case that X is prohibited from having the property P'.

Since inheritance is overrideable, it is often referred to as *default inheritance.* The usual properties are assumed for an entity unless there are good reasons (i.e. contradictory propositions) for thinking otherwise. For example, the usual way to form plural nouns in English is to add the S-morpheme ('mS') to the noun stem. This generalization can be made for all nouns. The relevant proposition would look something like (47).

(47)

(plural of noun) is ((stem of noun) + mS)

However, there are a small number of nouns (e.g. *salmon*) which exceptionally do not follow the normal plural rule. These would have to be specially marked so as to override the general rule. For example,

(48)

a (plural of SALMON) is <salmon>
b not: (plural of SALMON) is (<salmon> + mS)

In the case of words such as *hoof* which have coexistent default and exceptional plural forms, a proposition such as (49) is added to introduce the exceptional form and nothing is added to block the default form from being generated by (47).

(49)

(plural of HOOF) is <hooves>

Thus, *hoofs* can be generated using (47) and *hooves* can be generated using (49).

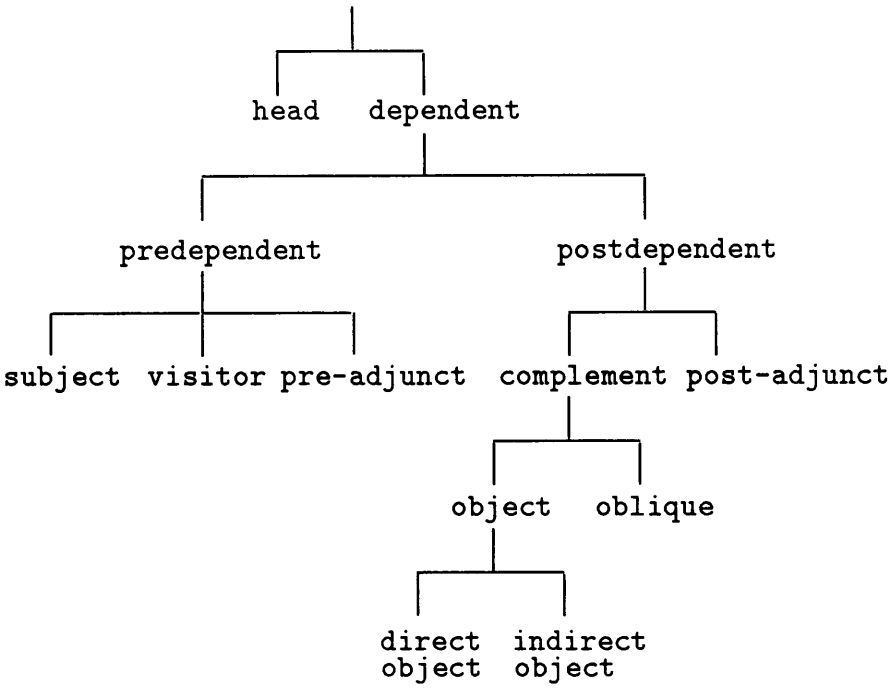


Figure 9.4: part of the WG grammatical relation hierarchy

Default inheritance is used in most modern linguistic theories (Gazdar 1987), although few of them make this explicit — HPSG (Flickinger et al. 1985; Flickinger 1987) is a notable exception. It is widely assumed that the primary use of default inheritance in linguistics is for expressing generalizations in morphology. However, since everything is expressed in relation to word-sized units in WG, syntax and semantics can also make use of default inheritance. The portion of the WG inheritance hierarchy presented in Figure 9.4 shows how the grammatical relations (i.e. types of dependency relation) can also be arranged in an inheritance hierarchy.

Thus, proposition (50) implies proposition (51).

(50)

X has (a subject)

(51)

X has (a predependent)

Table 9.1: inheriting properties for w1

Stored propositions	Added propositions
	w1 isa DOG
DOG has (a structure)	w1 has (a structure)
DOG isa common-noun	w1 isa common-noun
common-noun isa noun	w1 isa noun
noun has (a number)	w1 has (a number)
noun has (mano pre-adjunct)	noun has (mano pre-adjunct)
noun isa word	w1 isa word
word has (a head)	w1 has (a head)
word follows (pre-dependent of word)	w1 follows (pre-dependent of word)
word precedes (post-dependent of word)	w1 precedes (post-dependent of word)

The following simple (overrideable) propositions take care of normal English word order.

(52)

a word has (a dependent)
b (pre-dependent of word) precedes word
c (post-dependent of word) follows word

When a sentence is analysed in WG, every word is assigned a unique identifier such as w1 ('word 1'). Each word is analysed to establish its lexeme and morphosyntactic features. Once its lexeme has been found, the word instance can be attached to the bottom of the inheritance hierarchy underneath its lexeme. It can then inherit as many properties as possible from higher nodes. Consider w1, the first word in sentence (53).

(53)

Dogs chase large white rabbits.

In Table 9.1, the column on the left shows propositions contained in the grammar, while the column on the right lists the new propositions added for w1. Only a representative sample of propositions are shown.

The effect of the inheritance process is to build up a feature set for the word. Although absent from Table 9.1, constraints on slots are also inherited

during the process.

More detailed introductions to inheritance in WG can be found in Fraser and Hudson (1990), Hudson (1990a: chapter3), and Fraser and Hudson (1992).

9.2.3 A single-predicate system

I have already noted that the predicates precedes and follows are not both necessary. In fact, as Hudson points out (Hudson 1990: 24ff), only one predicate is really required, namely the is predicate. If this predicate is instead represented with the symbol ':', the grammar begins to look very similar to any other unification-based grammar. For example, the following examples show equivalent (a) standard WG five-predicate propositions, (b) WG one-predicate propositions, and (c) unification grammar feature structures (Shieber 1986).

(54)

a DOG isa noun
b (category of DOG) : noun
c DOG $\longmapsto \left[\begin{array}{ll} cat: & Noun \end{array} \right]$

(55)

a verb has ([1–1] subject)
b (quantity of (subject of verb)) : [1–1]
c $\left[\begin{array}{ll} cat: & Verb \\ arg1: & Subject \end{array} \right]$

(56)

a (subject of verb) is (a noun)
b (subject of verb) : (a noun)
c $\left[\begin{array}{ll} cat: & Verb \\ subject: & \left[\begin{array}{ll} cat: & Noun \end{array} \right] \end{array} \right]$

(57)

a (pre-dependent of word) precedes word
b (position of (predependent of word)) : before it
c $\left[\begin{array}{ll} cat: & Word \\ predep: & \left[\begin{array}{ll} posn: & before \end{array} \right] \end{array} \right]$

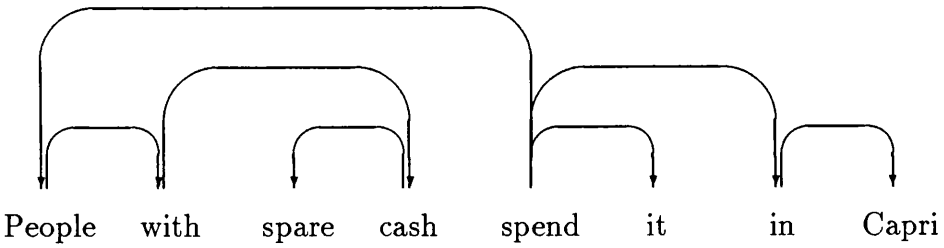


Figure 9.5: a WG dependency analysis

In his single-predicate version of WG, Hudson introduces extra 'positional names': before, after, adjacent-to and next-to. 'it' identifies the word referred to by the most deeply embedded concept to the left of the ':' predicate (i.e. 'word').

The purpose of this section is to emphasize the similarities between the expressiveness of the WG formalism and the expressiveness of other unification-based formalisms (e.g. GPSG, LFG, and Hellwig's DUG). This is not, however, to claim that they are identical nor that the insights typically expressed in these frameworks are the same. (For example, WG provides a much richer system of quantitators than any of the other frameworks.) The extent to which one theory differs from another is a complex question and one which can only be hindered by differences of notation. I have tried to show how easy it can be to convert WG grammars into a more familiar notation. This is a first step towards theory comparison. The next step goes beyond the scope of the present work.

9.2.4 Syntax in WG

Syntactic structure is expressed in terms of dependencies between word pairs, with the sole exception of coordinate constructions for which minimal constituent structure is used (Hudson 1989b; Hudson 1990: chapter 14). The sentence shown in Figure 9.5 illustrates a typical WG dependency analysis.

The sentence shown in Figure 9.6 is an example of the use of constituency

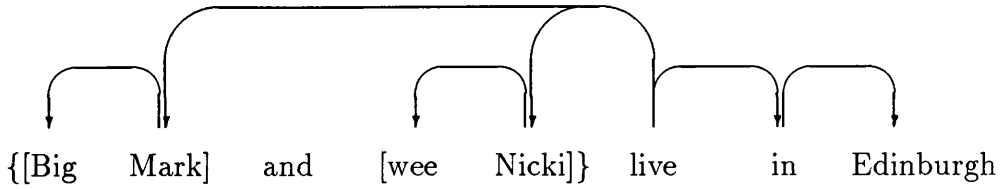


Figure 9.6: the use of constituency in WG

in WG. The brackets simply serve to identify the boundaries of the coordinate structure and its component conjuncts. Dependencies between elements of the coordinate structure and elements outside the coordinate structure are controlled by the Dependency in Coordinate Structure (DICS) principle. This states that:

> any word which is outside a coordination C but which is in a dependency relation D to some conjunct-root of one conjunct of C must also be in relation D to one conjunct root in every other conjunct of C (Hudson 1990: 413).

A 'conjunct-root' is simply a head of a conjunct. In the case of Figure 9.6, the conjunct-roots are *Mark* and *Nicki.*

Apart from the exceptional case of coordination, all other syntactic structure is expressed in terms of pairwise dependencies.

WG makes use of a modified adjacency principle since, under certain circumstances, words are allowed to depend on more than one head.

The Adjacency Principle

> D is adjacent to H provided that every word between D and H is a subordinate either of H, or of a mutual head of D and H (Hudson 1990: 117).

The sentence in Figure 9.7 shows an example of a dependency structure which is permitted by WG's adjacency principle but forbidden by the standard version of adjacency as defined by Gaifman (1965) (*I* is separated from its head *to* by *want* which does not depend on either *I* or *to*).

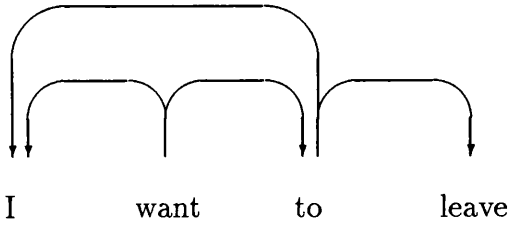


Figure 9.7: a structure permitted by WG's version of adjacency

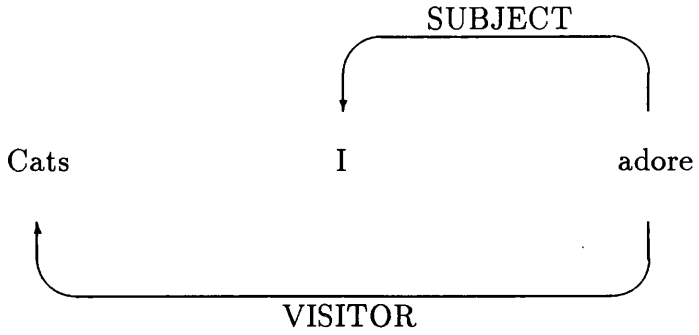


Figure 9.8: the use of visitor links to bind an extracted element to the main verb

The WG analysis of extraction relies upon a word having more than one head. In this analysis, the extracted word is first bound to the main verb by a semantically empty dependency link known as the 'visitor' relation. The grammar would include rules such as those in (58).

(58)

a finite verb has (ano visitor)
b (visitor of verb) precedes (subject of verb)

Thus, in the sentence *Cats I adore*, *cats* is bound as the visitor of *adore* as shown in Figure 9.8 (visitor links are drawn below the sentence).

It is a simple matter to use the visitor link to establish the object relation between the verb and *cats*. The general form of the rule for identifying normal

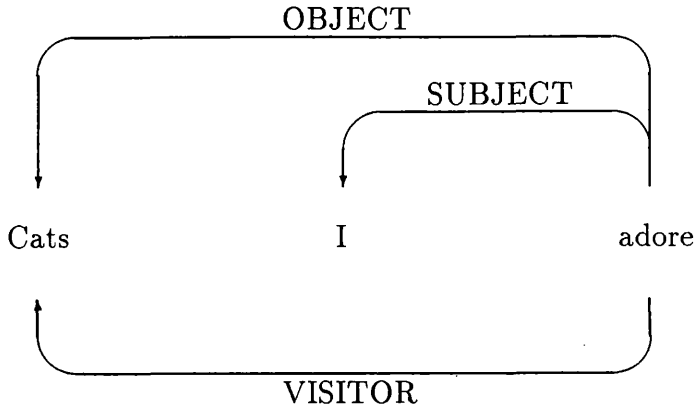


Figure 9.9: the use of the visitor link to relate the extracted element to the main verb as its object

postdependents with the unusual visitor link is shown in (59).

(59)

(visitor of word) is (a (post-dependent of word))

This would lead to the analysis shown in Figure 9.9.

Since the visitor relation is semantically vacuous, the propositional content of the sentence is the same as it would be if extraction had not taken place. However, the presence of the visitor link introduces markedness to the construction, as would be expected. The example sentence does not represent a convincing argument for the use of visitor links since a simple rule could have allowed the object to depend on the verb directly without the mediation of the visitor. (60) offers a better example since there is more intervening material between the extracted item and its head.

(60)

Cats I think you know I adore

Only one extra rule is required to copy the visitor link from verb to verb, thus producing a 'hopping' analysis. The rule appears in (61).

(61)

(visitor of word) is (a (visitor of (complement of word)))

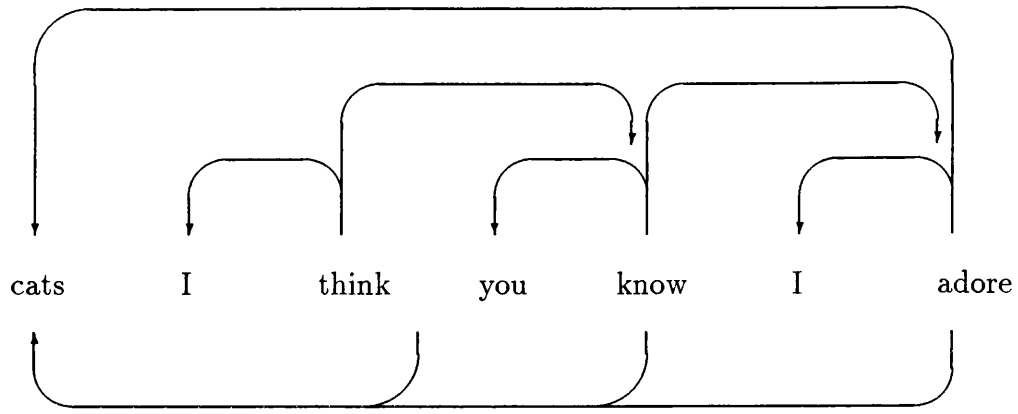


Figure 9.10: the use of visitor links to interpret the object of an embedded sentence

This rule allows sentences like (60) to be analysed without difficulty. The resulting structure is shown in Figure 9.10.

A more detailed exposition of the use of visitor links in WG can be found in Hudson (1988b).

Apart from (i) allowing constituency in coordinate constructions, (ii) allowing multiple heads, and (iii) providing a modified adjacency principle, WG abides by the definition of DG supplied by Gaifman. Exceptions (i)–(iii) may be regarded as extensions to the expressiveness of the standard dependency formalism, whose formal properties and consequences are as yet undefined formally.

9.2.5 Semantics in Word Grammar

Semantics in WG relies upon two basic premises:

1. Virtually every word in a sentence is linked to a single element in the semantic structure.

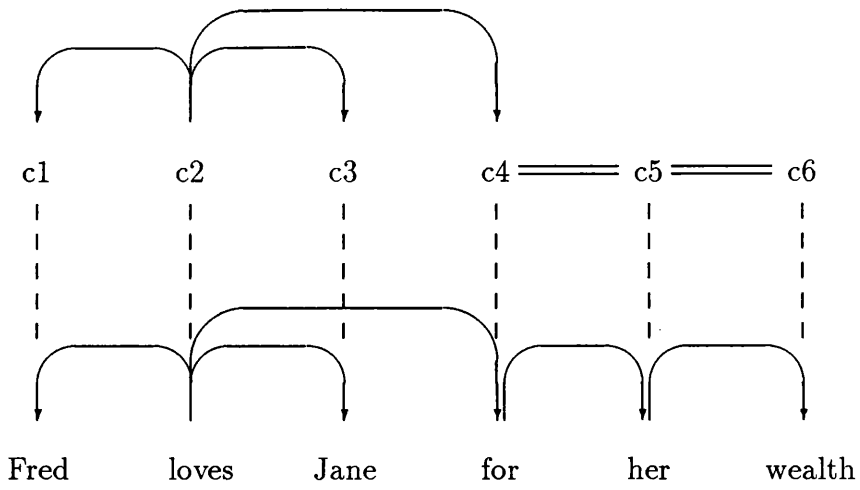


Figure 9.11: semantic structure is very similar to syntactic structure in WG

2. There is a high degree of congruence between syntactic dependencies and semantic dependencies.

The elements in semantic structure to which words are linked are called 'referents'. These are taken to be mental concepts rather than objects in the world. The two basic kinds of relation which may hold between referents are *dependency* and *identity*. A simple example of a possible WG semantic rule which is parasitic upon the syntactic structure, is given in (62).

(62)

(referent of (subject of LOVE)) is (actor of (referent of LOVE))

The diagram shown in Figure 9.11 (from Hudson 1990: 123) should serve to illustrate the extent of congruence between syntactic and semantic structures in WG. In the diagram, the labels c1, c2, etc. are the conceptual referents of the words to which they are linked by dotted lines. Arrows between referents show semantic dependencies. Equality operators between referents show identity.

This degree of isomorphism between syntax and semantics allows the semantics simply to be 'read off' the syntactic structure in many cases. One of my own early WG parsers succeeded in constructing semantic structures for

a respectable range of sentences with minimal effort required (Fraser 1988). However, it would be foolish to pretend that all semantic analyses are equally easy. Some difficult problems remain to be solved. To date, semantics in WG has not received as much attention as syntax. It is to be hoped that this imbalance will be corrected before too long. In the meantime, the only computer system to attempt any WG semantic analysis, other than my own, is Gorayska's small-scale 'WG semantic analyzer' (Gorayska 1987).

9.3 Word Grammar parsing

In early 1985 Richard Hudson produced a very modest WG parser written in BBC Basic and running on a home computer with just 32K of RAM (Hudson 1985b). At that stage Hudson described himself as "an amateur with more enthusiasm than programming skills". However modest the parser may have been, it became the inspiration for my own larger scale parser (written in Prolog) which formed the basis of my Masters dissertation (Fraser 1985). The main strengths of this system were its complete separation of grammar and parser and its simple but effective implementation of default inheritance. These features have continued to inform subsequent systems developed by Hudson and myself at University College London. Unfortunately, my parser failed to solve the problem of implementing the adjacency principle and so it failed in the most important task of a parser, namely building appropriate syntactic structures.

Early in 1986 we became aware of a group of computer scientists at Imperial College, London who were beginning to show interest in the ideas contained in Hudson's 1984 monograph. This group, and especially Derek Brough, wrote a number of very small trial parsers (Brough 1986). Around this time a former student of Hudson's, Max Volino, also wrote a small parser based on WG. Hudson himself had moved on from his computational small beginnings and was now using Prolog on a much more powerful machine. Hudson (1986a)

reports Hudson's first parser written in Prolog.

In late 1986 I started to work as Hudson's research assistant and began to develop some of the ideas first presented in my Masters dissertation. This soon resulted in the production of a parser which combined an inheritance mechanism with a functional (though clumsy) parsing strategy (Fraser 1988). Like all WG parsers developed up until then, this one incorporated an explicit check on the adjacency of words to be linked. This was very expensive computationally so the parser ran rather slowly. It was the first WG parser to build simple semantic structures as well as syntactic structures. Later that year, Barbara Gorayska (a former doctoral student of Hudson's) produced a more sophisticated WG semantic analyzer (Gorayska 1987) although it was not part of a parsing system. Parsers loosely based on WG were also produced as final year projects by Francis Bell, an undergraduate at Westfield College in London and, in 1987, by Phil Grantham, a postgraduate student at Sheffield Polytechnic (Grantham 1987).

During the period 1987–8, the two largest scale WG parsers produced to date were being developed in parallel at University College London by Hudson and myself. While we exchanged views and insights on theoretical matters during this period, we kept the implementational and algorithmic details of the systems to ourselves, thus ensuring that two distinct implementations evolved. In the remaining sections of this chapter, these two parsers are described in more detail.

9.3.1 Fraser's parser

Objectives of the parser

I had two main objectives in writing my parser. The first objective, which it shared with my earlier WG parsers, was simply to see what a WG parser would look like. Could it be a minor modification of an existing parsing algorithm or would it involve distinct problems requiring distinct solutions? Once a few

trial systems had been constructed I felt that I was in a position to identify some problems which seemed to be common to all of the parsers. Solving these problems became the principal focus of the parser I report here.

The main difficulty which plagued the early WG parsers was the time they took to parse sentences. They ran very slowly, even when working with small grammars on powerful machines. My best WG parser up to that time had taken 254 seconds to find a first reading for the seven word sentence *This sentence was analyzed by a computer*, even though the grammar-lexicon contained little more than what was required to process the sentence and in spite of the fact that the program was running on a single-user Sun workstation (Fraser 1988: 58). At least part of the reason for the poor performance could be attributed to some features of the version of Prolog I was using. My program had made extensive use of the `assert` and `retract` predicates to add facts to, and remove facts from the Prolog database. Alarmed by the poor performance times, I carried out a series of benchmark tests and discovered that it was much quicker to maintain a record of the current state of the parse in long environment lists which could be passed between predicates than to write to and erase from the Prolog database. This problem was easily solved, and the parser described here seldom asserts and never retracts during parsing.

However, not all speed-related problems sprung from the mundane details of the implementation. Some had more significant theoretical origins. Chief amongst these were the role of the adjacency constraint in the parser and the question of how best to generate all readings for a sentence.

In all of the WG parsers available up to that time, the adjacency constraint was implemented as an explicit permissibility check on a hypothesized dependency relation between two words, no doubt because that is the way in which it is presented in Hudson (1984). In this respect these parsers differ from all of the other parsers described in this thesis which either have no adjacency constraint or which build the constraint into the parser's control strategy. By

profiling my earlier parsers I discovered that most of the processing time was devoted to selecting potential word pairs, checking that they could contract a dependency relation, checking whether potential dependency pairs were adjacent and then discovering that they were not. Given an n word sentence, it is possible to hypothesize dependency relations between any word and every one of the other words in the sentence, i.e. $n-1$ other words. The sentence as a whole (assuming no lexical ambiguity) could generate a maximum number of $n(n-1)$ hypothetical relations. Most of these relations would be rejected by the adjacency constraint (and, of course, by the dependency requirements of each word). It struck me that this was approaching the problem the wrong way round. If a parser were constructed in such a way that it never hypothesized a relation between two words unless they were adjacent, this ought to avoid a considerable amount of wasted effort.

The solution to this problem was to construct the parser around an explicit stack and to stipulate that the only place which could be searched for a dependent or head for the current word was at the top of the stack. The main difficulty for this approach was in establishing dependencies between word pairs when one of the words had been extracted. The solution I adopted was to separate dependency relations into those which had to be *discovered* by search and those which could be *derived* from the dependency relations already in existence.

The second problem I addressed in my parser was how to increase efficiency in the discovery of all possible readings for a sentence. I did not want to use a chart because I was unsure how to represent discontinuous groups of dependents and, more problematically, how to deal with the uncertainties raised by the possibility of multiple headedness. Choosing a completely different approach I decided to construct a backtracking parser which was designed in such a way as to spot implausible analyses as early as possible, thus keeping to a minimum the amount of useless structure which would be built. Needless

to say, this could not prevent the parser from duplicating effort in some cases.

The parser

As we have seen, one of the central claims of DG in general and WG in particular is that a grammar need only refer to word-sized units. However, there is no theoretical reason why a WG parser should share the same restrictions as the grammar it uses. I propose that, while a grammar may only refer to word-sized items, a parser should be allowed to refer in addition to two other kinds of data structure, namely *molecules* and *stacks.*

Following the example of Tesnière, I draw an analogy from molecular chemistry and the process of chemical bonding. An atom with an overall positive or negative charge is called an ion and is said to have a valency. Similarly, a single word is said to have a valency. Where ions have positive charge, words have a requirement for dependents; where ions have negative charge, words have a requirement for a head. When a positively charged ion meets a negatively charged ion (and other factors permit) the two ions bond to form a single molecule. Any imbalance in charge between the two ions remains as a property of the molecule (although ultimately it is the property of a single nucleus). In similar fashion, a word which requires (or allows) a dependent can bond with a word which requires a head to form a molecule unless any constraints prevent it. Any dependency slots (charges) not involved in this bond remain as properties of the molecule. Molecules can bond with other molecules. Well-formedness is analogous to molecular stability in chemistry — in my model a molecule with saturated valency can serve as a sentence (so long as its root does not require a head).

For obvious reasons, the parsing procedure presented here is called the *bonding algorithm.*

Molecules A molecule is a structure consisting of a root word plus all of its subordinates discovered so far. Molecules are 4-tuples of the form shown in

(63).

(63)

```
[Negative-list, Positive-list, Subordinates, Derivable]
```

Negative-list is a list of unfilled head slots. *Positive-list* is a list of unfilled dependent slots. The general form of a slot is shown in (64).

(64)

```
[NUMBER, TYPE, SLOT-LABEL, SLOT-TYPE, POSITION]
```

NUMBER is a unique identifier for the word which has the slot (e.g. w5). TYPE is that word's word type (e.g. verb). SLOT-LABEL identifies the kind of dependency relation which must hold between the word and its slot filler (e.g. subject). SLOT-TYPE is the word type required of the slot filler (e.g. noun). POSITION indicates the filler's position relative to the word which has the slot. There are three values for POSITION, namely `before`, `after` and `either`.

There is, of course, a certain amount of arbitrariness in the association of positive charge with dependency requirements and negative charge with head requirements rather than *vice versa*. The significant point to note is that they are mutually attractive opposites.

Subordinates is a structured list containing a record of all of the root word's subordinates and the dependency relations involved.

Derivable is a list of slots and information detailing how to derive their fillers from existing dependency relations.

Stacks The bonding algorithm makes use of a single parse stack, and only molecules may be pushed onto it. The way in which the stack is used ensures that only adjacent words can be bonded.

Preliminaries The parser works in a left-to-right, bottom-up, single-pass manner. The parser reads one word at a time, constructing for each word a

frame of slots and constraints on fillers. The information which is used to build a frame is obtained from the grammar by a process of property inheritance. For example, the propositions shown in (65) could be inherited for the first word of sentence (66).

(65)

word-1 isa proper-noun

word-1 has (a head)
word-1 has (mano pre-adjunct)
word-1 has (ano post-adjunct)

(pre-adjunct of word-1) is (a adjective)
(post-adjunct of word-1) is (a preposition)
(pre-adjunct of word-1) precedes word-1
(post-adjunct of word-1) follows word-1

(66)

John loves Mary

The same information can be expressed much more compactly when it is converted into molecule format. The molecule which would be constructed for word-1 is shown in (67).

(67)

```
[ [ [ 1, proper-noun, [a, head], word, either] ],
  [ [ 1, proper-noun, [mano, pre-adjunct], adjective, before],
    [ 1, proper-noun, [ano, post-adjunct], preposition, after] ],
    [],
    []
]
```

(68) shows the molecule initially constructed for the second word of sentence (66).

(68)

```
[ [],
  [ [ 2, finite verb, [a, subject], noun, after],
    [ 2, finite verb, [a, object], noun, after] ],
    [],
    [] ]
```

For the sake of simplicity I have ignored the requirement in English for subject-verb agreement. This can be accommodated in the framework but it would

require some digression.

Molecular bonding At the heart of the parser lies a process for combining molecules to form larger molecules. In general, if some element of the Positive-list of one molecule can be combined with some element of the Negative-list of another molecule then the second molecule can be merged into the first to produce a new, larger molecule.

In order to facilitate description of the process of molecular bonding, I shall identify the elements of Positive-lists and Negative-lists by means of the names given in (69).

(69)

```
[ number, type, [quantitator, slot], slot-type, order ]
```

An attempt can be made to bond (67) and (68) by trying to unify an element from one Negative-list with an element from the Positive-list of the other molecule. We shall say that the two unify if the following conditions hold:

1. A isa B, where A is the Negative-list **type** and B is the Positive-list **slot-type**; and
2. C isa D, where C is the Positive-list **type** and D is the Negative list **slot-type**; and
3. the Positive and Negative orders unify (**before** unifies with **before**, **after** unifies with **after**, **either** unifies with anything, but **before** and **after** will not unify with each other).

Let us consider the first element of the Positive-list of (68) and the first (and only) element of the Negative-list of (67). These are shown in (70).

(70)

```
+ve  [ 1, proper-noun, [a, head], word, either ]
-ve  [ 2, finite verb, [a, subject], noun, before ]
```

When we try to unify these lists we find that:

1. 'proper-noun isa noun' succeeds; and

2. 'finite verb isa word' succeeds; and

3. 'unify `either` with `before`' succeeds

therefore all of the conditions are satisfied and the molecules may bond. The structure of the resulting molecule is shown in (71).

(71)

```
[ [],
  [ [ 2, finite verb, [a, object], noun, after],
    [ 1, proper-noun, [ano, post-adjunct], preposition, after] ],
  [ subject, 2, 1],
  [] ]
```

Several interesting things have happened here. First of all, the matching elements — a Negative element of (67) and a Positive element of (68) — have collapsed into a single element which is recorded in the Subordinates list (read this as 'the subject of word-2 is word-1'). In addition, two Positive elements of (67) have been deleted. The reason for this will soon become apparent.

Using the stack Only two molecules are available for bonding at any time, namely the top two molecules on the parse stack. I shall refer to the top-most molecule as M1 and the next one down as M2. To begin with, a test is made to see if M2 can depend on M1 (i.e. if some element in M1's Positive-list will unify with some element in M2's Negative-list). If this test succeeds then the two molecules bond to form a new molecule. If not, then a test is made to see if M1 can depend on M2 (i.e. if some element in M1's Negative-list will unify with some element in M2's Positive-list). Again, if they unify, the two molecules bond to form a new one. This becomes the new M1 and the next highest stack element becomes available as M2. If two molecules will not bond, then the stack remains unchanged. The next word of the sentence is read and a new molecule is constructed and added to the parse stack.

By the end of a sentence, there should be exactly one molecule left on the stack. If there is more than one then the parser has failed to find a single

dependency structure for the input string.

The parser only ever searches its immediate left context. In this way the operation of the stack implicitly applies the adjacency constraint. Thus, one of the objectives of the parser has been satisfied: the parser never attempts to establish a dependency relationship between a pair of words unless they are adjacent. Note also, that (unlike in earlier WG parsers) there is no search involved. There is only one place to look for a head or for a dependent, namely M2. If it is not there then there is no need to look any further.

Another strength of this stack-based approach is that it provides neat ways of identifying and closing down doomed search paths as early as possible — thus satisfying the other main objective of the parser. Recall that when we combined molecules (67) and (68), we produced a new molecule (71). However, in the process, we lost the two slots shown in (72).

(72)

a `[ 1, proper-noun, [mano, pre-adjunct], adjective, before]`
b `[ 1, proper-noun, [ano, post-adjunct], preposition, after]`

It should be obvious that the first word of a sentence can not possibly have a pre-adjunct. However, it is possible to appeal to a more general principle which states that any M1 which has optional slots for dependents with the **before** order feature, will have these options closed if it is found that there is nothing else on the stack. This is because there are not, and never will be, any available fillers. This accounts for the disappearance of slot (72a). Likewise, if M1 has non-optional slots for preceding fillers and there is nothing else on the stack, then no single dependency structure will ever be able to link all of the words in the string into a coherent sentence. This fact can be used to spot impossible analyses before further structure is built fruitlessly. If this heuristic were not applied, parsing could continue until the end of the input string before the problem was spotted.

The reason for the erasure of the post-adjunct slot (72b) is that there is a rule which states that when an M1 becomes the head of an M2, any optional

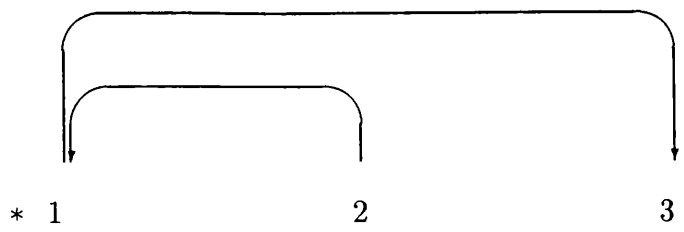


Figure 9.12: a prohibited dependency structure

`after` slots the M2 may have had are removed. This is because structures of the sort shown in Figure 9.12 can not occur.

Had the slot been obligatory and not just optional, this would have signalled that further processing would be pointless: no successful parse could ever result.

Thus, at the cost of two simple tests at bonding time, the amount of needless processing can be significantly reduced. I shall show below how the parser's efficiency can be further enhanced by examining the gross characteristics of the stack whenever a molecule is pushed onto it.

First, though, here is a PARS description of my parsing algorithm.[3]

[3]It is necessary to define an extra condition 'obligatory_slots(X,Y)' for this PARS description. This condition succeeds if word Y has any obligatory slots in position X (e.g. obligatory_slots(before, C)), otherwise it fails. It is also necessary to define a special action 'strip_optional_slots(X,Y)' which strips out any optional slots belonging to Y with positional feature X (e.g. strip_optional_slots(after, C)).

INITIALIZATION: read input words into a list
(in molecule format);
C is the current word in the list;
C:=1;
X is a pointer;
X:=1;
initialize an empty stack;
the result is stored in the variable Result.

1. **IF** empty(Stack)

THEN IF obligatory_slots(before, X),

THEN fail

ELSE strip_optional_slots(before, X),

push(X),

C:=C+1,

X:=C,

goto(2)

ELSE IF X → top(Stack)

THEN IF obligatory_slots(after, top(Stack))

THEN fail

ELSE strip_optional_slots(after, top(Stack)),

record(X → top(Stack)),

pop(Stack),

goto(1)

ELSE IF top(Stack) → X

THEN IF obligatory_slots(before, X)

THEN fail

ELSE strip_optional_slots(before, X),

record(top(Stack) → X),

X:=top(Stack),

goto(1)

ELSE push(X),

C:=C+1,

X:=C,

goto(2).

2. **IF** C=*e*

THEN Result:=top(Stack),
pop(Stack),
IF empty(Stack)
THEN succeed
ELSE fail

ELSE goto(1).

Algorithm 9.1: Fraser's 'bonding' algorithm

Derived dependency relations Consider sentence (73), in which the object *the thesis* has been extracted out of its normal post-verbal position.

(73)

The thesis I wrote

At a certain point in the analysis of this sentence, M1 will be the molecule *I wrote* (headed by *wrote*) and M2 will be the molecule *the thesis* (headed — according to normal WG practice — by the determiner *the*). Recall from our discussion of visitors that a tensed verb may have a preceding visitor. In this case, *the (thesis)* is recognized as the visitor of *wrote*. When *the (thesis)* becomes visitor of *wrote* it is absorbed into the molecule headed by *wrote* and disappears from view. However, it is still necessary to identify *the (thesis)* as the object of *wrote*. This is where the 'Derivable' component of a molecule finds its use. The Derivable list contains identity propositions. In this case, there is a proposition which equates the object of a tensed verb with the visitor of that tensed verb. The Derivable list is checked after each new dependency relation is established and any additional relations which may be derived are added to the parse record. In this way, the parser is able to build all of the multiple-headed structures which are sanctioned by WG theory.

Additional optimizations The root of a sentence differs from all of the other words in a sentence in that it has an empty Negative-list (i.e. it does not require a head). This makes it easily identifiable during parsing. One useful consequence of the adjacency constraint is that no (non-derived) dependency relation will ever cross the root. Therefore, when the root is pushed onto the stack, the stack *must* be empty, otherwise the molecule or molecules left on the stack will never be integrated into the molecule headed by the root. This is a robust test which, together with those already mentioned, contributes to the parser's early recognition of fruitless search paths.

There is at least one fragile — but nonetheless useful — heuristic which can also improve the average performance of the parser. Apart from the hand-analyzed BKB corpus compiled for the DLT project (7), the only corpora analyzed in terms of dependency structure known to me were constructed by Dick Hudson, Monika Pounder and myself at University College London. These were very small, exploratory corpora, which had no claims whatsoever to statistical significance. However, a striking feature of the dependency trees was observable. If an arbitrary word in any of the corpora were chosen, and it were assumed that the sentence were being parsed by an incremental, left to right parser like the one I have just described, then at the chosen point in the analysis, the maximum number of unsatisfied dependencies hardly ever exceeded three, and certainly never exceeded four. If this result could be shown to be valid for a corpus of significant size, it would have implications for the design of backtracking parsers. If it is valid, then the chances of a successful result would be very slim from a parser state in which four or more molecules were resident on the stack. This constraint is fragile — after all, it is possible to stack up arbitrarily many adjectives before a noun — but it may prove to have a useful heuristic function in the majority of cases.

Implementation details The parser is implemented in Poplog Prolog on a Sun 3/52 workstation. It can analyze a wide range of English constructions

while maintaining consistently high levels of efficiency. The parser analyzes sentences left to right incrementally in real time — it takes 0.23–0.25secs to establish a dependency relation, with a vocabulary of approximately 500 lexical items. This is roughly 1/64 of the time taken by the parser's immediate predecessor.

Complexity The absolute time taken by the parser is, of course, dependent on the hardware and software platforms used. Some implementation-independent measure of performance is more desirable. In particular, the asymptotic complexity of the parser is of interest. It is worth pointing out that the optimizations to the bonding algorithm described above do not affect the asymptotic complexity; they only affect the size of the constant in the calculation.

Assuming for a moment that the parser operates with a completely unambiguous grammar, the maximum amount of work required to find the dependents of a word and the head of a word is constant for all words. Therefore the parser takes time proportional to n, i.e. it operates in linear time. Although no formal complexity proof has been constructed, it is hard to see how the formal result could differ from the one arrived at informally here. Empirical experiments with the parser support this result. (The average time of 0.23–0.25 secs taken to establish a dependency relation was constant even for sentences of more than forty words in length.)

Given the prevalence of ambiguity in natural language, it is unrealistic to suppose that a practical version of the parser would be able to operate without being forced to backtrack. Like most parsers which make no use of charts, the time taken to find every reading for an ambiguous sentence is proportional to n^n in the worst case. The effects of stack-related early recognition of failure have not been taken into consideration in arriving at this figure. It seems likely that addition of a chart to the parser would result in polynomial time complexity.

9.3.2 Hudson's parser

Objectives of the parser

Hudson had two main objectives in writing his WG parser. First, he was interested in developing a tool which would help him to write consistent large scale grammars of natural languages. It is very difficult when writing realistically-sized grammars to maintain internal consistency and to anticipate all the consequences of the addition of some new grammar rule or rules. One solution to this problem is to build a computational environment, a 'grammarians workbench', which allows the grammar writer to modify the grammar and then to check the consequences of the modification by parsing a set of test sentences. The test sentences have previously been parsed 'by hand' so the target structures are known. Ideally, any modifications to the grammar should increase the number of test sentences which the parser analyses correctly. Since Hudson's workbench is designed to be used by linguists rather than computer scientists, grammar rules can be written in a slightly modified dialect of the WG notation reviewed above. This is automatically compiled into a denser, less readable system-internal representation. It is not necessary to be familiar with this representation in order to understand the algorithm. A grammarian's workbench should be usable with a range of grammars so that alternative analyses can be tried. This requires that the analysis system be completely separate from the grammar. In Hudson's system (as in my own) the parser and the grammar are clearly distinct, even to the extent of residing in different computer files. The grammars are collections of declarative facts which can be slotted into the procedural parser. The only linguistic objects that the parser knows about are very general objects (which are not specific to any language or construction) such as 'dependent', 'head' and 'word'.

Hudson's second object in writing his parser was to produce a model of human sentence processing. The desire to produce a parser which is, in some sense, a cognitive model leads to a design strategy which eschews parsing

techniques which are computationally efficient but cognitively unmotivated. In so far as WG has ambitions to be a theory with claims to make about cognition — and it does — the aim of building a computational cognitive model should be satisfied by following the theory as closely as possible in the implementation. The theory calls for incremental processing of sentences and the generation of all possible alternative analyses for each sentence. This is a considerable simplification of what humans seem to do. There is evidence that while people do process sentences incrementally, they do so by entertaining several analyses for a limited period only before selecting some particular reading for a word. This means that most of the time alternative analyses are not carried all the way through a sentence. One consequence of this is that it is possible to make a wrong decision which subsequently has to be undone. Garden path sentences (Marcus 1980) illustrate this phenomenon. Hudson's parser is thus only a model of certain aspects of incrementality and ambiguity handling since it *always* processes incrementally and *always* builds all possible readings for a sentence in parallel. What is perhaps WG's most interesting cognitively-motivated principle, the 'Best Fit Principle' (BFP), is not modelled at all in the parser. The BFP is designed to allow the grammar to be used to analyse sentences which are to some extent ill-formed. That is, they do not reflect anything in the competence grammar directly. The BFP is worded as follows:

The Best Fit Principle

> An experience E is interpreted as an instance of some concept C if more information can be inherited about E from C than from any alternative to C (Hudson 1990: 47).

In effect, the BFP is a pragmatic principle which always steers processing in the direction of the greatest net gain in information (in this respect it is rather like the Principle of Relevance of Sperber and Wilson 1986). It calls for *constraint relaxation* in matching a word instance to its model in the isa hierarchy since it is accepted that the match may not be exact. (For a review of some constraint relaxation techniques in natural language processing see

Fraser and Wooffitt 1990.) This could be expected to have a profound effect on the design of a computer model. Sadly, all WG parsers produced to date implement an 'Exact Fit Principle' rather than a BFP. It is to be hoped that the next generation of WG parsers will tackle this problem.

The parser

Hudson's parser is written in Prolog2 and runs on an IBM XT. It processes each sentence from left to right, one word at a time. When a word is read into the system it is first analyzed morphologically into a stem and (optionally) an affix. The stem is used to locate where to attach the word instance in the inheritance hierarchy. The affix (or absence of one) is used to determine the word's morphosyntactic features. I shall not describe the morphological analyzer here. Details can be found in Hudson (1989c: 327ff; a fuller treatment of morphology in WG can be found in Hudson (1990a: Appendix 8). Each word is assigned a unique identifier which, for convenience, is an integer corresponding to the position of the word in the input string. Additionally, each reading of a word is assigned a distinct number which is also an integer. The first reading found is assigned the identifier 1, the second reading is assigned 2, etc. Thus, any word in the system is identified by a two element list. The first element identifies its position, the second element identifies its reading. In sentence (74), the first occurrence of *saw* would have a nominal and a verbal reading. Thus, two distinct words would be identified: [2,1] and [2,2]. The second occurrence of *saw* similarly has two readings, distinguished as [4,1] and [4,2]. (In the latter case the ambiguity is only local).

(74)

I saw his saw

Next, the parser has to inherit properties for each reading identified. It is not clear whether it is better to inherit properties all at once or in a demand-driven way. In this parser all inheritable properties are collected together at once and

built into a feature structure associated with the word instance. The feature structure includes properties inherent to the word such as tense, number, etc. It also includes information about the word's possible dependents and, more controversially, about its head. That is, most words other than finite verbs will have associated with them a proposition such as

(75)

[6,1] has (a head)

and possibly an additional proposition which identifies the kind of word which may serve as a head:

(76)

(head of [6,1]) is (a noun)

The algorithm always tries to link a word to a preceding word. It never searches the right context. It begins by trying to find a dependent for the current word, starting with the closest preceding headless word and working back towards the first word. This process continues until all dependents are found for the current word or until no more options are available. Next, the current word searches the previous context trying to find another word (only roots of partial trees are considered) which could serve as its head. If it is successful then the next word is read in, morphologically analysed, assigned default properties and made the current word in the parser. If no head is found for the current word then checks are made to see whether (on the basis of local knowledge) the current word could possibly be the sentence root or, alternatively, if it could depend on a word which has not been read in yet. If either of these options is not ruled out then the next word becomes the current word. If neither option is possible then an attempt is made to take the current word as the root of a conjunct in a coordinate structure. If this is possible then it is necessary to copy any dependency relations which hold between any other conjunct roots and words outside the coordinate structure. If none of these tests succeeds then the parse has failed. The parse succeeds when the final word has been

processed and all of the words are subordinate to a single root.

Hudson describes his algorithm as follows (quoted from Hudson 1989c: 334):

1. try to take the nearest preceding word X that has no head as a dependent of W.
   (a) If successful, repeat 1, with reference to the last word before X that has no head;
   (b) Otherwise, go to 2.
2. Try to take a root of the nearest preceding word Y as head of W.
   (a) If successful, stop.
   (b) Otherwise, go to 3.
3. Try to take W as a word which need not have a preceding head, either because it needs no head at all, or because it may have a following head.
   (a) If successful, stop.
   (b) Otherwise, go to 4.
4. Try to take W as the root of a conjunct which shares its external relations with earlier conjunct-roots of a coordination.
   (a) If successful, stop.
   (b) Otherwise, fail.

Algorithm 9.2 presents a PARS description of Hudson's parsing algorithm (omitting the conjunct root test for simplicity).

INITIALIZATION: read input words into a list;
C is the current word in the list;
C:=1;
initialize a stack, Stack;
push(Stack, C);
C:=2;
X is a global variable;
the result is stored in the variable Root;
the action 'root(Root)' succeeds if Root
does not require a head.

1. **IF** C = e

   THEN goto(3)

   ELSE IF empty(Stack)

   THEN goto(2)

   ELSE IF C → top(Stack)

   THEN record(C → top(Stack)),
   remove(top(stack)),
   pop(Stack),
   goto(1)

   ELSE X:=C–1;
   goto(2).

2. **IF** X = 0

   THEN push(C),
   C:=C+1,
   goto(1)

   ELSE IF X → C

   THEN record(X → C),
   C:=C+1,
   goto(1)

   ELSE X:=X–1,
   goto(3).

3. Root:=top(Stack),
   pop(Stack),

   IF (root(Root) & empty(Stack))

   THEN succeed

   ELSE fail.

Algorithm 9.2: Hudson's dependency parsing algorithm

Unlike Hudson's previous parsers (Hudson 1985b; Hudson 1986a), this one includes no explicit adjacency test. Instead, adjacency checking is implicit in the parsing algorithm. This is interesting since Hudson is concerned with cognitive modelling. By making the adjacency principle inhere in the parser, he is making the claim that the adjacency constraint applies in all languages. This would be an unreasonable claim to make for the traditional version of the adjacency constraint. However, it may not be unreasonable given Hudson's revision of the principle. This is an empirical question which awaits further investigation.

We have seen how structured names for word instances distinguish them on the basis of position and reading. However, this does not cover all possible ambiguities. There may also be ambiguities of attachment. For example, in sentence (77) the phrase *with a telescope* could modify either *saw* or *the man.* The alternative analyses are shown in Figures 9.13 and 9.14. (The standard WG analysis requires nouns to depend on determiners rather than *vice versa.*)

(77)

I saw the man with a telescope.

To distinguish the different attachments, it is necessary to add another component to a word instance's identifier. So, for example, the instance of *with* which depends on *saw* might be identified as [5,1,1], whereas the instance which depends on *telescope* would be identified as [5,1,2]. Although not shown in the formal specification of the algorithm, the parser must generate new

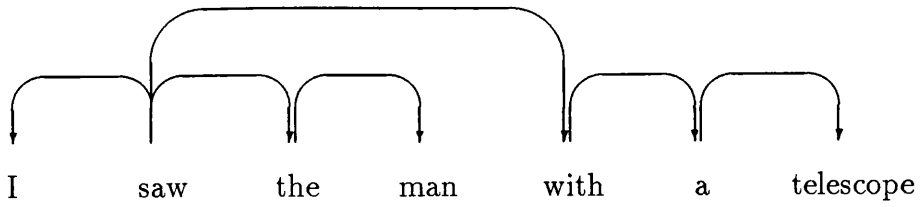


Figure 9.13: *with a telescope* depends on *saw*

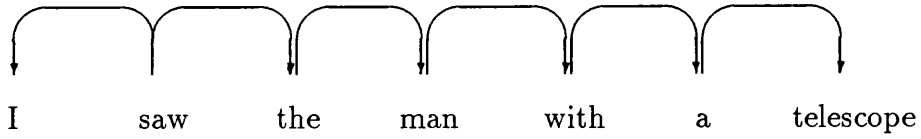


Figure 9.14: *with a telescope* depends on *the man*

identifiers during the parse to cope with cases like this. Needless to say, two instances sharing the same position in the sentence may not enter into any dependency relationship with each other whatsoever.

By means of the above naming convention, all possible readings for the sentence are generated breadth-first. The parser consequently runs rather slowly but, given its academic rather than engineering motivations, this is not a serious fault.

To date, the parser has been tested with a fairly small grammar but it has been able to handle an impressive range of English constructions. These include a variety of different kinds of complement and adjunct structures, shared dependent (a.k.a. multiple head) structures, negatives, and coordinate constructions, including examples with gapping.

9.4 Summary

Both parsers described here work bottom-up, left to right, with a single pass. Furthermore, both alternate between dependent-seeking and head-seeking.

Table 9.2: main features of Fraser's Word Grammar parser

Search origin	bottom-up
Search manner	depth-first
Search order	left to right
Number of passes	one
Search focus	heads seek dependents; then dependents seek heads
Ambiguity management	chronological backtracking; (early identification of failure)

Table 9.3: main features of Hudson's Word Grammar parser

Search origin	bottom-up
Search manner	breadth-first
Search order	left to right
Number of passes	one
Search focus	heads seek dependents; then dependents seek heads
Ambiguity management	all trees constructed in parallel

This is made possible by the fact that WG words are subcategorized for heads as well as for dependents. The parsers differ in respect of their treatment of ambiguity. My parser aims to produce a first parse as quickly as possible by spotting problems early and backtracking over the shortest possible distances. Hudson's parser is much slower and much more thorough, generating all possible parses breadth-first without the help of a chart.

The main features of my parser are summarized in Table 9.2. Those of Hudson's parser are summarized in Table 9.3.

Chapter 10

Covington's parser

10.1 Overview

In this chapter I describe a dependency parser written by Michael Covington, a research scientist at the University of Georgia, USA. Covington is unusual in that he brings together expertise in classics and history of linguistics with more contemporary interests in artificial intelligence. A comparison of two of his publications, *Syntactic Theory in the High Middle Ages* (Covington 1984) and *Prolog Programming in Depth* (Covington et al. 1987), serves to illustrate his unusual blend of interests.

Section 10.2 presents a brief review of some of Covington's work on mediæval grammar which informs his work in DG. Section 10.3 describes the unification-based grammatical formalism Covington assumes, and Section 10.4 describes his dependency parser.

10.2 Early dependency grammarians

Covington's work in the history of linguistics is more pertinent to the concerns of this thesis than might at first be apparent. Covington traces the origins of DG back to the Modistae, a group of mediaeval grammarians starting with Mærtin of Dacia in the mid 1200s who attempted to make 'modes of signifying' the basis of all grammatical analysis (Covington 1984: 25). One of the most important principles of modistic syntax is that the relation between two words

in a construction is not symmetrical; one of the words is the *dependens*, the other is the *terminans*. Thomas of Erfurt offers a metaphorical definition in his *Grammatica Speculativa*:

> Just as a composite entity in nature consists of matter and form, of which one is actual and the other is potential, in the same way construction in language comes about through the exerting and fulfilling of dependencies. The dependent constructible is the one that by virtue of some mode of signifying seeks or requires a terminus to fulfill its dependency; the terminant is the constructible that by virtue of some mode of signifying gives or supplies that terminus (Covington 1984: 48, Covington's translation).

Superimposed on the dependens-terminans relation is another, the relation of *primum* to *secundum*. Covington notes that

> the relation of *primum* to *secundum* is similar to the basic relation posited by modern dependency grammar, in that the *secundum* presupposes the presence of the *primum* (Covington 1986: 31).

This concern of Covington's with the origins of grammatical theory in general and DG in particular informs his work in parsing. In introducing his parser he ties it to the work of the Modistae:

> In a sense, the algorithm is not new; there is good evidence that it was known 700 years ago. But it has not been implemented on computers [before] (Covington 1990a: 1).

To say that Covington's work is informed by mediaeval grammatical theory is not to say that his parser slavishly follows its dictates. His parser is **not** an implementation of the grammatical theory of Thomas of Erfurt!

10.3 Unification-based dependency grammar

Covington bases his DG on a variation of Miller's 'D-rules' (Miller 1985). Instead of using atomic symbols like N and V he uses feature structures of the

kind that are commonly used in unification-based grammars (Shieber 1986). The following rule:

(78)

$$\begin{bmatrix} category: & X \\ gender: & G \\ number: & N \\ case: & C \end{bmatrix} \leftarrow \begin{bmatrix} category: & Y \\ gender: & G \\ number: & N \\ case: & C \end{bmatrix}$$

indicates that a word of category Y with gender G, number N and case C can depend on a word of category X with gender G, number N and case C. The rule says nothing about word order. By convention the head is always written first. If the feature structure corresponding to some word unifies with the left hand side of the rule and the feature structure corresponding to some other word unifies with the right hand side of the rule, the two words can enter into a dependency relationship in which the head is the word whose feature structure matches the left hand side of the rule.

A simple semantics can be built into this framework as follows:

(79)

$$\begin{bmatrix} category: & verb \\ number: & N \\ person: & P \\ semantics: & X(Y,Z) \end{bmatrix} \leftarrow \begin{bmatrix} category: & noun \\ number: & N \\ person: & P \\ semantics: & Y \\ case: & nom \end{bmatrix}$$

This rule allows subjects to depend on verbs and also ensures that the subject becomes the verb's first argument.

This kind of simple semantics is used to manage optionality and obligatoriness in the grammar. If an argument is obligatory then it is also unique. Once an obligatory argument is found it instantiates a variable in the feature matrix which can not be subsequently reinstantiated. Therefore there can not be multiple matches. If this semantic constraint were not present, the above rule could be used to provide the verb with as many 'subjects' as there were nouns in the sentence. There must be an explicit check at the end of parsing to ensure that no semantic arguments remain uninstantiated. In order to add

optional dependents to a word, the rules relating to these dependents must be written so as to add feature-value pairs rather than to supply values for existing features.

Even variable word order languages place some constraints on order such as the requirement that prepositions precede their nouns. This is handled by marking rules where necessary as 'head first' or 'head last' and requiring the dependents to be ordered accordingly. The grammar and parser Covington describes do not provide a mechanism for handling strict contiguity requirements. Covington proposes a scheme for implementing these by marking the head of the constituent in question with a feature *contig* which would be copied recursively to all its dependents. An explicit check would ensure that all words bearing this feature were contiguous.

10.4 Covington's parser

Covington declares his principal objective in writing his parser to be the investigation of parsing techniques for languages with variable word order and, in particular, languages with discontinuous constituents. The parser is implemented in VM/Prolog on an IBM 3090 Model 400-2VF computer.[1] There is no morphological analyser; all forms of a word are stored in the lexicon. The features used in lexical entries include:

phon the word's phonological or orthographic form;

cat the word's syntactic category;

case, num, gen, pers grammatical agreement features;

id a unique identifier for each word;

dep an open list containing pointers to the feature structures of all the word's dependents.

The parser makes an initial pass through the sentence, looking up each word in the lexicon and replacing the word in the input string with its feature structure.

[1] Covington's paper describing his parser (Covington 1990a) won first prize in the Social Sciences, Humanities and Arts section of IBM's Supercomputing Competition (see *The Finite String* **16**:3, September 1990, page 31; *LSA Bulletin* **129**, October 1990, page 16).

There is no reason why this lexical scan phase should not be interleaved with the linking procedure in an incremental parser.

Two lists are maintained by the parser: 'PrevWordList' which contains all words that have been input to the parser so far, and 'HeadList' which contains only words which are not dependents of other words. At the start of parsing both of these lists are empty. At the end, HeadList should contain a single item, the only word without a head left in the sentence, i.e. the sentence root.

Parsing proceeds by processing each of the words in the sentence in turn, as follows (quoted from Covington 1990a: 19):

Covington's parsing algorithm

1. Search PrevWordList for a word on which the current word can depend. If there is one, establish the dependency; if there is more than one, use the most recent one on the first try; if there is none, add the current word to HeadList.

2. Search HeadList for words that can depend on the current word (there can be any number), and establish dependencies for any that are found, removing them from HeadList as this is done.

```
INITIALIZATION:  read input words into a list;
                 C is the current word in the list;
                 C:=1;
                 initialize two empty stacks: Stack1 and Stack2;
                 push(Stack1, C);
                 C:=2;
                 Root is the result variable;
                 X is a global variable;
                 X:=1.

1. IF C=e
   THEN goto(4)
   ELSE IF X=0
        THEN push(Stack2, C),
             goto(2)
        ELSE IF X → C
             THEN record(X → C),
                  goto(2)
             ELSE X:=X-1,
                  goto(1).

2. IF empty(Stack1)
   THEN goto(3)
   ELSE IF C → top(Stack1)
        THEN record(C → top(Stack1)),
             pop(Stack1),
             goto(2)
        ELSE push(Stack2, top(Stack1)),
             pop(Stack1),
             goto(2).

3. IF empty(Stack2)
   THEN X:=C,
        C:=C+1,
        goto(1)
   ELSE push(Stack1, top(Stack2)),
        pop(Stack2),
        goto(3).
```

4. Root:=top(Stack1),
   pop(Stack1),
   IF empty(Stack1)
   THEN succeed
   ELSE fail.

Algorithm 10.1: Covington's dependency parsing algorithm (no adjacency requirement)

Notice that the parser begins by searching for a word on which the present word may depend and afterwards searches for words which can depend on the present word. This is unusual; for example, my own dependency parser and those of Hudson, and Starosta and Nomura all begin by searching for dependents for the current word and thereafter proceed to searching for a head for the current word. The reason for proceeding in this way is simple. If a word has both a head and a dependent occurring on the same side, the dependent is almost always closer to the word than the head. By searching for the dependent first, the possibility of considering the dependent as a potential head is ruled out. Perhaps this difference is not yet relevant to Covington's system since he has so far tested his algorithm only against data from Russian and Latin, both of which have variable word order and rich case systems.

As the parser stands, it could be expected to produce spurious parses for a fixed order, virtually case-free language like English. Covington claims that his parser could be modified to respect the sort of adjacency required for English by modifying his two algorithm steps as follows:

Modifications to algorithm to introduce adjacency

1. When looking for the word on which the current word depends, consider only the previous word and all words on which it directly or indirectly depends.

2. When looking for potential dependents of the current word, consider only a contiguous series of members of HeadList beginning with the one most recently added.

A PARS description of Covington's modified algorithm is given below.

INITIALIZATION: read input words into a list;
C is the current word in the list;
C:=1;
initialize two empty stacks: Stack1 and Stack2;
push(Stack1, C);
C:=2;
Root is the result variable;
X is a global variable;
Top is a global variable;
H is a local variable
(it is not bound between subroutine calls).

1. **IF** C=e

   THEN goto(5)

   ELSE IF C-1 → C

   THEN record(C-1 → C),
   goto(3)

   ELSE X:=C-1,
   goto(2).

2. **IF** H → X

   THEN IF H → C

   THEN record(H → C),
   goto(3)

   ELSE X:=H,
   goto(2)

   ELSE push(Stack2, C),
   goto(3).

3. **IF** empty(Stack1)
   THEN goto(4)
   ELSE Top:=top(Stack1),
      IF C → Top
      THEN record(C → Top),
         pop(Stack1),
         IF top(Stack1)=(Top-1)
         THEN goto(3)
         ELSE goto(4)
      ELSE pop(Stack1),
         pop(Stack1),
         IF Top(Stack1)=(Top-1)
         THEN push(Stack2, Top),
            goto(3)
         ELSE push(Stack1, Top),
            goto(4).
4. **IF** empty(Stack2)
   THEN C:=C+1,
      goto(1)
   ELSE push(Stack1, top(Stack2)),
      pop(Stack2),
      goto(4).
5. Root:=top(Stack1),
   pop(Stack1),
   IF empty(Stack1)
   THEN succeed
   ELSE fail.

Algorithm 10.2: Covington's dependency parsing algorithm (including adjacency requirement)

Covington's claim is that with these requirements added, the algorithm would be equivalent to that of Hudson (1989c). Certainly, the algorithms are similar in spirit although Hudson's parser can deal with phenomena such as coordination and movement which Covington's can not handle. Links would

not be established in the same order in both parsers since, as I have already pointed out, Hudson's parser searches for dependents first and Covington's parser searches for heads first. This difference is not trivial. In many cases it leads to Covington's parser failing to find an analysis where Hudson's parser succeeds. Consider sentence (80).

(80)

I like blue cheese

When *cheese* is being parsed, the algorithm requires *blue* to be considered as its head. This fails. Since only *blue* and any word on which *blue* depends (in this case none) may be considered as a head for *cheese*, *cheese* must be added to HeadList. Next HeadList is searched in order to find dependents for *cheese*. The only dependent which is found is *blue* so it is removed from HeadList. There are no more words in the sentence so parsing terminates. However, *cheese* has not been linked to its head *like*. The parsing algorithm has failed to find a structure for (80). Having read an earlier draft of this chapter, Covington accepts these criticisms. A modified version of his parser, in which dependents are searched for first, has now been published (Covington 1990b). It appears to work unproblematically.

Covington's main interest, however, is in the version of his parser which has no adjacency constraint. He points out that though his parser is capable of finding discontinuous constituents, it nonetheless 'prefers' analyses in which constituents are continuous. This is because it always begins searching as close as possible to the current word, and works backwards. When an analysis fails, the parser uses Prolog's backtracking facility to 'unpick' what has been built back to the point where the wrong choice was made and then starts building a new analysis. This approach to recovery from failure (it is also the mechanism which produces exhaustive enumeration of all possible readings of the sentence) is computationally expensive since there is no way of preventing backtracking from discarding structure which will have to be rebuilt. In Covington's favour,

it must be said that he presents his parser as a prototype so it is probably too early to criticize it on grounds of implementational inefficiency.

Covington does, however, address the question of the time complexity of his parser. The time required to parse an n-word sentence using the most efficient CFPSG algorithms, is proportional to at most n^3. Covington suggests that the same is true of any dependency parser with an adjacency constraint. He makes his case as follows (quoted from Covington 1988):

1. A dependency parser must attach every word in the sentence (except the main verb) to some other word.

2. Without backtracking, this would require, at most, examining every combination of two words, checking whether a dependency relation between them is possible. There are n^2 such combinations.

3. However, the dependency parser may have to backtrack, i.e., discard attachments already made and replace them with other possibilities. At worst it must repeat all its previous work every time it parses another word, thus introducing another factor of n. Hence the total worst-case time is proportional to n^3.

Inevitably, parsing without an adjacency constraint will be more complex since the search space will be larger. Covington suggests a worst case time proportional to n^n. In defence of a parser with such a high complexity he notes that (i) the complexity is due to allowing discontinuous constituents, not to the use of dependency; (ii) worst case complexity is irrelevant to natural language processing (after all, humans are typically unable to process 'worst cases'); (iii) the complexity can be reduced by putting arbitrary limits on how far away from the current word the search for heads and dependents may proceed.

However, even if we were to accept his observations, it is still the case that, other things being equal, the parser with the lowest complexity is to be prefered over any alternatives. The arguments he offers could be made for

any high-complexity parser so they do not distinguish his parser from others of similar complexity. Neither do they justify the selection of this parser over others of lesser complexity.

Covington acknowledges that coordinate constructions pose a problem for DGs; his parser does not handle them. Since he has placed special emphasis on producing a variable word order parser, it can be argued that he has selected the task to which dependency parsers are best-suited. After all, his parser operates with the minimum of constraints; it spots possible dependency pairs and thereafter has no further constraints to check to see if the words are accessible to each other. The parsing complexity may be high but the algorithmic complexity is low. If, on the other hand, he modifies his parser so that it embodies an extra adjacency constraint, the search space is reduced but the algorithmic complexity is increased. Furthermore — in addition to the problems I have already noted — the adjacency constraint he proposes is not sufficient to allow the parser to produce correct analyses of normal movement phenomena in English. What is required is an adjacency constraint plus some principled way of analysing the small number of discontinuous constituents which regularly occur in fairly fixed word order languages like English.

10.5 Summary

Covington's parser is loosely inspired by the work of the medieval Modistae. It has as its primary objective the parsing of variable word order languages. I have presented two versions of the parser. Whereas the first version has no adjacency constraint at all, the second version does include one. Both parsers implement left to right, bottom-up, depth-first search. They both also establish dependencies by first seeking heads and then seeking dependents. As I have observed, this results in the failure of the version with an adjacency constraint to parse some sentences correctly. Each version yields one parse only, although it is possible to produce all parses by forced backtracking.

Table 10.1: main features of Covington's first two dependency parsers

Search origin	bottom-up
Search manner	depth-first
Search order	left to right
Number of passes	one
Search focus	dependents seek heads; then heads seek dependents
Ambiguity management	chronological backtracking

The main features of both versions of Covington's parser are summarized in Table 10.1.

A more recent version reverses the order of search so that dependents are searched for before heads. This algorithm is virtually identical to that of Hudson, as described in PARS in the last chapter.

Chapter 11

The CSELT lattice parser

11.1 Overview

In this chapter I describe the SYNAPSIS parser developed at the Centro Studi e Laboratori Telecomunicazioni[1] (CSELT) in Turin. This is not the only parser to be produced at CSELT which makes use of the notions of dependency, or at least valency. A system called SHEILA ('Syntax Helping Expectations In Language Analysis') analyzes and 'understands' information from a news agency wire by using a mixture of PSG and DG (Danieli et al. 1987). PSG is used to construct the major phrases of a sentence; DG is used to establish dependencies between major phrases. The rational for this approach is that phrase structure parsing is well-understood and consequently should be used where possible and effective, i.e. in building immediate constituents. However, dependency is useful for linking the major constituents of the sentence because, by and large, syntactic and semantic dependencies are isomorphic. This is claimed to assist in early disambiguation since semantic constraints can be brought to bear immediately a syntactic dependency is postulated. This system bears a striking similarity to Niedermair's divided valency-oriented parser (Niedermair 1986; briefly described on page 64, above).

The object of the SYNAPSIS parser differs significantly from that of all the other parsers described here: it is designed specifically for the purpose of analyzing *spoken* rather than written language. The difference turns out

[1]The research division of the Italian telecommunications company.

to be non-trivial as we shall see in Section 11.2. The parser is described in Section 11.3.

11.2 The problem: lattice parsing

One of the most important differences between spoken language and written language is the markedness of word boundaries. In written language, word boundaries are clearly indicated by the presence of a space. In computer systems it is normal to regard the space not simply as a gap — an absence of writing — but rather, as an explicit boundary marker. In spoken language, while pauses may occur between words, there are no guarantees that this will happen in every case. Rather the opposite is the case: it is normal for words to be run together to the extent that the final segment of a word is coarticulated with the initial segment of the following word. Thus, the speech recognition problem does not consist solely in the identification of what lies between word boundaries; it also requires the hypothesization of the boundaries themselves. If the set of hypotheses is to include the correct segmentation then it is likely to have to contain some alternatives. For example, a short speech signal could be segmented as *I see* or *icy*. Given the limitations of present speech recognition technology, most sentences are analysed in terms of many alternative segmentations and, for the signal chunk between each hypothesized pair of word boundaries, there will be several different word candidates. Far from outputting a single string of words, a connected speech recognizer typically outputs a *lattice of hypothesized paths*, one of which hopefully corresponds to the 'correct' analysis of the sentence. Figure 11.1 shows a very simple lattice based on the two words *I know*. This is a portion of a larger lattice presented in Phillips (1988). In reality, most lattices are likely to be much more complicated than this.

Not all paths through a lattice are equally likely. When a speech recognizer constructs a word hypothesis it weighs the evidence for and against the validity

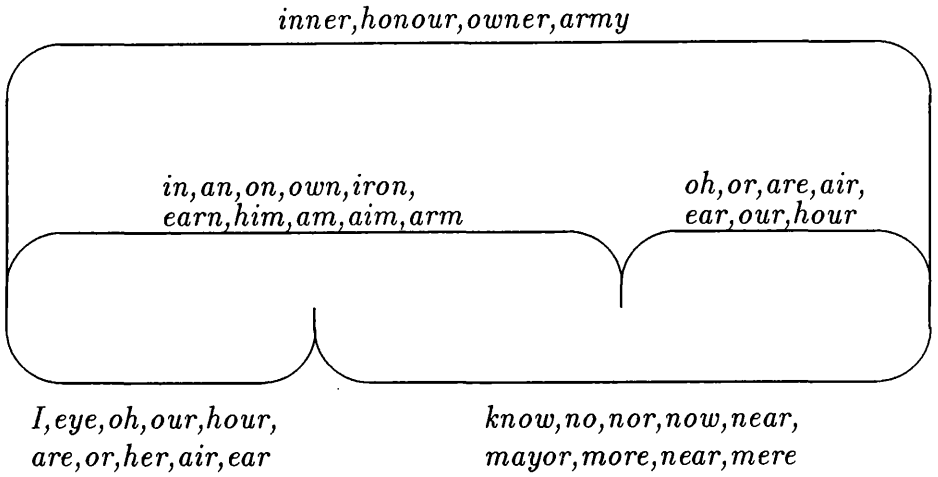


Figure 11.1: a simple lattice for the uttered words *I know*

of the hypothesis and assigns a numeric 'confidence score' to the hypothesis. If the confidence score is greater than some threshold then the hypothesis is entered in the lattice, otherwise it is discarded. Since all words in the lattice have an associated confidence score, it is possible to rank-order paths through the lattice on the basis of confidence scores. In an ideal system operating under ideal circumstances, the highest-scoring path would correspond to the 'correct' analysis. However, there are no guarantees that this will be the case. In fact, there are no guarantees that the 'correct' analysis will be represented in the lattice at all, although parts of it almost certainly will. We shall see in the next section how the SYNAPSIS parser is able to analyze some sentences correctly, even when certain words are missing from the lattice.

Clearly, most of the paths through the lattice will be incoherent at the levels of syntax and semantics. Ideally there will be a single path through the lattice which satisfies the higher level constraints, although the possibility of there being more cannot be ruled out *a priori*. The task of recognizing a spoken sentence should thus reduce to the task of constructing a lattice and

then parsing every path to find the syntactically and semantically coherent one(s).

There is a simple reason why this approach is impractical for most purposes: there are too many possible paths through the lattice. Speech understanding is a real time activity. Most speech interfaces are conceived with the aim of facilitating *rapid* hands-off interaction with a computer. There may simply be insufficient time for all possible paths to be considered (assuming the constraints of state-of-the-art computing technology) if the speech interface is to produce an interpretation within the limits of the desired response time. In the case of 'conversational' computer systems such as the *Sundial* system (Peckham 1991), rapid response may be necessary for other reasons. For example, it has been shown that in everyday human-human conversation, speakers seldom leave unfilled pauses of more than about 1 second (Jefferson 1988). If speaker A asks speaker B a question and speaker B does not respond within the crucial $\approx$1 second period, speaker A will feel compelled to take the initiative and begin a new conversational turn. There are certainly exceptions to this generalization and it is unlikely that the phenomenon transfers exactly to human-computer conversations. However, initial results from 'Wizard of Oz' experiments (Fraser and Gilbert 1991b) in which subjects conversed with a simulated computer, suggest that a related phenomenon can be found in human-computer interactions (Fraser and Gilbert 1991a, Fraser et al. forthcoming). Clearly, a conversational computer must be able to understand an utterance and generate a reply before the human user starts responding verbally to an 'accountable silence'.

The lattice to be searched by a speech recognizer is typically very large indeed. A ten word sentence, analysed as a lattice consisting of ten edges, each having four competing hypotheses, would yield more than a million possible paths. Most speech recognition systems construct lattices containing many more than four hypotheses for each edge. For example, the CSELT speech

recognizer constructs lattices containing approximately fifty times the number of actual words uttered. I shall use a much lower figure to illustrate the nature of the lattice parsing problem. Suppose a ten word sentence is analysed as a lattice containing ten competing hypotheses for each word. This lattice would yield more than ten billion possible paths. Phillips reports that "An actual parser I have used would usually find a parse [for a ten word sentence] after trying a couple of hundred million paths — an average of six or seven words for each position" (Phillips 1988). Assuming it were possible to produce one hundred parses per second for a ten word sentence, it would take about eleven and a half days to produce one hundred million parses!

According to Gazdar and Mellish, "Ambiguity is arguably the single most important problem in NLP" (Gazdar and Mellish 1989: 7). It introduces the possibility of multiple syntactic analyses of parts or all of a sentence. However, the word class or word sense ambiguity which preoccupies most computational linguists and to which Gazdar and Mellish refer, is normally considered from the starting point of a string of distinguished words. When the starting point is a lattice, and the indeterminacy of the acoustic signal is compounded with the indeterminacy of the grammar, the combinatory explosion of possible paths from signal to analysis is alarming.

It is unrealistic to expect a parser to search a lattice and find a solution by 'brute force' within a reasonable time period. The magnitude of the problem precludes the use of such a technique. The CSELT SYNAPSIS parser is an attempt to solve the problem by applying appropriate 'intelligence' rather than 'brute force'. It is an attempt to use the information to be found in the acoustic signal to limit the search space of the parser, and the information contained in the grammar to constrain the search space of the word recognizer. As such, its concerns are different from those of the other parsers I have described and it is not readily comparable with them at an algorithmic level. However, the fact that it is both based on DG and algorithmically innovative makes

it particularly relevant to our present concerns. It also serves to illustrate a promising application of DG in NLP.

11.3 The solution: the SYNAPSIS parser

Section 11.3.1 provides a brief overview of the SYNAPSIS parser. This is followed in Sections 11.3.2 to 11.3.4 by a more detailed examination of the form of syntactic and semantic information used by the parser. Section 11.3.5 describes the basic SYNAPSIS parsing strategy and Section 11.3.6 outlines a suggestion for parallelizing the parser.

11.3.1 Overview of SYNAPSIS

The SYNAPSIS (SYNtax-Aided Parser for Semantic Interpretation of Speech) parser is part of a larger question-answering system for extracting information from a database by means of relatively unconstrained spoken natural language requests (Fissore et al. 1988). The database used during development contained information about the geography of Italy. The earliest references to SYNAPSIS in the literature are dated 1988, although SUSY, the overall speech understanding system (recognizer + parser + generator + synthesizer) of which SYNAPSIS is just one part, is described in Poesio and Rullent (1987).

The principle motivating the design of SYNAPSIS was that syntactic, and indeed, semantic constraints should be brought to bear as early as possible in the interpretation of a lattice. That is, knowledge of syntax and semantics should provide expectations to guide search in the lattice, thus ensuring that syntactically or semantically impossible structures were not considered. The parser had to implement a *top-down* strategy. On the other hand, since it was observed that correct words were usually — though not always — amongst the highest confidence-scoring words, a useful search strategy would be to consider the highest-scoring words first. Therefore, the parser should embody *bottom-up* features as well.

Since the search space is so large, it was considered appropriate to apply as many top-down constraints on search as possible. Semantic constraints, as well as syntactic constraints should be allowed to trickle down. A semantic representation based on *caseframes* (Fillmore 1968) was adopted, for reasons which have as much to do with the specific problems of speech recognition as they have to do with the usual range of issues which confront linguists. The conventional motivations for choosing caseframes concern the requirement that the semantics be formally explicit, descriptively adequate, and compositional. Caseframe semantics satisfy these criteria. A first speech-related motivation is that the semantics be word-based, rather than phrase-based. Since the primitive units in the lattice are words, it is desirable that single words should trigger semantic rules. This is true of caseframes which are associated with single words. (Effectively the caseframe expresses the semantic valency of the word). Another motivation is the desire to "correlate semantic significance with acoustic certainty" (Giachin and Rullent 1989: 1538). It is claimed that caseframes facilitate this because "the header word, being the most 'meaningful' one, tends to be uttered more clearly, and hence is easily recognized with good acoustical score" (*ibid.*). For these reasons, caseframes have been adopted in a number of speech understanding systems (e.g. Brietzmann and Ehrlich 1986; Hayes et al. 1986).

Caseframes encode only semantic slots for a given word, and constraints on the semantic type of each slot-filler. However, it is not sufficient to rely on semantic constraints alone. For example, the semantic caseframe for the verb *put* will indicate that it requires a PATIENT of some material type (i.e. the thing which is 'put') and a GOAL of type LOCATION (i.e. where the PATIENT is 'put'). This says nothing about the realization of these cases. For example, it places no constraints on the relative ordering of *put*, its PATIENT and its GOAL. It is necessary to combine syntactic constraints with semantic constraints in order to maximize the useful information in top-down

predictions.

The way in which syntactic and semantic information is combined is of vital importance. One approach would be to add simple positional features to the case slots, after the fashion of Conceptual Dependency (Schank 1975; Schank and Riesbeck 1981). This would produce a 'semantic grammar'. There are a number of arguments against this way of tackling the problem. Firstly, it misses a lot of syntactic generalizations. Most semantic grammars are written piecemeal with new 'concepts' being added when required and (usually barely adequate) word order features being added to each new semantic entry. There is typically no principaled way of dealing with general types of construction such as relative clauses. Secondly, the grammars are necessarily tied to some semantic domain. A semantic grammar developed in the context of Italian geography could not readily be ported to a stock control application in spite of the fact that many sentence types would be common to both domains. Thirdly, semantic grammars are not readily modifiable since syntactic and semantic constraints tend to be mixed up together in a collection of *ad hoc* rules. (For a discussion of the shortcomings of semantic grammars see Ritchie 1983.)

The approach adopted in SYNAPSIS is to keep a sharp distinction between syntax and semantics during grammar development and to ensure that appropriate generalizations are made within distinct knowledge bases. In this way, formal rigour and consistency can be maintained. When the syntax and the semantics are completed for some phase of the project, they are automatically compiled into a unified framework similar to a feature grammar with mixed syntactic and semantic features. Grishman observes that parsers based on conceptual dependency can be characterized as being "guided by semantic...patterns and then applying (limited) syntactic checks, whereas most parsers are guided by syntactic patterns and then apply semantic checks" (Grishman 1986: 121). SYNAPSIS treats neither syntax nor semantics as primary, but instead it merges the two into a genuinely mixed grammar which

is nonetheless easily portable and modifiable.

Syntax in SYNAPSIS is expressed in terms of DG. The choice is natural, given the adoption of caseframe semantics. Both systems are word-based and both directly encode the notion of a head or governor and a set of dependents or modifiers. I have already observed that syntactic and semantic dependencies are isomorphic in many cases. Hudson's description of WG can be taken as a particularly emphatic expression of a widely-held view amongst DG practitioners:

> The parallels [between syntactic structure and semantic structure] are in fact very close — virtually every word is linked to a single element of the semantic structure, and the dependency relations between the words are typically matched by one of two relations between their meanings: dependency or identity. Moreover, if word A depends on word B, and the semantic relation between them is dependency, then the dependency nearly always goes in the same direction as in the syntax — the meaning of A depends on that of B (Hudson 1990: 123).

In order to avoid terminological commitment to regarding either syntax or semantics as basic, rules containing merged syntactic and semantic constraints are given the neutral name *knowledge sources.*[2]

These knowledge sources are used by the parser in a mixed top-down and bottom up control strategy which embodies the principles of *best-first* search.

11.3.2 Dependency grammar

The DG used in SYNAPSIS is defined as a tuple

$$DG = \{C, R\}$$

in which C is a set of lexical categories and R is a set of rules of the form:

(81)

a $X_0 = X_1, X_2, ..., *, ..., X_n$

b $X_i = *$

[2]Although the term 'knowledge source' is typically associated with blackboard systems, the SYNAPSIS system is not described as such in any of the published accounts I have seen.

$X_i \in C$ and $n \geq 0$.

Standard constraints on sentence well-formedness apply. This is a very slight modification to the Gaifman rule format. Notice that because the grammar is defined as a tuple, rather than as a 4-tuple, it is not possible to refer to specific words in the grammar. This makes it difficult to express the strongest possible predictions which the grammar ought to be able to make, namely predictions of single words. For example, the English verb *depend* requires a nominal subject and a complement which must be the word *on*. This observation is very robust and could be used to direct word recognition with pinpoint accuracy. It would be simple to express the rule in a 4-tuple DG as follows:

(82)

depend = NOUN * *on*

The best that can be done in a DG of the sort used in the SYNAPSIS project is the following:

(83)

DEPEND = NOUN * ON

where 'DEPEND' and 'ON' are classes which each possess exactly one member.

Gaifman format rules and their immediate notational relatives may be appropriate for describing formal languages but, like standard phrase structure rules, they are ill-equipped for making the full range of generalizations relevant to the syntax of natural languages. In order to cope with phenomena such as morphosyntactic agreement, it is necessary to augment the basic rule set. Previous chapters have documented how a popular approach has been to define DGs in terms of complex feature sets which are combinable by unification. The approach adopted in SYNAPSIS is to attach conditions to rules. These conditions take the form of a word class label (which must be present in the rule) followed by arbitrarily many feature-value pairs. Instead of a value, a variable (a character preceded by '?') may be used. Where the same variable is used in two conditions applying to the same rule, coreference is indicated.

For example,

Rule:	VERB =	ART ADJ NOUN * ART ADJ NOUN
Conditions:	VERB:	(PERSON (3)) (NUMBER ?X)
	ART:	(NUMBER ?X) (GENDER ?Y)
	ADJ:	(NUMBER ?X) (GENDER ?Y)
	NOUN:	(NUMBER ?X) (GENDER ?Y)
	ART:	(NUMBER ?Z) (GENDER ?W)
	ADJ:	(NUMBER ?Z) (GENDER ?W)
	NOUN:	(NUMBER ?Z) (GENDER ?W)

The rule and conditions indicate that if the head verb is in the third person, the article, adjective, and noun preceding it must agree with it in number and with each other in gender. The article, adjective, and noun following the verb are not required to agree with it at all but they must agree with each other in gender and number. (This example is appropriate for Italian but not wholly appropriate for English's much sparser agreement system). Published accounts do not make clear how the particular symbol in a rule (e.g. one of the two 'NOUN' symbols) is distinguished in the conditions.

It would be straightforward to convert a grammar expressed in this form into a unification-based representation such as PATR-II (Shieber 1986).

11.3.3 Caseframes

A caseframe represents the semantic valency of a head word. It contains any number of case slots (i.e. θ-roles) and constraints on the types of possible slot fillers. Some slots must be filled; others are optional; they correspond to necessary parts of the state, action, or entity the caseframe describes but they do not necessarily have to be made explicit in linguistic accounts of the state, action, or entity. (This is reminiscent of Wilks' (1875) *Preference Semantics.*

Caseframes in SYNAPSIS are represented in terms of *Conceptual Graphs* (Sowa 1984). A detailed introduction to the conceptual graph notation is unnecessary for the purposes of the present discussion. The example shown

[LOCATED-IN-REGION]
→ (AGNT:Compulsory) → [MOUNT+PROVINCE+LAKE]
→ (LOC:Compulsory) → [REGION]

Figure 11.2: a SYNAPSIS caseframe

in Figure 11.2 should serve to illustrate what a caseframe looks like. (The example is taken from Giachin and Rullent 1989: 1538.)

This indicates that the word whose meaning is identified as '[LOCATED-IN-REGION]' requires an AGENT of type MOUNT or PROVINCE or LAKE and a LOCATION of type REGION. Neither slot may be left unfilled in a semantically well-formed utterance. Notice that both the syntax and the semantics are expressed in declarative formalisms.

11.3.4 Knowledge sources

The dependency rules and the caseframes are not used serially, with one rule set producing an initial analysis which is passed to the other for completion. Instead, the syntactic and semantic rules are combined to form a unified syntactico-semantic grammar in which both types of constraint apply at the same time. In principle, the combining of syntactic and semantic constraints could be done 'on the fly' during sentence processing, thus creating the resources to meet the particular needs of the moment. In practice, this would almost certainly be costly in terms of processing time and it could result in the same combination having to be performed many times during a single recognition session. The obvious solution — and the one adopted in SYNAPSIS — is to pre-compile the syntactic and semantic information into its unified format.

Figure 11.3 shows parts of a syntactic dependency rule. It refers to a present indicative verb with two dependents, one preceding it and the other following it. The following noun must agree in number with the verb. Comments are preceded by ';;'.

Figure 11.4 is a caseframe indicating that the word whose meaning is iden-

```
VERB(prop) =   NOUN(interr-indir-loc) <GOVERNOR> NOUN(subj)
               ;; Features and agreement
               <GOVERNOR> (MOOD ind) (TENSE pres) (NUMBER ?X) ...
               NOUN-1 ...
               NOUN-2 (NUMBER ?X)
```

Figure 11.3: a SYNAPSIS dependency rule

```
[TO-HAVE-SOURCE]
→ (AGNT:Compulsory) → [RIVER]
→ (LOC:Compulsory) → [MOUNT]
```

Figure 11.4: another SYNAPSIS caseframe

tified as '[TO-HAVE-SOURCE]' requires an AGENT of type RIVER and a LOCATION of type MOUNTAIN. Neither slot may be left unfilled in a semantically well-formed utterance.

Combining the semantic information expressed in Figure 11.4 with the syntactic information shown in Figure 11.3 produces the knowledge source (KS) shown in Figure 11.5. (All of these data structures are taken from Giachin and Rullent 1988: 198.)

The 'composition' entry indicates that the syntactico-semantic head, which is of semantic type TO-HAVE-SOURCE, must be preceded by an element of semantic type MOUNT and followed by an element of semantic type RIVER. The first 'constraint' entry states that the head word must be a present indicative verb and that the MOUNT element must be realized as a noun. The

```
;; Composition
TO-HAVE-SOURCE = MOUNT <HEADER> RIVER
;; Constraints
<HEADER> -MOUNT ((H-cat VERB) (S-cat NOUN) (H-feat MOOD ind TENSE
pres...)...)
<HEADER> -RIVER ...
;; Header activation condition
ACTION(TO-HAVE-SOURCE)
;; Meaning
(TO-HAVE-SOURCE ! * agnt 1 loc 0)
```

Figure 11.5: a SYNAPSIS knowledge source

'header activation condition' is a flag to tell the parser how to use the KR. The 'meaning' entry is used to construct the compositional semantics of the construction headed by a verb of semantic type TO-HAVE-SOURCE.

Having examined the knowledge representations used in SYNAPSIS, we are now ready to consider the parsing procedures it uses. Two versions of the parser will be presented: a straightforward sequential parser and a parallel version designed to decrease the amount of time required to produce plausible interpretations of spoken sentences.

11.3.5 The sequential parser

The input to the parser is an entire lattice. In other words, syntactic and semantic constraints are used together to find a plausible path through an existing lattice; they are not used to guide the construction of the lattice. One argument in favour of left-to-right incremental processing is that a real-time system cannot afford to wait until the end of the sentence has been reached before starting to analyse what has been said. The argument advanced by the SYNAPSIS designers is that a real-time system cannot afford to start parsing as soon as the left-hand-side of the lattice has been built since there is always the possibility that word recognition may be locally poor and this would lead to a lot of wasted effort. It is much more prudent, they argue, to wait until the end of the sentence and then begin parsing from the word in the lattice with the highest confidence score. There is a reasonable chance that the highest scoring word will have been recognized correctly, and this allows fairly reliable top-down predictions to be used to guide search in the less well-scored parts of the utterance. Their claim is that this non-linear incremental process results in a quicker and, more importantly, a more reliable result than would be produced by a left-to-right analysis. It is worth flagging one problem with the CSELT approach, namely the fact that identifying the end of a spoken utterance is a non-trivial task. Full stops are not typically vocalized! It is possible to imagine a number of heuristics which might be useful, such as timing pauses

against some threshold, or arbitrarily insisting that a sentence may not exceed n seconds in duration. However, none of these would be foolproof. It is not clear how SYNAPSIS copes with this problem.

What I have just outlined is a *best-first* parsing strategy which begins with the highest-scoring word hypothesis and uses it to generate predictions which can be tested against the next highest-scoring hypotheses and so on. A parser scheduler controls the process by means of a number of operators:

ACTIVATION This operator selects the highest-scoring word hypothesis and finds a KS for which it could be the header. The word hypothesis and the prediction are combined to produce a tree-structured *deduction instance* (DI). The tree structure derives from the fact that the instantiated header has unfilled case slots. One way of viewing DIs is as phrase hypotheses.

VERIFY This operator is used to fill a case slot in the current DI with a word hypothesis.

MERGE This operator is like VERIFY except that it is used to fill a case slot in the current DI with another DI rather than a word hypothesis. In other words, it is used to merge two tree structures.

PREDICTION This operator is used if the current DI is a *fact*, i.e. it has no unfilled slots. If the DI is of type T, then this can be used to instantiate another DI having a slot for a filler of type T.

At least one other operator, SUBGOALING, is available. It is rather more complex since it is used to decompose and rearrange existing tree structures. It is not necessary to be familiar with its action in order to appreciate the general strategy of the parser.

Roughly speaking, parsing proceeds as follows. To begin with, the highest-scoring word is used to construct a DI (ACTIVATION). Next, the empty slots in the DI are used to generate predictions. For example, a slot may

require a filler which is syntactically a NOUN and semantically a REGION. All the word hypotheses in the lattice are checked using the VERIFY operator. When several hypotheses meet these conditions, the best scoring hypothesis is activated while the others are stored in a 'waiting' list until such time as the current score is worse than their score. In the meantime, the best-scoring hypothesis is used as the filler for the relevant case slot.

When a word is used to create a new DI, the word's confidence score is assigned to the DI where it is known as the *quality factor* of the DI. When a word is added to an existing DI, the confidence score of the word hypothesis and the quality factor of the DI are combined to produce a new quality factor for the DI. The best way to compute this new quality factor is an open research question. Versions of SYNAPSIS have been tried out which calculate quality factors on the basis of joint probabilities (i.e. the sum of the word hypotheses scores), and of score density (with or without shortfall) (Woods 1982).

Once there is at least one DI available in the system, control passes back and forth between *deduction* and *activation* cycles. Deduction starts from the highest-scoring DI and tries to extend it in the following ways:

1. if it is a fact DI (i.e. it has no empty slots), by making it the filler for a slot in another DI (PREDICTION), or
2. finding a filler in the word lattice for one of its case-slots (VERIFY), or
3. merging it with another DI (MERGE).

The highest-scoring candidate is always chosen first, whether it is a DI or a word hypothesis. When the best DI has a quality factor worse than the best word hypothesis, the activation cycle begins and a next highest-scoring word in the lattice is extracted and used to construct a new DI (ACTIVATION).

The parse is complete when a single DI with no unfilled compulsory case slots covers the same time period as the entire lattice.

The parser is described as following a best-first search strategy but the (available) SYNAPSIS literature does not indicate whether a depth-first or a

breadth first strategy is adopted at choice points with no measure of 'goodness' available to guide the choice. For example, it is not clear from the literature how indeterminacies caused by lexical ambiguities are resolved. One solution might be to rank order knowledge sources having the same header type and to insist that the highest-ranking knowledge source be used first. Taking this suggestion further, knowledge sources having the same header type could be assigned probabilities relative to each other (established on the basis of corpus analysis). These probabilities could potentially be used to weight lexical confidence scores in the computation of quality factors.

The way in which SYNAPSIS constructs analyses bears a certain similarity to the method of the HWIM system (Woods 1982) which builds an 'island' around the highest-scoring word in the lattice. The crucial difference is that the SYNAPSIS system does not require phrases to be contiguous, whereas a standard island parser does.[3] Presumably the possibilities for building spurious discontinuous constituents are less for Italian than for English because of the additional explicit morphosyntactic agreement in Italian. More importantly, the co-presence of semantic constraints and syntactic constraints ought to rule out most of the spurious discontinuities a purely syntactic parser would allow. Desired discontinuities (e.g. questions, topicalizations) would be parsed without difficulty.

Jollies

Function words cause serious problems for all speech recognition systems. Because most function words are both short and typically unstressed, they are often not recognized at all. If the function words are not recognized they are absent from the lattice. This can cause problems for parsers when they try to build constructions which require function words. Even if the presence of a function word is spotted, it may be very difficult to identify which function

[3]Perhaps SYNAPSIS should be termed an 'archipelago parser' rather than an 'island parser'.

word it is. In general, the longer a word is, the easier it is to identify with confidence. The shorter a word is, the harder it is to recognize. So, for example, it is much easier to recognize *hippopotamus* than *an* (which could be confused with *on, and, a, at, ant,* etc.).

In the SYNAPSIS system, words which are considered to have only a functional role are known by the charming name *jollies.* A robust speech parser ought to be able to proceed without jollies in most cases. On the other hand, it ought to be able to find them if they are present in the lattice since some jollies make a useful contribution to parsing. For example, if they are recognized they help to ensure that the correct path through the lattice is temporally coherent. Not all jollies are short, and some may have good confidence scores associated with them, so it is desirable to use them when they are available.

In SYNAPSIS, "the general philosophy is to ignore a jolly unless there are substantial reasons to consider it" (Giachin and Rullent 1988: 199). All jollies are treated as terminal slots in their KS. There may be syntactic or even semantic constraints on them but they do not contribute to the compositional semantics. Since they are assumed to have no semantic predictive power, jollies are not available for manipulation by the standard operators. Instead, a special operator, JVERIFY is used specifically for the purpose of filling jolly slots.

The operation of JVERIFY depends on the JOLLY-TYPE of a jolly slot. There are three JOLLY-TYPES: SHORT-OR-INESSENTIAL, LONG-OR-ESSENTIAL, and UNKNOWN. The type of the jolly slot is worked out during parsing on the basis of "the lexical category assigned to the jolly slot, the temporal, morphologic and semantic constraints imposed on that slot by other word hypotheses, and the availability of such data" (*ibid.*).

If the jolly is of type LONG-OR-ESSENTIAL, it must be found in the lattice. Failure to find it will result in the parse failing just as though it were a content word which were missing.

If the jolly is of type SHORT-OR-INESSENTIAL, it is ignored. That is,

the lattice is not searched in order to find it. However, it is necessary to assign a short time period to the slot, just in case a jolly is present. If this time period were not inserted, the correct path through the lattice would not be temporally coherent. To allow for a range of durations the time period is given fuzzy boundaries.

If the jolly is of type UNKNOWN, it is treated much as though it were of type SHORT-OR-INESSENTIAL, but this is followed by a brief search of the lattice to see if any jollies of greater duration than the maximum dummy duration can be found in the lattice. This is done just in case a long jolly with a good confidence score is present. If one is found, it is entered in the slot; otherwise the dummy is left in place.

Figure 11.6 shows a simplified DI, based on the KS in Figure 11.5 and the sentence *Da quale monte nasce il Tevere?* ("From which mountain does the Tiber originate?"). (The example is taken from Giachin and Rullent 1988: 198.) The DI shows *nasce* as root, with *monte* and *Tevere* as slot fillers. *Monte* has two slots, neither of which has yet been filled. The SPEC slot will eventually be filled with *quale*, while the JOLLY slot corresponding to *da* may remain unfilled unless it is judged to be of type LONG-OR-ESSENTIAL. *Tevere* also has a JOLLY slot but the jolly has already been classified as 'missing'. Notice that this does not necessarily mean that it is absent from the lattice, although that may be the case. What it means is that the jolly has been judged to be superfluous to requirements.

Statistics

The sequential SYNAPSIS parser was implemented in Common Lisp. It makes use of around 150 KSs and has a 1011-word lexicon. No details of the linguistic coverage are available, although the grammar is said to have a branching factor of about 35. SYNAPSIS was tested on 150 lattices produced from normally intoned continuous speech recorded in an office environment. Overall,

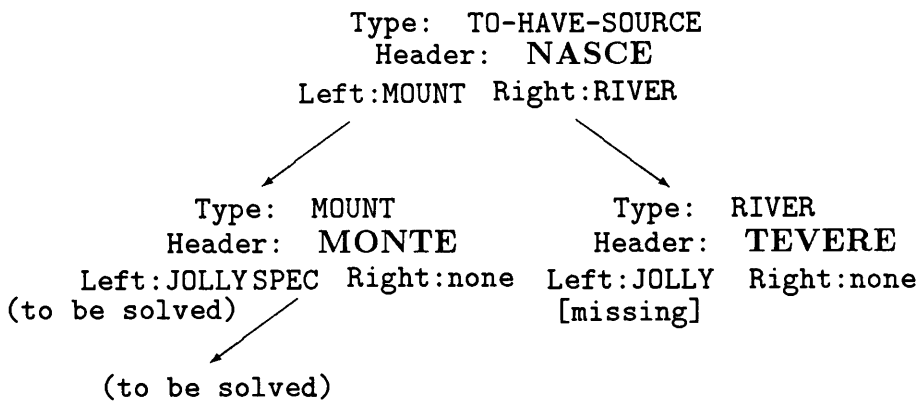


Figure 11.6: a simplified DI showing jolly slots

about 80% of the utterances were analyzed correctly. About 75% of lattices with missing jollies were analyzed correctly. This figure did not increase significantly as the number of missing jollies per utterance increased. Thus, the SYNAPSIS parser may be judged to be a very successful lattice parser by current standards.

11.3.6 The parallel parser

A crucial factor in parsing spoken language is processing speed. The sequential version of SYNAPSIS took an average of about 40 seconds to parse sentences in the test set. (The average sentence length was 7–8 words). This is clearly too long for most practical purposes. In response to the need for better speed results, the developers of SYNAPSIS implemented a parallel version of their parser. While a full exposition of its detail would be inappropriate here, it is worth mentioning a few of its main features.

The sequential parser is based on a processor which uses the KSs to build DIs. The parallel parser consists of n processors called *distributed problem solvers* (DPSs), each with the full inferencing capabilities of the sequential parser. However, each DPS only has access to a subset of KSs. Thus, each DPS can be viewed as the expert on a small number of syntactico-semantic

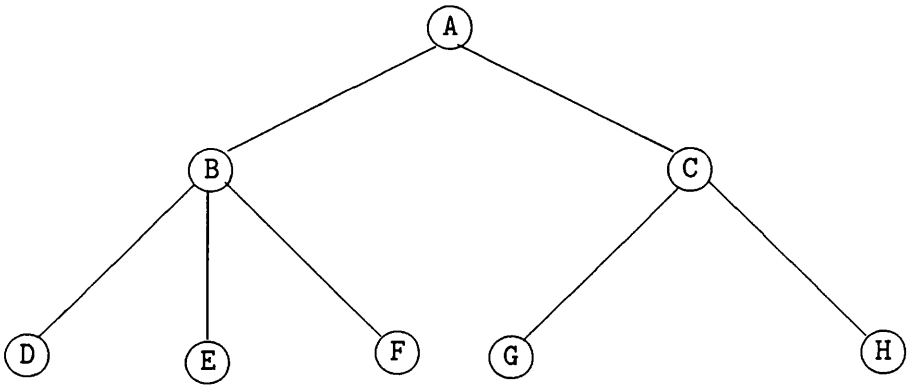


Figure 11.7: a single parse tree

constraints. In most cases it will be necessary for the experts to collaborate in order to solve a parsing problem.

Distributing the knowledge base does not automatically yield a speed-up. If anything, the opposite could be expected since there is now a communications overhead. To effect a speed-up it is also necessary to distribute the tasks in such a way that the DPSs are working concurrently. This is achieved by breaking up the parse trees (i.e. the DIs) into one-level sub-trees. For example, the tree in Figure 11.7 would be represented as a collection of sub-trees, as shown in Figure 11.8. (The representation used here is non-standard. Lines connect lower dependents to higher heads. The reason for employing this graphical device is similar to that which motivated the use of non-standard trees in the DLT project, namely the need to represent dependency structure independently of word order. Each node in this tree represents a single word. Left to right ordering is not significant.)

Since a parse tree (DI) can now be distributed amongst several DPS, it is possible for different parts of it to be developed concurrently. For example, one processor might know about KSs of type MOUNT while another knows about KSs of type RIVER. The left and right branches of the DI shown in Figure 11.6 above could now be grown in parallel.

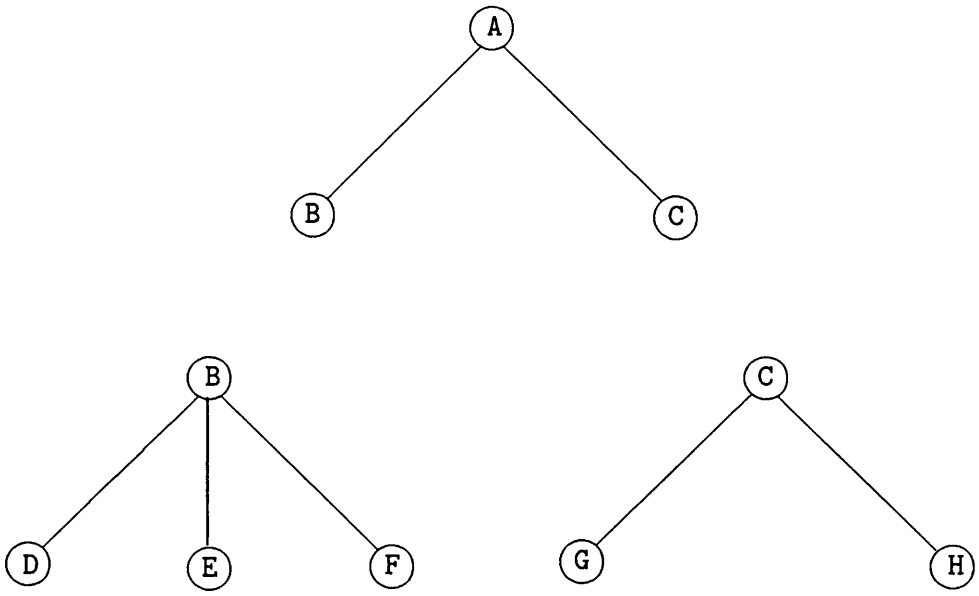


Figure 11.8: a distributed representation of the same parse tree

The parallel version of SYNAPSIS has been implemented on a pool of Symbolics Lisp Machines communicating via Ethernet. The system has been shown to work but the relatively slow Ethernet is a major hindrance to recording significant speed-ups. In fact, no parsing speeds for parallel SYNAPSIS are reported in the literature. The designers have signalled their intention to implement the parser on a Transputer-based distributed architecture.

This sketch of the parallel version of SYNAPSIS has necessarily been brief. More details can be found in Giachin and Rullent (1989).

11.4 Summary

SYNAPSIS is unique amongst the parsers reported in this survey in addressing the distinctive problems of parsing spoken, rather than written language. Instead of starting from an input with distinguished words it is necessary to start from a mesh of alternative hypotheses which may not even include all of the words uttered. SYNAPSIS uses a language model based on DG at the

syntactic level and caseframes at the semantic level. DG builds on the notion of *lexical combination*; caseframes build on the notion of *lexical concept combination.* In order to bring together syntactic and semantic constraints at parse time, DG rules and caseframes are combined to form syntactico-semantic knowledge sources. It is also possible to conceive of the two being brought together elegantly in a unification DG. For example, a unification-based version of Lexicase, with its battery of syntactic, semantic, and case features would provide a theoretically motivated base for a SYNAPSIS-type parser.

The special requirements of speech parsing have led to the development of a parallel version of the SYNAPSIS parser. This also marks SYNAPSIS out as unique in this survey of DG parsers.

I have not provided a formal PARS description of the SYNAPSIS parsing algorithm. PARS is designed for the description of text parsers and would have to be extended significantly to do justice to SYNAPSIS. The purpose of expressing algorithms in PARS is to facilitate comparison of different dependency parsing algorithms. SYNAPSIS is *so* different from the other parsers that a PARS description would not be of much assistance. This difference is underlined by the observation that although the other parsers differ in respect of the order in which they construct parse trees, each individual parser is only capable of constructing a parse tree in one order for each grammar. If SYNAPSIS were to be used to parse several different utterances of a test sentence, it would most probably add branches to its parse tree in a different order each time. This is because SYNAPSIS is strongly guided by acoustic confidence scores, as well as by the grammar.

In spite of the differences, an examination of SYNAPSIS is profitable in serving to illustrate some novel ways in which DGs can be applied in the solution of practical problems.

The main features of the serial SYNAPSIS parser are summarized in Table 11.1.

Table 11.1: main features of the SYNAPSIS dependency parser

Search origin	bottom, then mixed
Search manner	best-first
Search order	score-driven
Number of passes	one
Search focus	heads and dependents seek each other
Ambiguity management	best-scoring parse only

Chapter 12

Elements of a taxonomy of dependency parsing

> Let the teacher, or the man of science who does not always fully appreciate grammar, consider for a moment the mental processes a boy is putting himself through when he parses a sentence, and he will see that there is in intelligent and accurate parsing a true discipline of the understanding. (Laurie 1893: 92)

In this chapter I examine a number of dependency parsing parameters to see how they compare with the corresponding parameters of PSG parsing. In so doing, I outline the elements of a first general taxonomy of dependency parsers.

My approach is driven by the results of the preceding survey of existing dependency parsers. The parameters I shall investigate are those I have used in summarizing the properties of each parser surveyed, namely *origin of search* (Section 12.1), *manner of search* (Section 12.2), *order of search* (Section 12.3), *number of passes* (Section 12.4), *focus of search* (Section 12.5), and *ambiguity management* (Section 12.6). In addition, I shall examine the role of the adjacency constraint in dependency parsing (Section 12.7).

12.1 Search origin

In PSG parsing, search can proceed bottom-up, top-down, or some mixture of the two. At a coarse level, the same is true of dependency parsing. Table 12.1 records the origin (and direction) of search for each of the parsers surveyed.

Table 12.1: origin of search—summary

DEPENDENCY PARSER	SEARCH ORIGIN
Hays (bottom-up)	bottom-up
Hays (top-down)	top-down
Hellwig (PLAIN)	bottom-up
Kielikone (ADP)	bottom-up
DLT (ATN)	top-down
DLT (probabilistic)	bottom-up
Lexicase (Starosta)	top-down
Lexicase (Lindsey)	unspecified
WG (Fraser)	bottom-up
WG (Hudson)	bottom-up
CSELT (SYNAPSIS)	bottom, then mixed
Covington (1 & 2)	bottom-up

Do the familiar terms 'bottom-up' and 'top-down' have their usual meanings when applied to dependency parsers?

A first answer must be 'yes'. Bottom-up parsing starts from the words in a string and uses a grammar to combine the words into constructions. In the case of PSG, the constructions are phrases; in the case of DG, the constructions are head-dependent relata. Top-down parsing starts from the rules in a grammar and attempts to find realizations of structures generated from the rules in the string.

A second answer, however, must be 'no', the terms do not mean *exactly* the same for dependency and constituency parsers. Whereas in PSG there may be a tree of arbitrary depth between a grammatical start symbol at the 'top' and the word instances at the 'bottom', in DG this is not the case. The start symbol of a DG is a word or, at worst, a word class. There are no nodes intermediate between the 'top' (start) node and the 'bottom' (word) node attached to it. The number of nodes in a dependency tree may not exceed the number of words in the sentence whose structure the tree describes. Whereas PSG trees can be arbitrarily deep (unless the PSG is expressly constrained to prevent this), DGs — in just the way indicated — are necessarily shallow.

In the three following subsections I shall examine these concepts in more detail.

12.1.1 Bottom-up dependency parsing

It is immediately noticeable that the majority of the parsers listed in Table 12.1 search bottom-up, i.e. eight out of twelve, with one unspecified. This probably reflects the general word-centred view adopted by dependency grammarians.

A bottom-up PSG parser attempts to take some contiguous group of words and replace them by a single phrase; a bottom-up DG parser attempts to take a group of words (in many cases, exactly two words), and replace them by whichever word is deemed to be the head. Thus, both kinds of parser effect a reduction. Since a DG parser can only effect reductions by discarding one or more words while retaining a lexical head, there is a strict upper bound on the number of reduction steps required (ignoring any requirements for back-tracking), i.e. n–1, where n is the number of words in a sentence. No such upper bound can be placed on a PSG parser, unless the rules the grammar uses are restricted so that, for a rule of the form $\alpha \rightarrow \beta$, where α and β are non-terminals, $|\beta| > |\alpha|$.

Shift-reduce bottom-up dependency parsing

A shift-reduce dependency parsing algorithm can be defined as follows:

1. Let G be a DG in Gaifman format, except that '*' in the body of each rule is replaced by the symbol corresponding to the head of the rule.
2. Let S be an input string.
3. Shift a word from S onto a stack unless S is empty, in which case go to step 5.
4. Check whether one or more words at the top of the stack exactly matches the body of one of the rules in G. There are two possible outcomes:

(a) they match, in which case pop the matching words off the stack and push the word which matched the head of the rule back onto the stack. Repeat step 4.

(b) they do not match, in which case repeat step 3.

5. If there is exactly one element on the stack then succeed, otherwise fail.

This is essentially the algorithm implemented in Prolog in the file `shift_reduce.pl` in Appendix A.3.

In cases where there are equivalent PSG and DG analyses of a sentence, the number of reductions required is identical for shift-reduce parsers of both varieties.

PSG		**DG**
$\bar{V} \rightarrow \bar{N}\ V\ \bar{P}$	[1]	v(n,*,p)
$\bar{N} \rightarrow A\ N$	[2]	n(a,*)
$\bar{P} \rightarrow P\ \bar{N}$	[3]	p(*,n)
	[4]	*(v)

PSG and DG analyses of the sentence *Tall people sleep in long beds* are shown in Figure 12.1.

The PSG shift-reduce parse trace is given below. Word class assignments are not shown. The number of each rule used to effect a reduction is given in square brackets. ('□' indicates the bottom of the stack.)

□ A
□ A N
□ $\bar{N}$ [2]
□ $\bar{N}$ V
□ $\bar{N}$ V P
□ $\bar{N}$ V P A
□ $\bar{N}$ V P A N
□ $\bar{N}$ V P $\bar{N}$ [2]
□ $\bar{N}$ V $\bar{P}$ [3]
□ $\bar{V}$ [1]

The DG shift-reduce parse trace is given below.

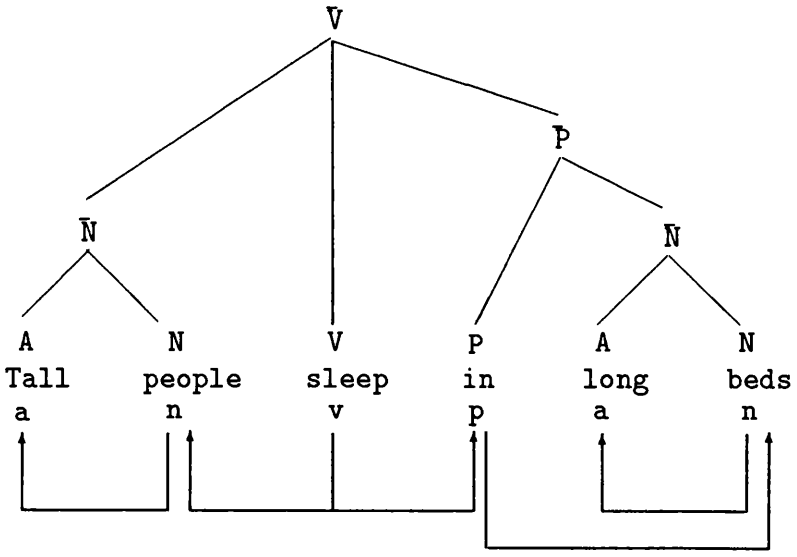


Figure 12.1: PSG and DG analyses of the sentence *Tall people sleep in long beds*

- a
- a n
- n [2]
- n v
- n v p
- n v p a
- n v p a n
- n v p n [2]
- n v p [3]
- v [1]

In this case, the numbers of shift and reduce operations are identical for PSG and DG systems. The number of shift operations is fixed for all versions of shift-reduce parsing, i.e. it is equal to the number of words in the sentence being parsed. The smallest number of reduce operations possible for any sentence is also the same, in principle, for PSG and DG, namely 1. This is because it is possible either to make all words in a sentence belong to a single phrase, or to make all words in a sentence depend on a single head. The maximum number of reduction operations is also the same for PSG parsing and DG parsing, with one important exception which I shall describe shortly.

In PSG parsing, the number of reductions equals the number of phrases. The maximum number of phrases for an arbitrary sentence is achieved with a binary branching phrase structure tree. The number of phrases in a binary branching tree for an n-word sentence is $n - 1$. In DG parsing, the number of reductions equals the number of primary dependencies in the sentence. (I use the term *primary dependencies* to mean dependencies which are found by search, rather than by derivation from existing dependencies, as in the case of dependent-sharing.) The maximum number of primary dependencies — in fact, the required number of primary dependencies — in an n word sentence is $n - 1$.

This equivalence excludes those PSGs which allow unit rewrite rules, i.e. productions having the form

$$\alpha \rightarrow \beta$$

in which both α and β are single non-terminal symbols. In this case, the maximum number of reductions required is not bounded, given an arbitrary sentence and an arbitrary grammar. I know of no version of DG which would not place an upper bound of $n - 1$ on the number of reductions.

Incremental bottom-up dependency parsing

The shift-reduce dependency parser I have just described is a reasonably faithful DG version of a well-known PSG parsing algorithm. However, none of the bottom-up DG parsers described in the preceding survey uses this kind of algorithm. The shift-reduce dependency parser is required to wait until all of the dependents of some head are available in a contiguous block at the top of the stack before it can effect a reduction. All of the other bottom-up dependency parsers I have described establish dependency links between heads and dependents as soon as both become available, and independently of any other dependency relations involving the same head. This results in the incremental

building of dependency structures. This process is centred on *relations* rather than *constructions.*

I believe that the difference between shift-reduce dependency parsing and incremental bottom-up dependency parsing can be characterized in the following way. In shift-reduce parsing, words (or word class labels or feature structures) are put on the stack and grammar rules are used to license reductions. In incremental parsing, sentence words are used to pick out rules headed by these words and these *rules* are then put on the stack. A slightly more complex general rule is then used to effect reductions. A first characterization of the Rule of Reduction is given below. The rule has 2 clauses, as follows (α and β are arbitrary strings of dependent symbols, including the empty string):

The Rule of Reduction

1. If a rule of the form X(α,Y,*,β) is the top element of a stack and the next element is a rule of the form Y(*), then pop the top two stack elements and push a new element of the form X(α,*,β) onto the stack.
2. If a rule of the form X(*,α) is the top element of the stack and the next element is a rule of the form Y(*,X,β), then pop the top two stack elements and push a new element of the form Y(*,α,β) onto the stack.

If all words in the input sentence have been read and the only rule on the stack has the form X(*) and there is a rule of the form *(X) in the grammar then succeed. Otherwise fail.

A trace of the bottom-up incremental parse of the sentence *Tall people sleep in long beds*, using the same grammar as before, is presented below. Stack items are separated by means of the '|' marker. The bracketed numbers indicate which clause of the Rule of Reduction has been applied.

□ a(*)	*tall*
□ a(*) | n(a,*)	*people*
□ n(*)	[1]
□ n(*) | v(n,*,p)	*sleep*
□ v(*,p)	[1]
□ v(*,p) | p(*,n)	*in*
□ v(*,n)	[2]
□ v(*,n) | a(*)	*long*
□ v(*,n) | a(*) | n(a,*)	*beds*
□ v(*,n) | n(*)	[1]
□ v(*)	[2]

The similarities with functional application in CG should be readily apparent. Clause [1] of the Rule of Reduction is the dependency correlate of CG backward application and clause [2] of the Rule of Reduction is the dependency correlate of CG forward application.

An implementation of an incremental shift-reduce dependency parser which makes use of the Rule of Reduction can be found in `incremental_shift_reduce.pl` in Appendix A.3.

In conclusion, DG provides a framework which is compatible with both PSG-style shift-reduce parsing (in which the DG formalism provides an upper bound on the number of reductions which the PSG formalism does not) and CG-style (weakly incremental) shift-reduce parsing.

12.1.2 Top-down dependency parsing

The less explored terrain of top-down dependency parsing offers several interesting divergences from PSG parsing.

Three of the parsers in the survey of DG parsers are classified as top-down parsers in Table 12.1. These each implement top-down search in distinct ways which I shall call *deep top-down parsing, shallow top-down parsing,* and *category-driven top-down parsing.*

Deep top-down parsing

The DLT ATN parser is an example of a deep top-down dependency parser. Parsing is successful if it is possible to traverse the main VERB network, using the words of the sentence. The (simplified) main VERB network for English would consist of a start state, a jump arc to the SUBJECT network, a verb arc, a jump arc to the OBJECT network, and a final state. The SUBJECT network could involve a number of jump arcs to other networks which could themselves contain jump arcs, and so on. Thus, it is possible for the parser to build quite a deep search tree on the basis of the network before the first word is ever examined. When that word is examined, it has the function of either falsifying the hypothesis developed during the preceding search, or allowing the hypothesis to be developed further.

Abstracting away from the detail of the ATN implementation of this search method, I shall try to show how it might work given a more conventional Gaifman type DG. First though, I shall reconsider non-ATN top-down PSG parsing. A top-down left to right PSG parser begins by selecting the start symbol and expanding each successive left-most symbol until a terminal is encountered. Either this matches the first word of the sentence or the hypothesis has been falsified and another must be tried.

For example, Figure 12.2 shows a PSG analysis of the sentence *A cat sleeps on the computer.* A top-down PSG parser would begin by selecting the start symbol (S) and seeing how it could be expanded (S $\rightarrow$ NP VP). It selects the left-most symbol (NP) and finds an expansion for it (NP $\rightarrow$ Det N). Once again, the left-most symbol is selected but this time it corresponds to a terminal. (For ease of exposition I ignore the distinction between words and word classes.) Now, for the first time, it is possible to establish contact between the hypothesized structure and the actual words of the sentence. An examination of the first word in the sentence reveals that it is a determiner, so the hypothesis may be extended with the expansion of the next left-most symbol, and so

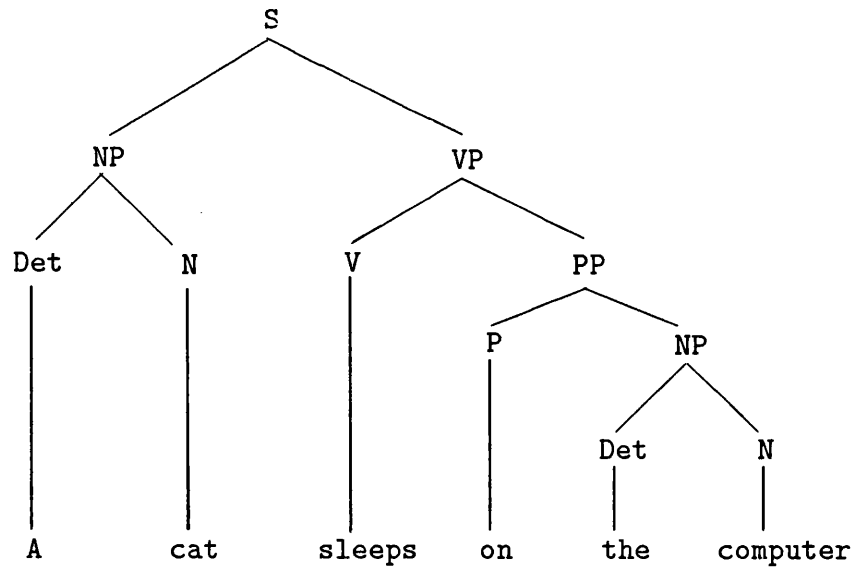


Figure 12.2: phrase structure of *A cat sleeps on the computer*

on.

If this process is used directly with a DG, problems are encountered. The start symbol (v) corresponds to a terminal (*sleeps*), but this word is not located at the start of the sentence. There are two possible courses of action here. Either the parser can look for a rule to expand for the start symbol, or it can search for the start symbol in the sentence. In this section I shall explore the first course of action, and in my discussion of shallow top-down parsing I shall explore the other.

Assume that the grammar contains a rule

(84)

v(n, *, n)

The left-most symbol can be selected. Like all symbols in a DG, this one must identify a word. Thus, it is necessary to see whether or not this matches the first word in the sentence. Since it does not, it is necessary to find a rule headed by n, e.g. (85).

(85)

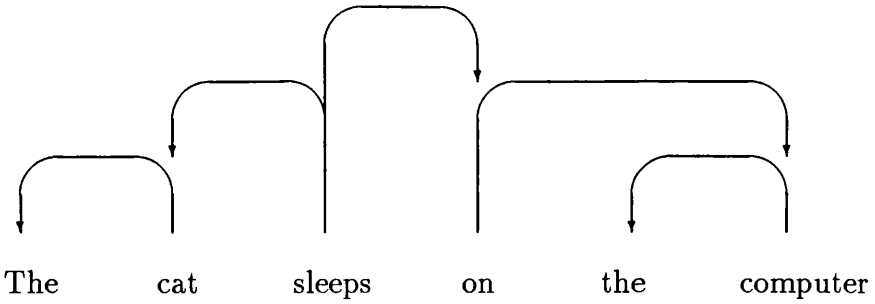


Figure 12.3: dependency structure of *A cat sleeps on the computer*

n(det, *)

Once again, the left-most symbol must be compared with the first sentence word. This time a match is found. It is still necessary to check whether or not 'det' may occur without left side dependents, before going any further. If it can (it can), then it is necessary to try to find the next left-most dependent. This involves selecting the left-most of det's right side dependents (if it has any) and then expanding leftward once again, testing each expansion against the first headless word in the sentence.

Figure 12.3 shows the dependency structure of the sentence. Before word 1 can be parsed, it is necessary to hypothesize word 3 and word 2 (although their actual position in the sentence as words 3 and 2 is not known until parsing successfully completes). Since this parsing method builds a structure of arbitrary depth before it finds a sentence word, I call it *deep top-down parsing.* This method of dependency parsing has not previously been described in the literature, although it is closely related to top-down PSG parsing. Unfortunately it carries an overhead not found in top-down PSG parsers, namely the necessity to check each left-most symbol against the first sentence word after every expansion.

A right to left variety of deep top-down dependency parser could also be defined.

It is possible for a deep top-down parser to enter a loop from which it can not escape. The following rules illustrate this, assuming that a right to left deep top-down parser is being used.

(86)

a p(*,n)
b n(*,p)
c n(*)

When searching for an 'n', the first 'n'-headed rule the parser encounters tells it to hypothesize a 'p' (86b). In order to find a 'p' it is necessary to hypothesize an 'n' (86a). So the loop is entered. Since the maximum number of heads possible in a dependency structure is equal to the number of words in the sentence minus one, the length of the hypothesized path may never exceed this number. This test can be used to terminate fruitless searches, whether caused by looping or some other reason.

Shallow top-down parsing

All top-down PSG parsing is 'deep' in the sense I have indicated. However, as I shall show in this section, it is possible to define a 'shallow' top-down dependency parser.

Shallow top-down parsing also begins by selecting the start symbol (i.e. the root symbol). Assume that the start symbol is v and once again the sentence to be parsed is *A cat sleeps on the computer.* Starting from the left, the sentence is scanned in order to find a v. When *sleeps* is reached there is a match. This word becomes the hypothesized sentence root. Now the grammar is searched for a rule headed by v. If one is found (e.g. v(n, *, n)), the left-most dependent is selected and the part of the sentence prior to *sleeps* is searched. This process continues recursively until the first word is found and there is no more left context to search. At this stage, the most deeply embedded right context is selected and searched, once more from the left. When all words in that right context are accounted for, control passes back to the next most

deeply embedded process which has a right context to search. In this way all of the words to the left of the root can be parsed. The same process can now begin for the root's right context. Parsing succeeds when heads have been found for all the words in the left and right contexts of the root (and the left and right contexts of all the root's subordinates).

A positive feature of this parser is that it never makes an hypothesis without checking immediately that it is at least lexically plausible. In this way a certain amount of spurious structure-building can be avoided.

The basic operation of the parser is simple: call the parsing procedure **divide-conquer** with inputs S and W, where S is a symbol and W is a word list. Initially, S is the root symbol. For descriptive simplicity I assume here that no word may have more than one preceding and one following dependent. (This is not a PARS description.)

PROCEDURE divide-conquer(S,W)

IF S is in W

THEN call the string to the left of S W′;
call the string to the right of S W″;
search the grammar for left side (L) and right side (R) dependents for S ;
if L exists, call divide-conquer(L,W′);
if R exists, call divide-conquer(R,W″)

ELSE fail.

This description is intended to convey a basic sense of how shallow top-down parsing works by recursively calling the same procedure with a shorter word string to search in each call. For obvious reasons I call shallow top-down dependency parsing 'divide and conquer' parsing. The above description omits a number of details which are necessary to the functioning of the parser. In particular, it fails to describe how the algorithm works when confronted with

rules allowing more than one dependent on each side of the head. A somewhat more complex algorithm is required to deal with this. It functions, when more than one dependent is hypothesized in the same string, by successively applying the basic divide-conquer procedure, with each dependent and the part of the string still unaccounted for serving as inputs on each procedure call. The parse succeeds if each dependent accounts for different parts of the string, and all of the string is accounted for. A Prolog implementation of the full algorithm can be found in the file `divide_conquer.pl` in Appendix A.3,

Suppose that it takes some constant amount of time k to check a word to see whether or not it is the word being sought. In the best case, the word being sought will always be at the start of the string, so the time taken to find each word will be exactly k. The time taken to parse an n-word sentence with an unambiguous grammar is therefore in the order of n in the best case. In the worst case, the word being sought will always be at the end of the search string. Thus, for an n word sentence, it will take kn to find the sentence root. The next time the divide-conquer procedure is called there will be n–1 words to search so this will take time $k(n$–$1)$. In the worst case, the time taken to parse a sentence, given an unambiguous grammar will be:

$$kn + k(n-1) + k(n-2) + \cdots + 2k + k$$

Thus, divide and conquer parsing with an unambiguous grammar takes, at worst, time in the order of n^2.

Now assume that the grammar is ambiguous. In the worst case, any word in a string could be the root of that string. Thus, the time required to find every reading for the sentence is proportional to:

$$kn \times k(n-1) \times k(n-2) \times \cdots \times 2k \times k$$

The time required to find every parse of an n word sentence with an ambiguous grammar is, in the worst case, proportional to $n!$. Presumably this figure can be improved by the use of a chart.

The divide and conquer variety of shallow top-down dependency parsing has not previously been described in the literature, unless this is what Hays intended by his top-down parser. As I noted earlier, Hays provides only an outline sketch of his top-down parser and it is not clear if he ever implemented it.

The attraction of the divide and conquer variety of parser lies not in the serial version of the algorithm, but rather in the parallel version. What the parser does is to take a string, divide it in two, decide what to search for in each half, and then proceed to repeat this process for each half. Once a string has been halved, search in one half can take place independently of search in the other. It is necessary for both searches to succeed in order for the original search to succeed, but otherwise there is no connection between the two. Thus, every time a process divides a string it can activate two new processes, one for each substring. The original process simply has to wait to receive the root of the subtrees describing its left and right contexts, in which case it can succeed and inform the process which created it. Alternatively, one of the processes it spawned will fail to find what it was looking for, in which case the original process will die.

Consider the case of parallel divide and conquer parsing with an unambiguous grammar. Each newly created process is assigned to a processor dynamically. The best and worst case parsing times remain the same. In the best case the word being searched for is always found first (On). In the worst case the word being searched for is always at the end of the string (On^2). However, the *average* time ought to be cut significantly because of the possibility of doing at least some search in parallel.

A number of interesting options exist for coping with ambiguity. In principle, it ought to be possible to assign to each word in the sentence as many processes as there are different readings for that word. Each process would then be required to find all possible dependents for a word, given a dependency rule.

Thus, all possible dependency trees of depth one would be found concurrently. In this approach, time would be consumed mostly in inter-process communication, rather than in search. Much more work needs to be devoted to this problem before any results can be reported.

Shallow top-down dependency parsing, such as divide and conquer parsing, in its capacity to divide a string into two substrings, each with a separate 'things to look out for' list, appears to have no counterpart in PSG parsing.[1]

Category-driven top-down parsing

In describing their Lexicase parser, Starosta and Nomura make the following claim:

> Lexicase parsing is bottom-up in the sense that it begins with individual words rather than some 'root' node S (Starosta and Nomura 1986: 132).

It is true that their parser does not proceed by trying to expand the sentence root. However, it does try to expand nodes which have been designated *a priori*. For example, the first step of their algorithm reads: "Link each preposition by contextual features with an accessible N, V, or P" (Starosta and Nomura 1986: 131). What is this if not an attempt to build all of the prepositional phrases top-down?

In recognition of the fact that this is not standard top-down parsing, and certainly not standard bottom-up parsing, I call it *category-driven* top-down dependency parsing. It works by effecting one-level expansions to designated categories in a designated order, not necessarily starting with the root symbol.

12.1.3 Mixed top-down and bottom-up dependency parsing

The CSELT parser implements a mixed top-down and bottom-up strategy. It begins by selecting a word in the sentence, not on the basis of some distin-

[1]Notice that this kind of parsing has got a lot of similarities to old-fashioned schoolroom parsing: 'first find the main verb, then find its subject and its objects, then...'

guished start symbol in the grammar but rather, on the basis of the recognition confidence score associated with the word. The grammar is then searched to find a rule headed by this kind of word. When a rule is found, it is associated with the word in the lattice. The rule is used to search top-down for dependents for the word. When dependents are found, the cycle repeats itself for each of the dependents of the original word.

The attraction of dependency grammar for mixed top-down and bottom-up parsing is that the distance between 'top' and 'bottom' is so small that opposite search approaches can be interleaved very simply and efficiently.

What each of these top-down, bottom-up, and mixed dependency parsing methods illustrates is the proximity of 'top' and 'bottom' in dependency structures. The start symbol (and every other symbol in the dependency tree) is also a symbol in the string. Here then is potential cause for confusion, even — as we have just seen — amongst designers of dependency parsers. And here, too, is something which clearly distinguishes dependency parsers from PSG parsers. It is this proximity of 'top' and bottom' which makes shallow top-down dependency parsing possible. It may be possible to implement a shallow top-down PSG parser, for example, one which uses an $\bar{X}$ or lexicalized grammar to identify the sentence root and each of its subordinates. However, it is clearly impossible with a conventional CFPSG.

Dependency parsers are tied to the words of the sentence. But, as the deep top-down dependency parser demonstrates, it is possible to ignore this constraint and parse — at least for a while — on the basis of hypothesized, rather than actual words. However, unlike some top-down PSG parsers, a deep top-down dependency parser may never loop indefinitely since every search path which contains more hypothesized symbols than there are actual symbols in the sentence, must be terminated.

The principal differences between the origin of search for conventional PSG parsers and dependency parsers may be summarized as follows:

1. The search path between the start symbol in a PSG and the string to be parsed may be arbitrarily long (unless an additional constraint on the grammar prevents this). In a DG, *the start symbol is an element of the string.* The only search which has to be done is that required to associate a specific instance of a symbol with a general reference to that symbol in the grammar.

2. The only exception to the above generalization obtains in the case of deep top-down dependency parsers which may construct longer search paths involving hypothesized words. The number of hypothesized words is, however, bounded by the number of words in the input string.

3. The co-presence of bottom-up and top-down constraints in actual words, allows dependency parsing search to alternate simply and usefully between proceeding top-down and proceeding bottom-up.

12.2 Search manner

There seem to be no significant differences between manner of search (depth-first *versus* breadth-first) for PSG parsers and manner of search for dependency parsers. Either a parser extends one search path as far as possible (depth-first) or it extends all possible search paths in parallel (breadth-first). The CSELT parser implements best-first search, a variety of depth-first search in which the best-scoring option is selected at each choice point. This too has an exact correlate in PSG parsing. It is to be expected that all other manners of searching problem spaces can also be employed in dependency parsers, e.g. beam search which takes a middle line between depth-first and breadth-first search, selecting a maximum of n paths (the 'beamwidth') to develop in parallel.

Table 12.2 summarizes the manner of search properties of the dependency parsers surveyed.

Table 12.2: manner of search—summary

DEPENDENCY PARSER	SEARCH MANNER
Hays (bottom-up)	depth-first
Hays (top-down)	unspecified
Hellwig (PLAIN)	depth-first
Kielikone (ADP)	depth-first
DLT (ATN)	depth-first
DLT (probabilistic)	breadth-first
Lexicase (Starosta)	breadth-first
Lexicase (Lindsey)	breadth-first
WG (Fraser)	depth-first
WG (Hudson)	breadth-first
CSELT (SYNAPSIS)	best-first
Covington (1 & 2)	depth-first

12.3 Search order

There is a limit to the number of possible search orders for an n word sentence. (By 'search order' I mean the order in which words are considered for inclusion in a sentence structure.) In practice, most parsers implement either left to right or right to left search orders. In the survey of dependency parsers — summarized in Table 12.3 — eight out of twelve parsers operate left to right, with three search orders unspecified. None operates right to left, but I can see no reason in principle why any of these parsers should not be able to search in this way with equal success.

An obvious attraction of searching from left to right is that this is usually the order in which sentences are presented to the parser and it is not necessary to wait until the last word has been typed or spoken before parsing can begin. There is particular interest in left to right parsing when the parser not only considers the words in the order in which they appear in the sentence, but also adds them to the developing syntactic structure in (more or less) that order, thus allowing the sentence to be interpreted incrementally left to right. Interest in incremental interpretation is shared by cogni-

Table 12.3: order of search—summary

DEPENDENCY PARSER	SEARCH ORDER
Hays (bottom-up)	left to right
Hays (top-down)	unspecified
Hellwig (PLAIN)	left to right
Kielikone (ADP)	left to right
DLT (ATN)	left to right
DLT (probabilistic)	unspecified, unimportant
Lexicase (Starosta)	left to right
Lexicase (Lindsey)	unspecified
WG (Fraser)	left to right
WG (Hudson)	left to right
CSELT (SYNAPSIS)	score-driven
Covington (1 & 2)	left to right

tive scientists who believe this to be the way that people process sentences (e.g. Marslen-Wilson and Tyler 1980) and computational linguists who want to build real time speech or language understanding systems.

A strand of research in CG instigated by Mark Steedman has investigated the possibility of combining categories using logical devices called *combinators* (Curry and Feys 1958; Turner 1979). This variety of CG is known as Combinatory Categorial Grammar (CCG) (Steedman 1987). An interesting feature of combinators is that the order in which they apply is unimportant; the result is always the same. This (along with the rules of functional composition and type raising) leads to the possibility of producing a strict left to right word-by-word interpretation of any sentence (Haddock 1987; Steedman 1990). Unfortunately, since combinators may apply in any order, they may apply in every order. This leads to the so-called *spurious ambiguity* problem (also known as the *derivational equivalence* problem): weighed against the advantage of being able to interpret a sentence left to right incrementally is the disadvantage of having to deal with (i.e. fend off) all of the other possible ways of arriving at the same conclusion. Thus, most effort in the development of CCG parsers has been devoted towards trying to solve the spurious

ambiguity problem (Hepple 1987). Different proposed solutions include:

1. Inserting only one of each set of semantically equivalent analyses in a chart (Pareschi and Steedman 1987). This carries an equivalence-checking overhead.
2. Only computing *normal form* derivations (Hepple and Morrill 1989). This carries a normal form checking overhead.
3. Compiling a left-branching grammar out of a CCG (Bouma 1989). This carries an initial compilation overhead, and possibly increases the size of the grammar.

DGs allow what may be termed 'weak incremental interpretation', by which I mean the following: as soon as two words which bear a direct dependency relation to each other become available in a sentence (i.e. as soon as the second word is read), the words can be related and accordingly interpreted. Thus, a subject can be interpreted as a subject and its referent can be interpreted as ACTOR, or whatever, as soon as the verb is encountered. There is no need to wait for the construction of a VP or anything else before interpretation can take place.[2]

All of the surveyed DG parsers which operate left to right with a single pass, support incremental interpretation in the weak sense defined above.

The CSELT SYNAPSIS and DLT ATN parsers embody unusual search orders. The CSELT parser always selects the highest-scoring word to process next, regardless of its position. The probabilistic DLT parser enters edges in a graph and then tries to navigate through the graph. There is no necessity for the edges to be entered in any specific order, and it is easy to imagine edges being added for all words in parallel.

Order of search options appear to be generally the same as for conventional PSGs, with most parsers opting for a left to right approach in practice.

[2]Except in the case of shift-reduce dependency parsers of the sort shown in `shift_reduce.pl` in Appendix A.3.

Starosta and Nomura suggest that the choice of search order should be guided by the prevailing direction of dependencies in the language to be parsed.

> [The Lexicase parser] scans from left to right or vice versa, depending on whether the language is verb-initial, verb medial, or verb final, but in fact it is a mechanism which works from head to dependent rather than primarily from one end to the other. (Starosta and Nomura 1986: 132)

Order of search is not crucial to the correctness of parses produced but it may have a significant effect on parsing efficiency. This also depends on the search focus of the parser. A left to right parser in which heads seek dependents would have to read up to the final word of a sentence in which all dependents precede their heads before it could build any structure. A left to right parser in which dependents seek heads would build almost all structure before reaching the final word. A parser in which heads and dependents seek each other would not be sensitive to variation in the order of search.

12.4 Number of passes

The number of passes made by parsers in the survey, by which is meant the number of times the read head of each parser scans a sentence during the parse, is summarized in Table 12.4.

Nine of the parsers make a single pass through the sentence. Recall that confining the number of passes to one is a prerequisite for incremental interpretation.

The parsers of Hellwig, Lindsey, and Starosta and Nomura all require more than one, and possibly very many passes. Starosta and Nomura's parser is particularly profligate, since it requires at least eight passes on each placeholder expansion cycle and there may be many such cycles. In general, increasing the number of passes increases the inefficiency of a parser (since the same symbols have to be checked many times) and is best avoided.

Table 12.4: number of passes—summary

Dependency Parser	Number of passes
Hays (bottom-up)	one
Hays (top-down)	one
Hellwig (Plain)	at least two
Kielikone (adp)	one
DLT (ATN)	one
DLT (probabilistic)	one
Lexicase (Starosta)	at least eight
Lexicase (Lindsey)	multi-pass
WG (Fraser)	one
WG (Hudson)	one
CSELT (Synapsis)	one
Covington (1 & 2)	one

In respect of possibilities and consequences, varying the number of passes of a DG parser appears to be identical to varying the number of passes of a PSG parser, except where this interacts with certain search focus variables, as the next section will explain.

12.5 Search focus

So far, the main difference noted between DG parsing and PSG parsing is in the nature of the top-down/bottom-up distinction. This section introduces another major difference which I have chosen to discuss under the heading 'search focus'. A discussion of PSG parsers would not contain such a section because it is not generally recognized to be of significance for them.[3]

The basic operation in DG parsing is the establishing of binary dependency relations between words. Suppose that X and Y are two words; there are a number of ways in which they might be considered as candidates to be related by dependency. These differences depend upon what I shall call the 'focus of search'. The parsers surveyed identify eight different foci of search. These are summarized in Table 12.5.

[3]HPSG parsing offers the exception to this generalization.

Table 12.5: focus of search—summary

DEPENDENCY PARSER	SEARCH FOCUS
Hays (bottom-up)	pair-based
Hays (top-down)	heads seek dependents
Hellwig (PLAIN)	dependents seek heads
Kielikone (ADP)	heads seek dependents
DLT (ATN)	network navigation
DLT (probabilistic)	heads and dependents seek each other simultaneously
Lexicase (Starosta)	heads seek dependents
Lexicase (Lindsey)	heads seek dependents
WG (Fraser)	heads seek dependents; then dependents seek heads
WG (Hudson)	heads seek dependents; then dependents seek heads
CSELT (SYNAPSIS)	heads and dependents seek each other
Covington (1 & 2)	dependents seek heads; then heads seek dependents

12.5.1 Network navigation

In network navigation parsers, search is focussed on finding an appropriate next token in the sentence to allow a transition network arc to be traversed. Network navigation parsers are of marginal interest in this context since they focus search on a data structure in the grammar-parser, rather than on the words of the sentence being parsed. The only example of a network navigation parser in the survey is the DLT ATN parser.

12.5.2 Pair selection

Pair selection parsers operate by selecting two words in the sentence to be parsed and consulting a look-up table to find out whether or not a pair of words of the chosen types may contract a dependency relationship. Hays' bottom-up parser is pair-based. He defined the two major operations required in his parser to be 'pair selection' and 'agreement testing'. Pair selection involved selecting an adjacent pair of words. Agreement testing involved looking up a

4000 × 4000 matrix to find out whether or not the words could be linked and, if so, which was the head and which was the dependent.

The focus of search is thus a pair of words. As we shall see, all of the other parsers focus search in a single word.

12.5.3 Heads seek dependents

Dependent-seeking parsers (Hays' top-down parser, the Kielikone parser, the Lexicase parsers of Starosta and Nomura, and Lindsay, my Divide and Conquer parser) always search for dependents for the current word. In the course of searching for a dependent (A) for the current word (B), the word which, in reality, should be the current word's head (C) may be tested to see if it can be a dependent of the current word. The test will fail and search will move on to consider another word. The inverse dependency relationship will not be tested until word C becomes the current word, at which point the original word B will be found as a dependent for C.

Notice that this approach to search is not tied to either top-down or bottom-up processing, as the surveyed systems illustrate. Starosta and Nomura's parser is a category-driven top-down parser; the parsers of Hays and myself operate in a shallow top-down fashion; the Kielikone parser operates bottom-up.

As far as I can ascertain, the same strategy is embodied in Proudian and Pollard's top-down HPSG parser.

> In HPSG it is the head constituent of a rule which carries the subcategorization information needed to build the other constituents of the rule. Thus parsing proceeds head first through the phrase structure of a sentence, rather than left to right through the sentence string. (Proudian and Pollard 1985: 168–9)

12.5.4 Dependents seek heads

Hellwig's parser illustrates the fact that a diametrically opposite search focus also works. In his parser, all search is directed towards finding a head for the

current word. Notice, however, that in his system words do not subcategorize for their heads. Rather, it is necessary to go and look in the subcategorization frames (slots) of other words in order to see if the current word can depend on a word (i.e. can fill another word's slot).

12.5.5 Heads seek dependents or dependents seek heads

As we have seen, the SYNAPSIS lattice parser alternates between top-down and bottom-up processing, according to the current state of the parse and the lattice. It also alternates between searching for dependents (VERIFY and MERGE operations) and searching for heads (the PREDICTION operator). The exact progression from one search focus to the other can not be defined *a priori* since this depends on the recognition confidence scores in the lattice.

12.5.6 Heads seek dependents and dependents seek heads

The DLT probabilistic parser works by searching an annotated corpus for every occurrence of each word in the sentence. A record is made of all of the upward and downward dependency relations in which each word is found to participate. These relations then serve as templates of relations into which each sentence word could possibly enter. Some pairs of templates will be inverse copies of each other, and these select each other during a process analogous to unification. Thus, all words search for all of their heads and dependents, and they do so — at least in principle — simultaneously.

12.5.7 Heads seek dependents then dependents seek heads

The WG parsers written by Hudson and myself begin by searching for dependents for the current word. Once all available (i.e. adjacent) dependents have been found, the focus of search shifts, and a head is sought for the current

word. The insight embodied in this strategy is that, under normal circumstances in a relatively fixed word order language like English, the head of a word does not intervene between that word and its dependents whereas the dependents may intervene between the word and its head.

The rationale for changing the focus of search for the current word is that it allows the parser to construct as much structure involving the current word as could possibly be constructed, given what has been processed so far. In fact, it makes it possible to build structure incrementally in a single linear pass through the sentence. This is not possible with either of the strategies of searching for dependents only or heads only.

The parsers which search for dependents only are Hays' top-down parser, the Kielikone parser, the Lexicase parsers of Starosta and Nomura, and Lindsay, and my Divide and Conquer parser. I have previously described Hays' top-down parser and my Divide and Conquer parser as single pass parsers, but this is slightly misleading since the single pass tracks not from left to right, but from root to leaves of the dependency tree. This point was also made by Proudian and Pollard (1985) and quoted above. I have also described the Kielikone parser as a single pass system but this too disguises some important details. Whenever a dependent can not be found for the current word, search suspends (the currently active schema is pushed on the PENDING stack) and another word becomes current. Thus, while words enter the parser one at a time from the left and there is never any attempt to perform the same operation on the same word more than once, words do not become current in strict linear order from left to right through the sentence. The same word can become current for several non-consecutive periods of time. Without the ability to suspend processing of the current word, the Kielikone parser would not be able to parse most sentences. Both of the Lexicase parsers make many passes through the sentence. The motivations and effect are much the same as for the Kielikone parser, although the Kielikone parser achieves its goal with

much greater efficiency.

Only Hellwig's parser searches for a head for the current word without searching for dependents. Once again, experience shows that this strategy will not work for a single pass parser. Hellwig's parser makes multiple passes through a sentence.

The WG parsers stand in stark contrast to these parsers. By searching first for dependents and then for heads for each word, they are able to parse in a single linear pass from the beginning to the end of the sentence. Once a word ceases to be the current word, it will never become the current word again. Thus, the strategy of seeking dependents and *then* seeking heads for the current word facilitates weak incremental processing interpretation.

12.5.8 Dependents seek heads then heads seek dependents

A similar approach is adopted in Covington's parsers, except that they search for a head for the current word and then for its dependents. I have shown how this strategy, while being perfectly adequate in a parser with no adjacency constraint, fails to work when an adjacency constraint is employed. Covington agrees with this analysis and now advocates searching for dependents before searching for heads (Covington 1990b).

12.6 Ambiguity management

The ways in which the surveyed parsers manage ambiguity is summarized in Table 12.6.

This thesis provides descriptions of a dozen dependency parsers, introduces some new ones and mentions quite a few more in passing. Clearly, a significant amount of effort has been and is being directed towards extending what is known about dependency parsing. However, very little of this effort has yet gone towards developing techniques for managing ambiguity in dependency parsing.

Table 12.6: ambiguity management—summary

DEPENDENCY PARSER	AMBIGUITY MANAGEMENT
Hays (bottom-up)	first parse only (heuristics guide search)
Hays (top-down)	unspecified
Hellwig (PLAIN)	WFST ('phrases' may be discontinuous)
Kielikone (ADP)	chronological backtracking (heuristics guide search)
DLT (ATN)	first parse only
DLT (probabilistic)	highest-scoring parse selected
Lexicase (Starosta)	packing/unpacking
Lexicase (Lindsey)	packing/unpacking
WG (Fraser)	chronological backtracking (early identification of failure)
WG (Hudson)	all trees constructed in parallel
CSELT (SYNAPSIS)	best scoring parse only
Covington (1 & 2)	chronological backtracking

Some information is available on ambiguity management for eleven of the parsers surveyed. Of these, four output at most one parse tree, regardless of how many possible analyses there are for the sentence being parsed. The DLT ATN parser either finds an analysis or fails. It can not undo any incorrect choices which may have led to a dead end in the parsing of an otherwise acceptable sentence. Hays' bottom-up parser also delivers at best a first parse, but it makes use of some simple heuristics in an attempt to make the best choices at each choice point. Both of the other systems which deliver at most one parse have the capability to deliver a larger number. In fact they may build all or most of any possible alternative parse trees. The DLT probabilistic dependency parser selects the parse which has the best global score, which is some function of the corpus-derived 'likelihoods' of all of its component dependencies. The SYNAPSIS lattice parser delivers the parse which is 'best' in respect of its global score, which is some function of the recognition confidence scores of its component words.

The Lexicase parsers embody a novel approach to ambiguity management. In slightly different ways they both package up different readings for a word in

terms of a 'placeholder' or 'master entry' which contains only the intersection of all of the different readings. (Since the grammar is fully lexicalized there is no formal difference between lexical and syntactic ambiguity.) As much structure as possible is built on the basis of the partially specified placeholders/master entries. On each successive cycle, placeholders/master entries are unpacked to form disjoint structures which then re-enter the parsing-placeholder expansion cycle independently. The rationale for this process is that as much common structure as possible should be build in a generic and underspecified parse tree before it is split into some number of disjoint more specific structures. This calls for multiple parser passes, but it is supposed to deliver all readings for a sentence, so this may be tolerable. Unfortunately, no published examples are available of this ambiguity management strategy in operation. I have been unable to re-create it to my satisfaction.

Four parsers use chronological backtracking to undo mistakes and, if required, to generate all possible parses. Both of Covington's parsers make use of Prolog's backtracking facility. The Kielikone parser uses heuristics to guide search so that backtracking on the way to a first parse is minimized. Of course, if all parses are required, the benefit of the heuristics will be lost. My Bonding parser also uses backtracking to undo mistakes and to generate multiple parses. It uses heuristics, not to guide choice in structure-building, but to spot doomed partial parses and so force backtracking as early as possible, thereby cutting down on the amount of effort devoted to developing fruitless paths.

All of these backtracking systems work, but they are far from the state of the art in ambiguity management for PSG parsing.

Hudson's parser builds all possible parse trees in parallel. Again, this works, but it is not a viable engineering solution since the same sub-structures can be built many times over in the course of a parse.

Hellwig's dependency WFST parser has the only system for managing ambiguity in this survey which could form the basis of an efficient solution. WFST

parsing is known to be an effective way of avoiding duplication of effort in finding all possible parses for some sentence. WFSTs have traditionally been thought of as graphs in which edges span contiguous phrases. Hellwig offers a solution to the problem of how to represent discontinuous collections of dependents in a table. However, there is currently no known solution to the problem of how to represent overlapping collections of dependents — of the sort introduced in shared dependent analyses — in a table.

As mentioned in Chapter 2, Hays offers a brief schematic description of a recognition algorithm based on a WFST (Hays 1964: 516-17). A Prolog reconstruction of that algorithm can be found in `hays_recognizer.pl` in Appendix A.3. A parser based on the same principles of WFST usage to minimise search can be found in `hays_parser.pl` in Appendix A.3.

WFST parsers offer a considerable efficiency improvement on most parsers which do not check a data structure of intermediate results before searching. However, even greater efficiency can result if a table is used to record current hypotheses as well as well-formed sub-strings. Such a system is usually known as an *active chart parser* (often abbreviated to 'chart parser'). The same hypothesis may be relevant in several different analyses of the same substring. By recording the hypothesis only once, effort can be saved much sooner than in a WFST in which only complete substrings (the result of chains of hypotheses) are entered. The classic reference on chart parsing is Kay (1986).

What does a hypothesis look like in a standard PSG chart parser? Suppose that 'S → NP VP' is a rule of the grammar. The following hypotheses may be recorded in a chart.

(a) S → .NP VP
(b) S → NP .VP
(c) S → NP VP.

Hypothesis (a) indicates that a sentence (S) consisting of an noun phrase (NP) followed by a verb phrase (VP) has been hypothesized, but no evidence has yet been found to support it. Hypothesis (b) is similar, except that the movement

of the dot in the right hand side of the rule to a position after NP indicates that an NP has been found, thus offering partial support for the hypothesis. The position of the dot at the right extreme of the right hand side in (c) indicates that evidence has been found to support the hypothesis in its entirety: an S consisting of an NP followed by a VP has been found in the string.

Each hypothesis must be associated with a particular substring. It is normal in chart parsing to identify sub-strings as edges in a graph. Thus, the first word in a string is usually identified by the edge which goes from node 0 to node 1; the second word goes from node 1 to node 2, etc. The string consisting of the first three words is represented by the edge which goes from node 0 to node 3. Following Gazdar and Mellish (1989: 194ff), I shall represent hypotheses on edges as follows:

$$< i, j, H >$$

where i is the start node, j is the end node, and H is a dotted rule.

To initialize a chart, an *inactive edge* (i.e. an edge in which the dot is at the extreme right hand side of the rule hypothesis) can be placed in the chart for every word class assignment allowed by the grammar for the words in the sentence.

Search may proceed in a number of different ways. Here I shall mention only one of these. Proceeding bottom-up, the following rule may be applied to introduce fresh hypotheses:

> **Bottom-up rule of PSG chart parsing**
>
> If you are adding edge <i,j,A→W1.> to the chart, then for every rule in the grammar of the form B→A W2, add an edge <i,i,B→.A W2> to the chart. A and B are categories and W1 and W2 are (possibly empty) sequences of categories or words. (Adapted from Gazdar and Mellish 1989: 197.)

The fact that the new edge begins and ends at the same node simply results

from the fact that no part of it has yet been attested in the string.

The way in which hypotheses are developed once they enter the chart is by means of application of what Kay calls the *fundamental rule*:

> **Fundamental rule of PSG chart parsing**
>
> If the chart contains edges <i,j,A→W1.B W2> and <j,k,B→W3.>, where A and B are categories and W1, W2 and W3 are (possibly empty) sequences of categories or words, then add edge <i,k,A→W1 B.W2> to the chart (Gazdar and Mellish 1989: 195).

A version of Gazdar and Mellish's Prolog implementation of a bottom-up chart parser, slightly modified to enable it to run as a single file under Quintus Prolog, can be found in the file `gazdar_mellish.pl` in Appendix A.3. (The reason for its inclusion will become clear shortly.)

We could go about reconstructing the notion of a chart parser in the context of dependency parsing in a number of different ways. In what follows I shall adopt a fairly conservative approach which maximizes similarities with PSG chart parsing. First, let us assume that a dot may be placed in the body of a DG rule with the interpretation that everything to the left of the dot has already been attested and nothing to the right of the dot has yet been attested. Thus the following sample dotted dependency rules are possible.

(a) verb(.noun,*,prep)
(b) verb(noun,.*,prep)
(c) verb(.noun,*,.prep)
(d) verb(.noun,*,prep.)

(Let '*' be a variable instantiated to the same category as the head of the rule in which it occurs.)

Example (a) hypothesizes a verb with a preceding nominal dependent and a following prepositional dependent; no part of the hypothesis has yet gained support. In example (b), the noun has been found, and in (c), the head verb has also been found. In example (d), the dot is at the extreme right hand side of the body of the rule, thus indicating that the whole structure has been

attested and the edge is now inactive.

The bottom-up and fundamental rules of PSG chart parsing can also be given a dependency reconstruction.

Bottom-up rule of dependency chart parsing

If you are adding edge <i,j,A(W1.)> to the chart, then for every rule in the grammar of the form B(A,W2), add an edge <i,i,B(.A,W2)> to the chart. A and B are categories and W1 and W2 are sequences of categories or words.

Fundamental rule of dependency chart parsing

If the chart contains edges <i,j,A(W1,.B, W2)> and <j,k,B(W3.)>, where A and B are categories and W1, W2 and W3 are sequences of categories or words, then add edge <i,k,A(W1,B,.W2)> to the chart.

If these rules of dependency chart parsing are applied, all possible dependency structures (and sub-structures) for an input string can be produced efficiently given a dependency grammar in Gaifman form. The file `nmf_chart.pl` in Appendix A.3 contains an implementation of this kind of bottom-up dependency chart parser. Careful comparison of this file with `gazdar_mellish.pl` will reveal that the two are virtually identical in most respects, and particularly in respect of the core parsing algorithm. The only difference worth noting is that dependency grammar rules of the form X(*) have no direct PSG correlates. They can not be used as the basis for hypotheses — equivalent hypotheses have already been entered in the chart at initialization — so they differ from unit rewrite PSG rules which do generate hypotheses. However, this difference does not interfere with the basic control structure of the parsing algorithm.

We shall return to a discussion of this parser in the last chapter.

12.7 Adjacency as a constraint on search

Most of the parsers surveyed assume an adjacency constraint. The effect of such a constraint is to limit severely the search space of the parser. This is clearly illustrated in the case of parsers like my Bonding parser which only needs to look at the top of a stack. This constraint is also built into most PSG parsers, since phrases are typically contiguous. At the opposite extreme, parsers like Covington's adjacency-free parser — which makes no use of an adjacency constraint — must search anything up to the whole of the rest of a sentence in order to find the word they are looking for.

Systems like Kielikone and Hudson's parser operate within the constraints of an adjacency constraint but use a dummy relation (e.g. 'visitor') to capture an otherwise non-adjacent word (such as an extracted wh-word) and establish a link between it and its actual head. This requires the principles of dependency to be defined so as to allow a word to depend on more than one head or to depend on the same head by means of more than one dependency relation (i.e. the moved word must be related by the dummy relation to one head and by the meaningful relation to that head or another head).

I believe that one of the major strengths of DG is that it makes a number of constraints explicit which are usually implicit in PSG. In this way, it allows the grammar writer and the parser designer to consider each constraint independently and to experiment with different versions of the constraints. For example, Hudson found the adjacency constraint to be too tight for his purposes so he revised it. He is not alone; almost all DG theories and a number of DG parsing systems customize the basic DG mechanism in some ways. Tinkering with the basic constraints of PSG in this way is almost unheard of (although when someone does this it tends to revolutionize the way linguists conceive of problems — witness $\bar{X}$ grammar and GPSG).

I suggest that a potentially fruitful area of research involves refining the adjacency constraint, so as to minimize the search space of a parser while max-

imizing the number of phenomena which can be covered. The strict adjacency constraint built into many of the parsers surveyed is too strict to allow for the parsing of variable word order languages. However, even variable word order languages do not allow clauses to intermingle, so some constraints must still apply. The definition of these constraints is a live research topic.

Hellwig has taken an interesting step in exploring one way in which well-formed structures violating the strict adjacency constraint may be parsed. This involves increasing the search space during parsing so that the top stack element is not the only one to be examined. However, search in his system is not unconstrained, as in Covington's system. Instead, Hellwig's parser searches the top stack element in a first parsing cycle, and then searches the next-to-top element in the next parsing cycle. Thus, the claim implicit in the design of the parser is that an element which is not immediately accessible to its head will not be separated from its head by more than one subtree. In this way, head-dependent pairs which are not adjacent in the standard sense can be found, so long as they conform to the 'next-but-one constraint'.

However, a cautious note must be sounded here. If real progress is to be achieved in this area, modifications and extensions to the basic Gaifman format of DG rules must be formally defined. Without explicit definition of the systems assumed, all results will be uncertain at best and useless at worst. Regrettably, strict formal definition has been the exception rather than the rule in DG studies. It is to be hoped that as interest in dependency parsing increases, the discipline imposed by the requirements of computers for formal rigour will help to overcome this shortcoming.

12.8 Summary

In this chapter, drawing on the survey of dependency parsers in the preceding chapters, I have tried to identify some of the dimensions of variation in dependency parsing and to draw out some principles and techniques. Variation was

found in search origin, search manner, search order, number of passes, search focus, ambiguity management, and in the use of an adjacency constraint on search. Substantial similarities with standard PSG parsing were found. The main differences concern search origin, search focus, and the use of an adjacency constraint.

DG trees can be seen as a special case of PSG trees in which every node directly dominates exactly one terminal symbol. One consequence of this is that traditional terms relating to the origin of search in constituency parsing, such as 'top-down' and 'bottom-up', can not be borrowed into dependency parsing without some specialization of meaning. I have tried to define these terms for the purposes of dependency parsing, and have added some new distinctions, such as the distinction between 'deep' top-down parsing and 'shallow' top-down parsing.

Search, in dependency parsing, can focus on a variety of different things. For a given word, the object of search may be to find a head for the word, or a dependent for the word, or both. In my discussion I identify eight different search foci, although others may be possible. The issue of *what* to search for seems to be particular to dependency parsing. I have shown how the choice of search focus can determine a number of design features, and may even determine whether or not the parser is able to parse successfully.

An adjacency constraint can reduce a large search space so that it could hardly be smaller. An adjacency constraint can also prevent a parser from discovering valid analyses. I have shown how different parsers embody different attempts to balance the requirements of constrained search within the context of natural language phenomena. I have also advocated DG as a particularly useful framework for exploring this problem.

Most importantly, I have identified work which still needs to be done. The management of ambiguity warrants special mention here, since very few dependency parsing systems take this problem seriously. The special requirements

of at least some extended versions of DG mean that, for them, existing tools for the management of ambiguity in constituency parsers are likely to be inappropriate.

Chapter 13

Conclusion

"Use your head!"
Traditional.

At the beginning of this thesis I set out the formal properties of DGs, as defined by Gaifman. I reported that his version of DG is equivalent to a subclass of the CFPSGs, namely the class in which every phrase contains exactly one category which is a projection of a lexical category. It is exactly this subclass of CFPSGs which most linguists assume in analyses of natural language. The differences between the grammatical systems, then, are not significant either in terms of their formal power or their adequacy for describing natural language. However, it must be added that many — perhaps the majority — of theoretical linguists who use DG have added extensions to the basic formalism, thereby creating new kinds of system of uncertain formal power. In this thesis I have focussed on those dependency systems which have a discernible core which may be expressed in terms of a Gaifman grammar.

The field of PSG parsing evolved — in computer science and in computational linguistics — with the assumption that PSG rules do not distinguish one item in a phrase (the head) as having privileged status. It is only comparatively recently (within the last decade or so) that most phrase structure grammarians have come to assume that every phrase does, indeed, have a head. Thus, head-driven parsing using PSGs has emerged as a live research

topic even more recently. The principal difference between DG and PSG is that DG rules necessarily identify the head of each construction, whereas PSG rules only identify the head of a phrase if some additional constraint is supplied (as in the case of versions of $\bar{X}$ grammar). Head-marking is intrinsic to DG, but extrinsic to PSG as originally defined. One would therefore expect to find a much longer record of work on head-driven parsing in the field of dependency parsing.

Unfortunately, what emerges from this survey of existing dependency parsing systems does not satisfy these expectations. There has been very little emphasis in the dependency parsing literature on exploring what is distinctive about parsing with head-marked rules. Some parsers, (for example, the DLT ATN and DCG systems) make no special use of heads at all. On the other hand, there have been hardly any visible attempts to relate developments in dependency parsing to well known and understood results in phrase structure parsing. Only Hellwig's WFST parser stands as a deliberate attempt to borrow an existing PSG parsing technique while attempting to make use of the headedness of DG rules.

The empirical evidence furnished by this survey is that almost all dependency parsers constructed so far operate bottom-up incrementally. The basic operation of these parsers is to construct pairwise dependency relations. The discovery of larger constructions (phrases) follows as a consequence of this, not as the result of special phrase-building operations. However, there is nothing in all this which could not have a PSG parsing correlate.

By categorizing as many of the parsers surveyed as possible using fairly well-understood parsing terms (e.g. top-down, depth-first search), I have begun to explore the space within which dependency parsing algorithms are located. The most important conclusion to draw here is that the space is — on almost every count — the same as that occupied by PSG parsing algorithms. It has not been necessary to introduce completely new terms to describe what is going on

in dependency parsing algorithms; existing terms will suffice. However, some minor divergences have come to light as, for example, in the case of top-down parsing which I have subcategorized into deep and shallow variants. Whereas deep top-down parsing can be implemented either in a dependency framework or any PSG framework, shallow top-down parsing appears to be particular to head-marking frameworks.

Thus, what emerges from the survey is the beginnings of a taxonomy of dependency parsing algorithms, in which it is clear that some configurations of properties have been much more thoroughly explored than others. In this way, I have identified certain clusters of properties which, though commonly reported in the the PSG parsing literature, are not represented in this survey. I have attempted to make good some of these deficits by describing what a dependency solution would look like and, especially, by supplying Prolog instantiations of these solutions in the Appendix.

And so we turn to the hypotheses introduced at the start of the thesis to help point up the similarities and differences between dependency parsing and standard PSG parsing.

> **Hypothesis 1**
>
> It is possible to construct a fully functional dependency parser based directly on an established phrase structure parsing algorithm without altering any fundamental aspects of the algorithm.

I have offered at least two existence proofs of this hypothesis in the text. In the first case, I showed how a shift-reduce parsing algorithm as standardly applied in PSG parsing could be taken over into dependency parsing. The PSG and DG versions of the algorithm differ only trivially in the way in which they represent knowledge. Otherwise, they are identical. If PSG and DG parses are followed through for the same sentence with equivalent grammars, the operation of the parsers is identical, shift for shift, reduction for reduction.

As an even clearer proof of the truth of Hypothesis 1, I borrowed Gazdar

and Mellish's existing implementation of a PSG bottom-up chart parser and showed how, with only the most modest of changes to the code, and none at all to the basic algorithm, it could work given an arbitrary dependency grammar.

This should not be surprising, since this is a very weak hypothesis. It is well understood that dependency rules include phrasal information so what is to stop them working in combination with phrase-building algorithms? However, it is not the case that arbitrary PSG rules incorporate dependency information. This is the motivation for the stronger hypothesis, Hypothesis 2.

Hypothesis 2

It is possible to construct a fully functional dependency parser using an algorithm which could not be used without substantial modification in a fully functional conventional phrase structure parser.

An existence proof for this hypothesis is provided by the divide-conquer algorithm. This works on the principle that top-down parsing need never hypothesize an expansion without immediately checking it in the string. It works solely because every rule in the dependency grammar explicitly mentions a lexical head, which can always be identified in the rule. This is not the case in an arbitrary PSG. This algorithm is particularly attractive by virtue of the possibilities it raises for dividing up the parse problem and solving (conquering) the different parts in parallel.

However, though Hypothesis 2 has been proven literally, it misses an important point. It is difficult to study the subject of dependency parsing without being drawn to this conclusion: it is invidious to contrast PSG parsers with dependency parsers; the more profitable comparison is that between parsers which make use of the notion 'head' and those which do not. While most of the standard PSG parsing algorithms are not head-driven, a small number (which use head-marked versions of PSG) are. Conversely, although a dependency rule without head-marking is inconceivable, this survey has shown that by no means all dependency parsers make significant use of information about heads.

The overwhelming weight of opinion in linguistic theory supports the marking of heads in phrases, but remarkably little progress has yet been won by the introduction of explicitly marked heads in parsing systems. Parsing in the dependency grammar tradition, which ought to be a rich information source, turns out to be generally disappointing, not least because the systems which have been developed have never been systematically related to any other (more mainstream) parsing results. I offer this thesis as a first step towards the integration of dependency parsing with mainstream work on head-driven parsing.

Appendix

Prolog Listings

A.1 Introduction

The programs listed in this appendix are written in Quintus Prolog (version 3.0.1). A restricted sub-set of Quintus built-in predicates has been used to encode the algorithms described in the main text. This sub-set is entirely consistent with standard 'Edinburgh' syntax *(Clocksin and Mellish 1987)*. However, a small number of non-standard predicates has been utilised to set up the environment in which the main algorithms are located. The most common of these is `ensure_loaded/1` which is broadly equivalent to 'Edinburgh' `reconsult/1`. It is used to load the predicates defined in another file. The argument of `ensure_loaded/1` may be either a filename (minus Quintus Prolog's compulsory '.pl' extension) or a term of the form `library(X)`, where X is the name of a Quintus library file. The only such file to be loaded is `files` which provides a collection of predicates for manipulating text files. The particular library predicate used in the programs listed here is `file_exists/1` which takes as its argument the name of a file. The predicate succeeds if the file exists (i.e. can be found in the current directory by the Prolog system). Most practical Prologs provide a broadly equivalent predicate, although predicate names differ from system to system.

Quintus Prolog requires that all dynamic predicates (i.e. predicates which may be asserted or retracted at runtime) be explicitly declared. This is usually done at the beginning of the file containing the relevant `assert/1` or `retract/1` predicate calls. Dynamic predicate declarations have the following form:

```
:- dynamic Predicate/N.
```

`Predicate` is the name of the dynamic predicate; `N` is its arity. Both `Predicate` and `N` must be instantiated. Each dynamic predicate declaration may simply be commented out for use with Prologs which do not require such declarations.

The listings set out below present a diverse range of recognition and parsing algorithms which are united in their use of dependency grammars, but divided in the ways in which they manipulate their data structures, including their internal representation of grammars. For this reason a compilation methodology has been used for those algorithms which make use of Gaifman-style dependency grammars (see Chapter 2 for details). The grammar writer writes a Gaifman dependency grammar using Gaifman's standard notation. This is subsequently compiled into the Prolog-internal representation most appropriate (i.e. efficient) for each algorithm. The compilation process only restructures grammar rules — it does not add or subtract information. The code for the Gaifman dependency grammar rule compiler is listed in the file `dg_compile.pl`.

Section A.2 indexes each predicate which appears in the listing according to the file in which it is defined. The files themselves are given in alphabetical order according to file name in Section A.3. A sample grammar to illustrate some basic features of the parsers appears in Section A.4.

A.2 Index of predicates

Predicate	File
add_spans_including_trees/3	hays_parser.pl
allowed_char/1	dg_compile.pl
alpha_numeric/1	dg_compile.pl
append/3	lib.pl
assert_if_new/1	lib.pl
begin_new_line/0	map_to_dcg.pl
build_cat_list/2	hays_generator.pl
concat/3	lib.pl
conquer/5	divide.pl
construct_assignments/0	map_to_dcg.pl
construct_assignments/2	map_to_dcg.pl
construct_call/0	map_to_dcg.pl
construct_embedded_call/0	map_to_dcg.pl
construct_rules/0	map_to_dcg.pl
cross_product/3	lib.pl
dep_write/3	map_to_dcg.pl
dcg_generate/0	dcg.pl
dcg_parse/0	dcg.pl
dg_compile/1	dg_compile.pl
dg_compile_loop/2	dg_compile.pl
divide/4	divide.pl
divide_conquer/0	divide.pl
divide_conquer/1	divide.pl
dot	lib.pl
drule/3	dg_compile.pl
each_member/2	lib.pl
each_tree/4	hays_generator.pl
embedded_stage_two/3	hays_generator.pl
embedded_x_product/3	lib.pl
enumerate/0	hays_generator.pl
enumerate/1	hays_generator.pl
enumerate_loop/0	hays_generator.pl
enumerate_surface/1	hays_generator.pl
extract_any_sub_string_with_trees/4	hays_parser.pl
extract_sub_string_and_trees/5	hays_parser.pl
ff_drule/3	dg_compile.pl
flush_comment/3	dg_compile.pl

flushline/2	dg_compile.pl
generate_one_root/1	dcg.pl
generate_tree/1	hays_generator.pl
get_all_chars/1	dg_compile.pl
get_all_chars2/3	dg_compile.pl
grammar_present/2	dg_compile.pl
group/4	dg_compile.pl
in_word/2	lib.pl
incorporate/2	dg_compile.pl
init/1	divide.pl
initialize_parse_table/2	hays_parser.pl
known_tree/1	hays_generator.pl
lower_case/1	dg_compile.pl
map_to_dcg/2	map_to_dcg.pl
multi_line/1	dg_compile.pl
note_grammar_present/2	dg_compile.pl
numeric/1	dg_compile.pl
padding_char/1	dg_compile.pl
parse_increasing_substrings/1	hays_parser.pl
print_set/1	dcg.pl
purge_grammar_rules/0	lib.pl
read_in/1	lib.pl
readword/3	lib.pl
restsent/3	lib.pl
return_admissible_trees/2	hays_parser.pl
reverse/2	lib.pl
reverse/3	lib.pl
rff_drule/3	dg_compile.pl
root/1	dg_compile.pl
saturate/2	dg_compile.pl
sentence_length/1	hays_parser.pl
separator/1	dg_compile.pl
show_complete_tree/0	hays_parser.pl
spans/3	hays_parser.pl
special_char/1	dg_compile.pl
sr_recognize/0	shift_reduce.pl
sr_recognize/1	shift_reduce.pl
sr_recognize_loop/2	shift_reduce.pl
sr_reduce/2	shift_reduce.pl
stage_one/1	hays_generator.pl
stage_two/2	hays_generator.pl
surface/2	hays_generator.pl

tabular_parse/0	hays_parser.pl
tokenize/1	dg_compile.pl
upper_case/1	dg_compile.pl
whittle/5	divide.pl
word_class/2	dg_compile.pl
word_classify/2	divide.pl
word_exs/3	map_to_dcg.pl
write_sentence_list/1	lib.pl
writeln/1	lib.pl

A.3 Code listings

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           dcg.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        A definite clause grammar incorporating some
%                       notions from dependency grammar. For more
%                       information on definite clause grammars see
%                       Pereira and Warren (1980).
%
%   VERSION HISTORY:    1.0 November 28, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
:- ensure_loaded(lib).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*************************************************************************
*
*  dcg_parse/0.
*
*  Parse a string using a definite clause grammar. Return a dependency
*  tree if the parse succeeds.
*  For example, typing:
*
*  | ?- dcg_parse.
*  |: the big mouse chased the timid cat.
*
*  produces the result:
*
*  Parse tree: verb(noun(det(*),adj(*),*),*,noun(det(*),adj(*),*))
*
*/
dcg_parse :-
        read_in(String),
        !,
        root(Root),
        Rule =.. [Root,Tree],
        phrase(Rule,String,['.']),
        writeln(['Parse tree: ',Tree]),
        nl.
dcg_parse :-
        writeln('PARSE FAILED'),
        nl.


/*************************************************************************
*
*  dcg_generate/0.
```

```
*
*  Generate all strings (and associated syntactic parse trees) defined
*  by the DCG.
*/
dcg_generate :-
        setof(Root,root(Root),Set),
        generate_one_root(Set),
        nl.


/*************************************************************************
*
*  generate_one_root(+RootList).
*
*  Generate all possible strings for a given sentence root.
*/
generate_one_root([]).
generate_one_root([First|Rest]) :-
        Rule =.. [First,Tree],
        setof([String,Tree],phrase(Rule,String),Set),
        print_set(Set),
        generate_one_root(Rest).


/*************************************************************************
*
*  print_set(+ResultList).
*
*  Print out a list of String/Tree generation result pairs, one
*  pair at a time.
*/
print_set([]) :-
        nl.
print_set([[String,Tree]|Rest]) :-
        writeln(['String: ',String]),
        writeln(['Tree:   ',Tree]),
        nl,
        print_set(Rest).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       THE GRAMMAR
%       A very simple definite clause grammar to illustrate how to
%       build dependency trees using DCGs.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

adj(X) --> [Head],
        { class(Head,adj),
          X = adj(*) }.

det(X) --> [Head],
        { class(Head,det),
          X = det(*) }.
```

```
noun(X) --> det(Det), [Head],
        { class(Head,noun),
          X = noun(Det,*) }.
noun(X) --> det(Det), adj(Adj), [Head],
        { class(Head,noun),
          X = noun(Det,Adj,*) }.

i_verb(X) --> noun(Noun), [Head],
        { class(Head, i_verb),
          X = i_verb(Noun,*) }.

t_verb(X) --> noun(Noun1), [Head], noun(Noun2),
        { class(Head,t_verb),
          X = t_verb(Noun1,*,Noun2) }.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       VALID SENTENCE ROOTS
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

root(i_verb).
root(t_verb).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       WORD CLASS ASSIGNMENTS
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

class(big,adj).
class(fierce,adj).
class(timid,adj).

class(a,det).
class(the,det).

class(cat,noun).
class(dog,noun).
class(mouse,noun).

class(snored,i_verb).
class(ran,i_verb).

class(chased,t_verb).
class(likes,t_verb).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           dg_compile.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        Compile a standard Gaifman format dependency
%                       grammar into several different forms, namely:
%                       Gaifman Prolog form, full form, and reversed
%                       full form.
%
%   VERSION HISTORY:    1.0 August 12, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
%  library(files) is a Quintus Prolog library. To run with other
%  prologs replace call to file_exists/1 in dg_compile/2 with the
%  local equivalent.
%
:- ensure_loaded(library(files)).
:- ensure_loaded(lib).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  DYNAMIC PREDICATE DECLARATIONS
:- dynamic multi_line/1.
:- dynamic root/1.
:- dynamic word_class/2.
:- dynamic drule/3.
:- dynamic ff_drule/3.
:- dynamic rff_drule/3.
:- dynamic grammar_present/2.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/*************************************************************************
*
*  dg_compile(+File).
*  dg_compile(+Compilation,+File).
*
*   Compile a Gaifman dependency grammar into a variety of
*   Prolog-readable forms. Three compilations are supplied.
*
*   Gaifman dependency grammars allow rules of the following three
*   varieties:
*
*   (i)     *(X)
*   (ii)    X(*)
*   (iii)   X(Y1,Y2,...,Yi,*,Yj...,Yn-1,Yn)
*
*   GAIFMAN PROLOG FORM
*   Gaifman Prolog Form (GPF) is the simplest Prolog implementation of
```

```
*   Gaifman's rule system, therefore it may be regarded as the
*   canonical implementation. A grammar in standard Gaifman form can be
*   compiled into GPF as follows:
*
*   (1) Replace every rule of type 1 with a GPF rule of type 'root(X).'
*   (2) Replace every rule of type 2 with a GPF rule of type
*       'drule(X,[],[]).'
*   (3) Replace every rule of type 3 with a GPF rule of type
*       'drule(X,A,B).' where A is a Prolog list consisting of Y1-Yi in
*       the same order as they appear in the original rule, and B is a
*       Prolog list consisting of Yj-Yn in the same order as they appear
*       in the original rule. If nothing precedes '*' in the original rule,
*       then A = []; if nothing follows '*' in the original rule then B = [].
*
*   To compile a Gaifman grammar contained in a file called 'grammar1' into
*   GPF, use:
*
*       dg_compile(gpf,grammar1).
*
*   Since GPF is the default compilation, the same result may be achieved
*   using:
*
*       dg_compile(grammar1).
*
*   FULL FORM
*   Full form dependency rules are produced using the following mapping:
*
*   (1) Replace every rule of type 1 with a full form rule of type 'root(X).'
*   (2) Replace every rule of type 2 with a full form rule of type
*       'ff_drule(X,[X]).'
*   (3) Replace every rule of type 3 with a full form rule of type
*       'ff_drule(X,A).' where A is the Prolog list consisting of the
*       concatenation of Y1-Yi, X, and Yj-Yn in that order.
*
*
*   To compile a Gaifman grammar contained in a file called 'grammar1' into
*   full form, use:
*
*       dg_compile(ff,grammar1).
*
*   REVERSED FULL FORM
*   Rerersed ull form dependency rules are produced using the following
*   mapping:
*
*   (1) Replace every rule of type 1 with a full form rule of type 'root(X).'
*   (2) Replace every rule of type 2 with a full form rule of type
*       'rff_drule(X,[X]).'
*   (3) Replace every rule of type 3 with a full form rule of type
*       'ff_drule(X,A).' If A is a Prolog list consisting of the
*       concatenation of Y1-Yi, X, and Yj-Yn in that order, then A1
*       is the mirror image of that list.
*
*   To compile a Gaifman grammar contained in a file called 'grammar1' into
*   reversed full form, use:
*
```

```
*       dg_compile(rff,grammar1).
*
*   To compile the same source file into all three formats at the same
*   time use:
*
*       dg_compile(all,grammar1).
*
*   The output of dg_compile/1 and dg_compile/2 is written directly to
*   the Prolog internal database (user).
*/
dg_compile(File) :-
        dg_compile(gpf,File).

dg_compile(Compilation,File) :-
        (
        file_exists(File)
        |
        writeln(['Unknown file: ',File]),
        abort
        ),
        writeln(['Compiling ',File,' into ',Compilation,' format.']),
        see(File),
        retractall(multi_line(_)),
        assert(multi_line(off)),
        tokenize(FirstRule),
        dg_compile_loop(Compilation,FirstRule),
        told,
        note_grammar_present(Compilation,File),
        close_all_streams,
        writeln('Grammar compilation completed.').

dg_compile_loop(Compilation,eof([])).
dg_compile_loop(Compilation,eof(Rule)) :-
        dot,
        phrase(valid_rule(X),Rule),
        incorporate(Compilation,X).
dg_compile_loop(Compilation,[]) :-
        tokenize(Rule),
        dg_compile_loop(Compilation,Rule).
dg_compile_loop(Compilation,FirstRule) :-
        dot,
        phrase(valid_rule(X),FirstRule),
        incorporate(Compilation,X),
        tokenize(NextRule),
        dg_compile_loop(Compilation,NextRule).

incorporate(all,dependency_rule(Head,Before,After)) :-
        assertz(drule(Head,Before,After)),
        append(Before,[Head|After],Phrase),
        assertz(ff_drule(Head,Phrase)),
        reverse(Phrase,RevPhrase),
        assertz(ff_drule(Head,RevPhrase)).

incorporate(gpf,dependency_rule(Head,Before,After)) :-
        assertz(drule(Head,Before,After)).
```

```
incorporate(gpf_sat,dependency_rule(Head,Before,After)) :-
        saturate(Before,Before1),
        saturate(After,After1),
        assertz(gpf_sat_drule(Head,Before1,After1)).
incorporate(ff,dependency_rule(Head,Before,After)) :-
        append(Before,[Head|After],Phrase),
        assertz(ff_drule(Head,Phrase)).
incorporate(ff_sat,dependency_rule(Head,Before,After)) :-
        saturate(Before,Before1),
        saturate(After,After1),
        Head1 =.. [Head,*],
        append(Before1,[Head|After1],Phrase),
        assertz(ff_sat_drule(Head1,Phrase)).
incorporate(rff,dependency_rule(Head,Before,After)) :-
        append(Before,[Head|After],Phrase),
        reverse(Phrase,RevPhrase),
        assertz(ff_drule(Head,RevPhrase)).
incorporate(rff_sat,dependency_rule(Head,Before,After)) :-
        saturate(Before,Before1),
        saturate(After,After1),
        Head1 =.. [Head,*],
        append(Before1,[Head|After1],Phrase),
        reverse(Phrase,RevPhrase),
        assertz(rff_sat_drule(Head1,RevPhrase)).

incorporate(_,sentence_root(Root)) :-
        assertz(root(Root)).
incorporate(_,class_assign(_,[])).
incorporate(_,class_assign(Class,[FirstWord|Rest])) :-
        assertz(word_class(FirstWord,Class)),
        incorporate(_,class_assign(Class,Rest)).

note_grammar_present(all,Grammar) :-
        note_grammar_present(gpf,Grammar),
        note_grammar_present(ff,Grammar),
        note_grammar_present(rff,Grammar).
note_grammar_present(Format,Grammar) :-
        assert(grammar_present(Format,Grammar)).

saturate([],[]).
saturate([First|Rest],[New|Result]) :-
        New =.. [First,*],
        saturate(Rest,Result).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       TOKENIZE A DEPENDENCY GRAMMAR
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*************************************************************************
*
*  tokenize(-ListOfTokens)
*
```

```
*  Produce a list of tokens for the current line in the standard input.
*/
tokenize(Result) :-
        get_all_chars(ListOfChars),
        group(ListOfChars,[],[],Result).


/*************************************************************************
*
*  get_all_chars(+Filename,-ListOfChars)
*
*  Construct a list of all legitimate characters on the current line
*  (in reverse order).
*/
get_all_chars(AllChars) :-
        get0(C),
        get_all_chars2(C,[],AllChars).

get_all_chars2(C,Result,eof(Result)) :-
        end_of_file(C).
get_all_chars2(C,Result,Result) :-
        multi_line(off),
        newline(C).
get_all_chars2(C,Current,Result) :-
        comment(C),
        flushline(C,C1),
        get_all_chars2(C1,Current,Result).
get_all_chars2(ThisChar,[LastChar|Current],Result) :-
        asterisk(ThisChar),
        oblique(LastChar),
        flush_comment(120,120,C1),
        get_all_chars2(C1,Current,Result).
get_all_chars2(C,Current,Result) :-
        close_curly(C),
        retractall(multi_line(_)),
        asserta(multi_line(off)),
        get0(C1),
        get_all_chars2(C1,[C|Current],Result).
get_all_chars2(C,Current,Result) :-
        open_curly(C),
        retractall(multi_line(_)),
        asserta(multi_line(on)),
        get0(C1),
        get_all_chars2(C1,[C|Current],Result).
get_all_chars2(C,Current,Result) :-
        allowed_char(C),
        get0(C1),
        get_all_chars2(C1,[C|Current],Result).
get_all_chars2(C,Current,Result) :-
        write('! Illegal character ignored: '),
        put(C),
        write(' (ASCII '),
        write(C),
        write(')'),
        nl,
```

```
        get0(C1),
        get_all_chars2(C1,Current,Result).


/*************************************************************************
*
*  group(+Inlist,?Current_Word,+Current_List,-Result)
*
*  Tokenize a list of character codes.
*/
group(eof(Anything),One,Two,eof(Result)) :-
        group(Anything,One,Two,Result).
group([],[],Result,Result).
group([],Current_List,So_Far,[Current_Atom|So_Far]) :-
        name(Current_Atom,Current_List).
group([H|T],[],So_Far,Result) :-
        special_char(H),
        name(Current_Atom,[H]),
        group(T,[],[Current_Atom|So_Far],Result).
group([H|T],[],So_Far,Result) :-
        padding_char(H),
        group(T,[],So_Far,Result).
group([H|T],Current_List,So_Far,Result) :-
        alpha_numeric(H),
        group(T,[H|Current_List],So_Far,Result).
group([H|T],Current_List,So_Far,Result) :-
        separator(H),
        name(Current_Atom,Current_List),
        group([H|T],[],[Current_Atom|So_Far],Result).


/*************************************************************************
*
*   Character manipulation utilities and definitions
*
*************************************************************************/

/*************************************************************************
*
*  flushline/0.
*
*  Flush the input buffer to the next end of line.
*/
flushline(C,C) :-
        end_of_file(C).
flushline(C,C1) :-
        newline(C),
        get0(C1).
flushline(_,C) :-
        get0(C1),
        flushline(C1,C).


/*************************************************************************
*
```

```
*  flush_comment(+CurrentChar,+PreviousChar,+ReturnChar).
*
*  Flush the input buffer to the end of the next multiline comment.
*/
flush_comment(C,_,C) :-
        end_of_file(C).
flush_comment(C,C1,C2) :-
        oblique(C),
        asterisk(C1),
        get0(C2).
flush_comment(C1,_,C3) :-
        get0(C2),
        flush_comment(C2,C1,C3).


allowed_char(C) :-
        padding_char(C).
allowed_char(C) :-
        alpha_numeric(C).
allowed_char(C) :-
        special_char(C).

separator(C) :-
        padding_char(C).
separator(C) :-
        special_char(C).

padding_char(C) :-
        space(C).
padding_char(C) :-
        tab_char(C).
padding_char(C) :-
        comma(C).
padding_char(C) :-
        period(C).
padding_char(C) :-
        newline(C).

alpha_numeric(C) :-
        lower_case(C).
alpha_numeric(C) :-
        upper_case(C).
alpha_numeric(C) :-
        underscore(C).
alpha_numeric(C) :-
        numeric(C).

lower_case(C) :-
        C >= 97,
        C =< 122.

upper_case(C) :-
        C >= 65,
        C =< 90.
```

```
numeric(C) :-
        C >= 48,
        C =< 57.

special_char(C) :-
        open_bracket(C).
special_char(C) :-
        close_bracket(C).
special_char(C) :-
        colon(C).
special_char(C) :-
        asterisk(C).
special_char(C) :-
        open_curly(C).
special_char(C) :-
        close_curly(C).
special_char(C) :-
        oblique(C).

end_of_file(-1).    % EOF
tab_char(9).        % tab
newline(10).        % nl
space(32).          % ' '
comment(37).        % %
open_bracket(40).   % (
close_bracket(41).  % )
asterisk(42).       % *
comma(44).          % ,
dash(45).           % -
period(46).         % .
oblique(47).        % /
colon(58).          % :
underscore(95).     % _
open_curly(123).    % {
close_curly(125).   % }



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       A DEFINITE CLAUSE GRAMMAR FOR DEPENDENCY GRAMMAR RULES
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
%  VALID RULE TYPES
%
valid_rule(X) -->
        dependency_rule(X).
valid_rule(X) -->
        class_assignment(X).
valid_rule(X) -->
        root_declaration(X).

%
```

```
%  WORD CLASS ASSIGNMENT RULES
%
class_assignment(X) -->
        [A], colon_string(B), set_of_words(C),
        {atom(A),
        X = class_assign(A,C)}.

colon_string(X) -->
        [':'].

set_of_words(X) -->
        open_set(A), word_list(X), close_set(C).

open_set(X) -->
        ['{'].

close_set(X) -->
        ['}'].

word_list(X) -->
        [A],
        {atom(A),
        X = [A]}.
word_list(X) -->
        [A], word_list(B),
        {atom(A),
        X = [A|B]}.

%
%  SENTENCE ROOT RULES
%
root_declaration(X) -->
        asterisk_string(A), open_brkt_string(B), [C], close_brkt_string(D),
        {atom(C),
        X = sentence_root(C)}.

asterisk_string(X) -->
        ['*'].

open_brkt_string(X) -->
        ['('].

close_brkt_string(X) -->
        [')'].

%
%  DEPENDENCY RULES
%
dependency_rule(X) -->
        [A], open_brkt_string(B), asterisk_string(C), close_brkt_string(D),
        {atom(A),
        X = dependency_rule(A,[],[])}.
dependency_rule(X) -->
        [A], open_brkt_string(B), word_list(C), asterisk_string(D),
        close_brkt_string(E),
```

```
        {atom(A),
        X = dependency_rule(A,C,[])}.
dependency_rule(X) -->
        [A], open_brkt_string(B), asterisk_string(C), word_list(D),
        close_brkt_string(E),
        {atom(A),
        X = dependency_rule(A,[],D)}.
dependency_rule(X) -->
        [A], open_brkt_string(B), word_list(C), asterisk_string(D),
        word_list(E), close_brkt_string(F),
        {atom(A),
        X = dependency_rule(A,C,E)}.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           divide_conquer.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        Divide & Conquer. A shallow top-down dependency
%                       parser.
%
%   VERSION HISTORY:    1.0 December 17, 1990
%                       1.1 August 8, 1992 (NMF)
%                       1.2 January 16, 1992 (NMF)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD_DECLARATIONS
:- ensure_loaded(library(files)).
:- ensure_loaded(lib).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


 /*************************************************************************
*
*  divide_conquer/1.
*  divide_conquer/0.
*
*  Parse a string. Version with filename argument loads a Gaifman Prolog
*  Form grammar. The parser is based on the 'divide and conquer' algorithm.
*  The basic idea is to use the head of a rule to split the string to be
*  parsed in two and then to recurse down each half in turn.
*/
divide_conquer(File) :-
        (
        file_exists(File),
        purge_grammar_rules,
        dg_compile(File)
        |
        writeln(['ERROR! Non-existent grammar file: ',File,'.']),
        abort
        ),
        divide_conquer.

divide_conquer :-
        write('Type the sentence to be parsed (end with a full stop)'),
        nl,
        write(': '),
        read_in(Sentence),
        word_classify(Sentence, Class_List),
        init(Class_List).
divide_conquer :-
        writeln(['*** PARSER FAILED ***']).
```

```
/*************************************************************************
*
*  word_classify(+Classless,-Classified).
*
*  Take a list of unclassified words and return a list of word classes,
*  basing assignments on the current grammar.
*/
word_classify([.],[]).
word_classify([Word|Rest_Words],[Class|Class_List]) :-
           word_classify(Rest_Words, Class_List),
           word_class(Word,Class).


/*************************************************************************
*
*  init(+String).
*
*  Begin the parse.
*/
init(List) :-
        root(Start),
        drule(Start, Left_Deps, Right_Deps),
        divide(List,Left,Right,Start)),
        conquer(Start,Left,Left_Deps,,[],Report1),
        conquer(Start,Right,Right_Depps,[],Report2),
        writeln(['Root = ',Start]),
        writeln(['Leftside = ',Reportt1]),
        writeln(['Rightside = ',Reporrt2]).


/*************************************************************************
*
*  divide(+String,-LeftPart,-RightParrt,+Head).
*
*  Find Head in String and return thee substring to its left as LeftPart
*  and the substring to its right as  RightPart.
*/
divide([],_,_,_).
divide([H|T],[],T,H).
divide([H|T],[H|Left],Right,Root) :-
        divide(T,Left,Right,Root).


/*************************************************************************
*
*  conquer(+Head,+String,+Dependents;,-Remainder_of_Substring,-Report).
*
*  Find trees rooted in each of the IDependents in String. These will
*  each depend on Head. Return any oif String not accounted for as
*  Remainder. Report what has been found.
*/
conquer(_,[],[],[],[]).                    % SUCCEED: all satisfied
conquer(_,[],[_|_],_,_) :-                 % FAIL: deps but no words
        !,
        fail.
```

```
conquer(Head,[Dep],[Dep],[],[(Dep,HHead)]) :-
        drule(Dep,[],[]).                % SUCCEED: only dep matches only word
conquer(Head,String,[Dep],Remainderr, [(Dep,Head)|Report3]) :-
        drule(Dep,Left_Deps,Right_DDeps),% ONE DEP: divide and conquer
        divide(String,Left,Right,Deep),
        conquer(Dep, Left, Left_Depps,[],Report1),
        conquer(Dep, Right, Right_DDeps,Remainder,Report2),
        append(Report1, Report2, Reeport3).
conquer(Head,String,[First_Dep|Restt_Deps],Remainder,Report5) :-    % MANY DEPS
        drule(First_Dep, Left_Deps,, Right_Deps),
        divide(String,Left,Right,Fiirst_Dep),
        conquer(First_Dep,Left,Leftt_Deps,[],Report1),
        whittle(First_Dep, Right, RRight_Deps, Remainder2,Report2),
        conquer(Head, Remainder2, RRest_Deps,Remainder1,Report4),
        append(Report1,Report2,Repoort3),
        append(Report3,Report4,Repoort5).
conquer(Head,[First_Word|Rest_Words],[First_Dep|Rest_Deps],Remainder,
                                        [(First_Dep,Head)|Report1]) :-
        drule(First_Dep,[],[]),
        conquer(Head,Rest_Words,Restt_Deps,Remainder,Report1).


 /*************************************************************************
*
*  whittle(+Head,+String,+Dependentss,-Remainder,-Report).
*
*  A reduced version of conquer/5 for whittling down String when
*  more than one tree must be found in it.
*/
whittle(_, Remainder, [], Remainder,,_).
whittle(Head,[Dep1|Rest_Words],[Dep11|Rest_Deps],Remainder,[(Dep1,Head)|Report1]) :-
        drule(Dep1,[],[]),
        whittle(Head,Rest_Words,Restt_Deps,Remainder,Report1).
whittle(Head,String,[Dep1|Rest_Deps]],Remainder,Report3) :-
        drule(Dep1,Left_Deps,Right_DDeps),
        divide(String,Left,Right,Depp1),
        conquer(Dep1,Left,Left_Deps,,[],Report1),
        conquer(Dep1,Right,Right_Depps,Remainder,Report2),
        append(Report1,Report2,Reporrt3).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           gazdar_mellish.pl
%
%   WRITTEN BY:         Gerald Gazdar & Chris Mellish, with minor
%                       modifications by Norman M. Fraser.
%
%   DESCRIPTION:        Contains the concatenation of several files
%                       (namely: buchart1.pl, chrtlib1.pl, library.pl,
%                       psgrules.pl, lexicon.pl, examples.pl) from the
%                       program listings in Gazdar & Mellish (1989).
%                       Some minor changes have been made to make the
%                       program run under Quintus Prolog. A few
%                       predicates which are irrelevant here have
%                       been removed (mostly from library.pl).
%
%   VERSION HISTORY:    January 16, 1993 (date created in this form)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  ORIGINAL NOTICE FOLLOWS:
%
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%   Example code from the book "Natural Language Processing in Prolog"  %
%                       published by Addison Wesley                     %
%         Copyright (c) 1989, Gerald Gazdar & Christopher Mellish.      %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%
%  Reproduced by kind permission.
%


/*************************************************************************/
%
% buchart1.pl A bottom-up chart parser
%
/*************************************************************************/

parse(V0,Vn,String) :-
        start_chart(V0,Vn,String).      % defined in chrtlib1.pl
%
add_edge(V0,V1,Category,Categories,Parse) :-
        edge(V0,V1,Category,Categories,Parse),!.
%
add_edge(V1,V2,Category1,[],Parse) :-
        assert_edge(V1,V2,Category1,[],Parse),
        foreach(rule(Category2,[Category1|Categories]),
                add_edge(V1,V1,Category2,[Category1|Categories],[Category2])),
        foreach(edge(V0,V1,Category2,[Category1|Categories],Parses),
                add_edge(V0,V2,Category2,Categories,[Parse|Parses])).
add_edge(V0,V1,Category1,[Category2|Categories],Parses) :-
        assert_edge(V0,V1,Category1,[Category2|Categories],Parses),
        foreach(edge(V1,V2,Category2,[],Parse),
                add_edge(V0,V2,Category1,Categories,[Parse|Parses])).
```

```
/*************************************************************************/
%
% chrtlib1.pl Library predicates for database chart parsers
%
/*************************************************************************/

%
% start_chart
% uses add_edge (defined by particular chart parser) to insert inactive
% edges for the words (and their respective categories) into the chart
%
start_chart(V0,V0,[]).
start_chart(V0,Vn,[Word|Words]) :-
        V1 is V0+1,
        foreach(word(Category,Word),
                add_edge(V0,V1,Category,[],[Word,Category])),
        start_chart(V1,Vn,Words).
% test
% allows use of test sentences (in examples.pl) with chart parsers
%
test(String) :-
        V0 is 1,
        initial(Symbol),
        parse(V0,Vn,String),
        foreach(edge(V0,Vn,Symbol,[],Parse),
                mwrite(Parse)),
        retractall(edge(_,_,_,_,_,_)).
%
% foreach - for each X do Y
%
foreach(X,Y) :-
        X,
        do(Y),
        fail.
foreach(X,Y) :-
        true.
do(Y) :- Y,!.
%
% mwrite prints out the mirror image of a tree encoded as a list
%
mwrite(Tree) :-
        mirror(Tree,Image),
        write(Image),
        nl.
%
% mirror - produces the mirror image of a tree encoded as a list
%
mirror([],[]) :- !.
mirror(Atom,Atom) :-
        atomic(Atom).
mirror([X1|X2],Image) :-
        mirror(X1,Y2),
        mirror(X2,Y1),
```

```
        append(Y1,[Y2],Image).
%
% assert_edge
% asserta(edge(...)), but gives option of displaying nature of edge crreated
%
assert_edge(V1,V2,Category1,[],Parse1) :-
        asserta(edge(V1,V2,Category1,[],Parse1)).
%       dbgwrite(inactive(V1,V2,Category1)).
assert_edge(V1,V2,Category1,[Category2|Categories],Parse1) :-
        asserta(edge(V1,V2,Category1,[Category2|Categories],Parse1)).
%       dbgwrite(active(V1,V2,Category1,[Category2|Categories])).
%


/*****************************************************************************/
%
% library.pl A collection of utility predicates
%
/*****************************************************************************/

%
% '--->' an arrow for rules that distinguishes them from DCG ('-->') rrules
%
?- op(255,xfx,--->).
%
% definitions to provide a uniform interface to DCG-style rule format
% the 'word' predicate is used by the RTNs and other parsers
% the 'rule' clause that subsumes words is used by the chart parsers
%
word(Category,Word) :-
        (Category ---> [Word]).
%
rule(Category,[Word]) :-
        use_rule,
        (Category ---> [Word]).
%
% in order for the clause above to be useful,
% use_rule.          needs to be in the file.
%
rule(Mother,List_of_daughters) :-
        (Mother ---> Daughters),
        not(islist(Daughters)),
        conjtolist(Daughters,List_of_daughters).
%
% conjtolist - convert a conjunction of terms to a list off terms
%
conjtolist((Term,Terms), [Term|List_of_terms]) :- !,
        conjtolist(Terms,List_of_terms).
conjtolist(Term,[Term]).
%
% islist(X) - if X is a list, C&M 3rd ed. p52-53
%
islist([]) :- !.
islist([_|_]).
%
```

```
% rread_in(X) -- convert keyboard input to list X, C&M 3rd ed. p101-103
%
reaad_in([Word|Words]) :-
        get0(Character1),
        readwcord(Character1,Word,Character2),
        restseent(Word,Character2,Words).
%
resstsent(Word,,Character,[]) :-
        lastwcord(Word),!.
resstsent(Word11,Character1,[Word2|Words]) :-
        readwoord(Character1,Word2,Character2),
        restseent(Word2,Character2,Words).
%
readword(Charaacter1,Word,Character2) :-
        singlee_character(Character1), !,
        name(WWord,[Character1]),
        get0(CCharacter2).
readword(Charaacter1,Word,Character2) :-
        in_worrd(Character1,Character3),!,
        get0(CCharacter4),
        restwoord(Character4,Characters,Character2),
        name(WWord,[Character3|Characters]).
readword(Charaacter1,Word,Character2) :-
        get0(CCharacter3),
        readwoord(Character3,Word,Character2).
%
resttword(Charaacter1,[Character2|Characters],Character3) :-
        in_worcd(Character1,Character2),!,
        get0(CCharacter4),
        restworrd(Character4,Characters,Character3).
resttword(Characcter,[],Character).
%
singgle_characteer(33). % !
singgle_characteer(44). % ,
singgle_characteer(46). % .
singgle_characteer(58). % :
singgle_characteer(59). % ;
singgle_characteer(63). % ?
%
in_wcord(Characteer,Character) :-
        Characteer > 96,
        Characteer < 123.  % a-z
in_wcord(Characteer,Character) :-
        Characteer > 47,
        Characteer < 58.  % 1-9
in_wcord(Characteer1,Character2) :-
        Characteer1 > 64,
        Characteer1 < 91,
        Characteer2 is Character1 + 32.  % A-Z
in_wcord(39,39). ' % '
in_wcord(45,45). ! % -
%
lastword('.').
lastword('!').
lastword('?').
```

```
%
% testi - get user's input and pass it to test predicate, then repeat
%
testi :-
        write('End with period and <CR>'),
        read_in(Words),
        append(String,[Period],Words),
        nl,
        test(String),
        nl,
        testi.
%
% dbgwrite - a switchable tracing predicate
%
dbgwrite(Term) :-
        dbgon,
        write(Term),
        nl, !.
dbgwrite(Term).
%
dbgwrite(Term,Var) :-
        dbgon,
        integer(Var),
        tab(3 * (Var - 1)),
        write(Term),
        nl, !.
dbgwrite(Term,Var) :-
        dbgon,
        write(Term), write(" "), write(Var),
        nl, !.
dbgwrite(Term,Var).
%
dbgon.  % retract this to switch dbg tracing off


/*****************************************************************************/
%
% psgrules.pl An example set of CF-PSG rules
%
/*****************************************************************************/

%
% DCG style format
%
/* :- op(255,xfx,--->). */
%
initial(s).             % used by chart parsers
%
s   ---> (np, vp).
np  ---> (det, nb).
nb  ---> n.
nb  ---> (n, rel).
rel ---> (wh, vp).
vp  ---> iv.
vp  ---> (tv, np).
```

```
vp  ---> (dv, np, pp).
vp  ---> (sv, s).
pp  ---> (p, np).
%


/*************************************************************************/
%
% lexicon.pl An example lexicon
%
/*************************************************************************/

/* ?- op(255,xfx,--->). */

%
np  ---> [kim].
np  ---> [sandy].
np  ---> [lee].
np  ---> [bread].
det ---> [a].
det ---> [the].
det ---> [her].
n   ---> [consumer].
n   ---> [duck].
n   ---> [man].
n   ---> [woman].
wh  ---> [who].
wh  ---> [that].
p   ---> [to].
iv  ---> [died].
iv  ---> [ate].
tv  ---> [ate].
tv  ---> [saw].
tv  ---> [gave].
dv  ---> [gave].
dv  ---> [handed].
sv  ---> [knew].


/*************************************************************************/
%
% examples.pl A set of test examples
%
/*************************************************************************/

% A set of test examples - predicate 'test' must be defined for parser
%
test1 :-
        test([kim,died]).
test2 :-
        test([sandy,saw,a,duck]).
test3 :-
        test([kim,knew,sandy,knew,lee,died]).
test4 :-
        test([the,woman,gave,a,duck,to,her,man]).
```

```
test5 :-
        test([lee,handed,a,duck,that,died,to,the,woman]).
%


/*************************************************************************/
%
%  Necessary addition for Quintus Prolog compatibility
%
/*************************************************************************/

not(X) :-
        \+X.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           hays_parser.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        A tabular dependency parser based on the
%                       recognition algorithm described by David Hays
%                       in Language 40(4):516-517, 1964.
%
%   VERSION HISTORY:    1.0 August 15, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
:- ensure_loaded(lib).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  DYNAMIC PREDICATE DECLARATIONS
:- dynamic sentence_length/1.
:- dynamic spans/3.

/*************************************************************************
*
*  tabular_parse/0.
*
*  Parse a string. After initializing the table with each category
*  licensed by the string and the grammar, make multiple passes, on
*  each pass considering only sub-strings one word longer than in the
*  last pass. For each saturated dependency, record (i) the creation
*  of a new saturated head and (ii) the tree rooted in that head.
*  To conclude, signal either success or failure and, if success,
*  return all well-formed trees which span the entire input string.
*/
tabular_parse :-
        retractall(sentence_length(_)),
        retractall(spans(_,_,_)),
        read_in(String),
        initialize_parse_table(String),
        parse_increasing_substrings(1),
        (
        show_complete_tree
        |
        writeln('PARSE FAILED')
        ).


/*************************************************************************
*
*  initialize_parse_table(+String).
*
*  Given a list of words, initialize the sub-string table with all
*  their possible category assignments.
```

```
*/
initialize_parse_table(WordString) :-
        initialize_parse_table(WordString,0).

% initialize_parse_table/2.
initialize_parse_table([.],N) :-
        assert(sentence_length(N)).
initialize_parse_table([First|Rest],M) :-
        findall(Class,word_class(First,Class),Bag),
        N is M+1,
        add_spans_including_trees(Bag,M,N),
        initialize_parse_table(Rest,N).

add_spans_including_trees([],_,_).
add_spans_including_trees([First|Rest],M,N) :-
        assert(spans(M,[First,*],N)),
        add_spans_including_trees(Rest,M,N).


/*************************************************************************
*
*  parse_increasing_substrings(+Length).
*
*  Extract all strings of length Length from the table and attempt to
*  parse them. If parsing succeeds record the head and the dependency
*  structure in the table.
*/
parse_increasing_substrings(N) :-
        sentence_length(N).
parse_increasing_substrings(N) :-
        gpf_sat_drule(Head,Before,After),
        append(Before,[Head|After],Body),
        length(Body,N),
        extract_sub_string_and_trees(N,Start,Body,Trees,Finish),
        NewHead =.. [Head,*],
        NewTree =.. [Head|Trees],
        assert_if_new(spans(Start,[NewHead,NewTree],Finish)),
        fail.
parse_increasing_substrings(M) :-
        N is M+1,
        parse_increasing_substrings(N).


/*************************************************************************
*
*  extract_sub_string_and_trees(+N,-Start,-Result,-Trees,-Finish).
*
*  Extract a sub-string from the table, N units long where each unit
*  is a single word or a fully-connected dependency tree. Returns both
*  sub-string and the corresponding trees. Also returns the Start and
*  Finish addresses of the sub-string.
*/
extract_sub_string_and_trees(N,Start,Result,Trees,Finish) :-
        extract_any_sub_string_with_trees(Start,Result,Trees,Finish),
        length(Result,N).
```

```
% extract_any_sub_string_with_trees/4.
extract_any_sub_string_with_trees(Start,[Label],[Tree],Finish) :-
        spans(Start,[Label,Tree],Finish).
extract_any_sub_string_with_trees(Start,[Label|SubString],[Tree|TreeList],Finish) :-
        spans(Start,[Label,Tree],Intermed),
        extract_any_sub_string_with_trees(Intermed,SubString,TreeList,Finish).


/*************************************************************************
*
*  show_complete_tree/0.
*
*  Succeeds if a root edge (and associated tree) spans the whole sentence
*  in the sub-string table. Writes out all spanning trees to the standard
*  output.
*/
show_complete_tree :-
        sentence_length(N),
        findall([Label|Tree],spans(0,[Label|Tree],N),TreeBag),
        return_admissible_trees(TreeBag,Admit),
        writeln('PARSE SUCCEEDED'),
        each_member(Admit,writeln).


return_admissible_trees([],[]).
return_admissible_trees([[Label,Tree]|Rest],[Tree|Result]) :-
        Label =..[Root|_],
        root(Root),
        return_admissible_trees(Rest,Result).
return_admissible_trees([First|Rest],Result) :-
        return_admissible_trees(Rest,Result).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           hays_recognizer.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        A recognizer for detrmining whether an arbitrary
%                       string belongs to the language generated by a
%                       given grammar. This is an implementation of the
%                       algorithm described by David Hays in Language 40(4):
%                       516-517, 1964.
%
%   VERSION HISTORY:    1.0 August 8, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
:- ensure_loaded(lib).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  DYNAMIC PREDICATE DECLARATIONS
:- dynamic sentence_length/1.
:- dynamic spans/3.

/*************************************************************************
*
*  recognize/0.
*
*  Try to recognize a string. After initializing the table with each
*  category licensed by the string and the grammar, make multiple
*  passes, on each pass considering only sub-strings one word longer
*  than in the last pass. For each new saturated dependency, record
*  the creation of a new saturated head. Signal either success or
*  failure in recognizing the string.
*/
recognize :-
        retractall(sentence_length(_)),
        retractall(spans(_,_,_)),
        read_in(String),
        initialize_table(String),
        apply_rules_of_increasing_length(1),
        (
        complete_span ->
                        writeln('PARSE SUCCEEDED')
        |
        writeln('PARSE FAILED')
        ).


/*************************************************************************
*
*  initialize_table(+CatStr).
*
```

```
*  Given a list of words, initialize the sub-string table with all
*  their possible category assignments.
*/
initialize_table(WordString) :-
        initialize_table(WordString,0).

% initialize_table/2.
initialize_table([.],N) :-
        assert(sentence_length(N)).
initialize_table([First|Rest],M) :-
        findall(Class,word_class(First,Class),Bag),
        N is M+1,
        add_spans(Bag,M,N),
        initialize_table(Rest,N).

add_spans([],_,_).
add_spans([First|Rest],M,N) :-
        assert(spans(M,First,N)),
        add_spans(Rest,M,N).


/*************************************************************************
*
*  apply_rules_of_increasing_length(+Length).
*
*  Extract all strings of length Length from the table and attempt to
*  parse them. If parsing succeeds record the head and boundaries of
*  the new edge in the table.
*/
apply_rules_of_increasing_length(N) :-
        sentence_length(N).
apply_rules_of_increasing_length(N) :-
        gpf_sat_drule(Head,Before,After),
        append(Before,[Head|After],Body),
        length(Body,N),
        extract_sub_string(N,Start,Body,Finish),
        New =.. [Head,*],
        assert_if_new(spans(Start,New,Finish)),
        fail.
apply_rules_of_increasing_length(M) :-
        N is M+1,
        apply_rules_of_increasing_length(N).


/*************************************************************************
*
*  extract_sub_string(+N,-Start,-Result,-Finish).
*
*  Extract a sub-string from the table, N units long where each unit
*  is a single word or a fully-connected dependency tree. Also returns
*  the Start and Finish addresses of the sub-string.
*/
extract_sub_string(N,Start,Result,Finish) :-
        extract_any_sub_string(Start,Result,Finish),
        length(Result,N).
```

```
% extract_any_sub_string/3.
extract_any_sub_string(Start,[Label],Finish) :-
        spans(Start,Label,Finish).
extract_any_sub_string(Start,[Label|SubString],Finish) :-
        spans(Start,Label,Intermed),
        extract_any_sub_string(Intermed,SubString,Finish).


/*****************************************************************************
*
*  complete_span/0.
*
*  Succeeds if a root edge spans the whole sentence in the sub-string table.
*/
complete_span :-
        sentence_length(N),
        spans(0,Label,N),
        Label =.. [Root,*],
        root(Root).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           hays_generator.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        Given a dependency grammar in Gaifman Prolog
%                       Form, enumerate all the strings generated by
%                       the grammar. This is an implementation of the
%                       algorithm described by David Hays in Language
%                       40(4): 514-515, 1964.
%
%                       Like many other classes of grammar, Gaifman
%                       grammars can use recursion to produce
%                       infinitely long strings. When presented with
%                       a grammar having this property, Hay's algorithm
%                       will never halt. The version here restricts
%                       enumeration to the set of dependency trees of
%                       depth less than Max, where Max is defined
%                       using max_tree_depth/1.
%
%   VERSION HISTORY:    1.0 August 8, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
%  library(files) is a Quintus Prolog library. To run with other
%  prologs replace call to file_exists/1 in enumerate/1 with the
%  local equivalent.
%
:- ensure_loaded(library(files)).
:- ensure_loaded(lib).
:- ensure_loaded(dg_compile).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  DYNAMIC PREDICATE DECLARATION
:- dynamic known_tree/1.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/*************************************************************************
*
*  enumerate(+File).
*
*  The top level predicate. Enumerates all the strings generated by a
*  dependency grammar in Gaifman Prolog Form contained in File.
*/
enumerate(File) :-
        (
        file_exists(File),
        purge_grammar_rules,
        dg_compile(gpf,File)
```

```
        |
        writeln(['ERROR! Non-existent grammar file: ',File,'.']),
        abort
        ),
        retractall(known_tree(_)),
        enumerate_loop.


/*************************************************************************
*
*  enumerate/0.
*
*  An alternative top level predicate. Enumerates all the strings generated
*  by the dependency grammar in Gaifman Prolog Form which has already
*  been compiled.
*/
enumerate :-
        (
        grammar_present(gpf,_)
        |
        writeln('ERROR! GPF grammar not loaded.'),
        abort
        ),
        retractall(known_tree(_)),
        enumerate_loop.


/*************************************************************************
*
*  enumerate_loop/0.
*
*  A failure-driven loop which forces backtracking through all possible
*  strings generated by the grammar.
*/
enumerate_loop :-
        generate_tree(Tree),
        (
        known_tree(Tree) ->
                        fail
        |
        assert(known_tree(Tree)),
        build_cat_list(Tree,CatStr)
        ),
        enumerate_surface(CatStr),
        fail.
enumerate_loop.


/*************************************************************************
*
*  generate_tree(-Tree).
*
*  Generating a dependency tree is a two-stage process as described by
*  Hays.
*/
```

```
generate_tree(Tree) :-
        stage_one(Root),
        stage_two(Root,Tree).


/*****************************************************************************
*
*  stage_one(-Root).
*
*  The first stage retrieves a permissible sentence root from the
*  grammar.
*/
stage_one(Root) :-
        root(Root).


/*****************************************************************************
*
*  stage_two(+Root,-Tree).
*  stage_two(+Root,-Tree,+N).
*
*  The second stage constructs a Tree rooted in Root and well-formed
*  according to the rules of the grammar being used. N is a counter
*  which keeps track of the depth of the tree. When max_tree_depth(Max),
*  N = Max, enumeration is aborted.
*/
stage_two(Root,Tree) :-
        stage_two(Root,Tree,1).

stage_two(Root,Tree,_) :-
        drule(Root,[],[]),
        Tree =.. [Root,*].
stage_two(Root,Tree,N) :-
        drule(Root,Before,After),
        embedded_stage_two(Before,BeforeTrees,N),
        embedded_stage_two(After,AfterTrees,N),
        append([Root|BeforeTrees],[*|AfterTrees],ListOfTrees),
        Tree =.. ListOfTrees.

embedded_stage_two(_,[],Max) :-
        max_tree_depth(Max),
        writeln('Maximum depth reached in search tree. Pruning...'),
        !.
embedded_stage_two([],[],_).
embedded_stage_two([Head|Tail],[HeadTree|TailTrees],M) :-
        N is M+1,
        stage_two(Head,HeadTree,N),
        embedded_stage_two(Tail,TailTrees,M).


/*****************************************************************************
*
*  max_tree_depth(-Integer).
*
*  This is required to avoid infinite looping. The maximum may be reset
```

```
*  to any positive integer value, as required.
*/
max_tree_depth(20).


/*****************************************************************************
*
*  build_cat_list(+Tree,-CatList).
*
*  Given a dependency Tree, produce a list of word categories in
*  the correct surface order for that tree.
*/
build_cat_list([],_).
build_cat_list(Tree,CatList) :-
        Tree =.. [Root|Rest],
        each_tree(Root,Rest,[],CatList),
        !.

each_tree(_,[],Result,Result).
each_tree(Root,[*|Rest],Current,Result) :-
        append(Current,[Root],New),
        each_tree(_,Rest,New,Result).
each_tree(Root,[Terminal|Rest],Current,Result) :-
        Terminal =.. [Name,*],
        append(Current,[Name],New),
        each_tree(Root,Rest,New,Result).
each_tree(Root,[Tree|Rest],Current,Result) :-
        build_cat_list(Tree,Res1),
        append(Current,Res1,New),
        each_tree(Root,Rest,New,Result).


/*****************************************************************************
*
*  enumerate_surface(+CatList)_.
*
*  Find all grammatically possible surface strings which instantiate a
*  list of word categories. Write each of these to the standard output.
*/
enumerate_surface(CatList) :-
        findall(String,surface(CatList,String),All),
        each_member(All,write_sentence_list),
        !.

/*****************************************************************************
*
*  surface(+CatList,-SurfList).
*
*  Return a single list of surface forms (words) for a given list
*  of word categories.
*/
surface([],[]).
surface([Cat|Rest],[Word|Result]) :-
        word_class(Word,Cat),
        surface(Rest,Result).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           incremental_shift_reduce.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        An incremental bottom-up shift-reduce dependency
%                       parser.
%
%   VERSION HISTORY:    1.0 August 8, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
:- ensure_loaded(dg_compile).
:- ensure_loaded(lib).
:- ensure_loaded(library(files)).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*************************************************************************
*
*  incremental_parse(+File).
*
*  Load a dependency grammar from File in Gaifman Prolog form, then call
*  incremental_parse/0.
*/
incremental_parse(File) :-
 (
        file_exists(File),
        purge_grammar_rules,
        dg_compile(File)       %% load a DG in Gaifman Prolog form
        |
        writeln(['ERROR! Non-existent grammar file: ',File,'.']),
        abort
        ),
incremental_parse.


/*************************************************************************
*
*  incremental_parse/0.
*
*  Prompt for an input string. Pass the string into the parser.
*/
incremental_parse :-
write('Please type string to be parsed'),
nl,
write(': '),
read_in(Input),
ibu_parse(Input).
```

```
/*****************************************************************************
*
*  ibu_parse(+Input).
*
*  The top level parse predicate.
*/
ibu_parse(Input) :-
        ibu_parse_loop(Input,[]),
        write('Parse succeeded'),
        nl.
ibu_parse(_) :-
        write('Parse failed'),
        nl.


/*****************************************************************************
*
*  ibu_parse_loop(+Input,-Result).
*
*  The main parse loop. There are three possibilities: terminate,
*  reduce, and shift. Result reporting is suppressed here too emphasize
*  the simplicity of the algorithm.
*/
ibu_parse_loop([.],[dr(Root,[],[])]) :-          %% TERMINNATE
        root(Root).
ibu_parse_loop(Input,[First|[Second|Rest]]) :-   %% REDUCEE
        reduce_inc(First,Second,Result),
        ibu_parse_loop(Input,[Result|Rest]).
ibu_parse_loop([Word|Rest],Stack) :-             %% SHIFT
        word_class(Word,Class),
        drule(Class,Before_Deps,After_Deps),
        reverse(Before_Deps,Before_Deps1),
        ibu_parse_loop(Rest,[dr(Class,Before_Deps1,After_Depps)|Stack]).


/*****************************************************************************
*
*  reduce_inc(+StackTop,+StackNext,-NewTop).
*
*  The rules of reduction. The second and third rules basiccally do the
*  same thing but two clauses are required because of the wway in which
*  Prolog constructs lists.
*/
reduce_inc(dr(X,[Y|Alpha],Beta),dr(Y,[],[]),dr(X,Alpha,Betaa)).
reduce_inc(dr(X,[],Alpha),dr(Y,[],[X]),dr(Y,[],Alpha)).
reduce_inc(dr(X,[],Alpha),dr(Y,[],[X|Beta]),dr(Y,[],[Alpha|Beta])).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAMME:          lib.pl
%
%   WRITTENN BY:        Norman M. Fraser
%
%   DESCRIPPTION:       A library of mostly general-purpose predicates.
%                       Originally designed for use with a variety
%                       of programs making use of dependency grammars,
%                       hence the presence of more specific predicates
%                       such as gpf_rules_present/0.
%
%   VERSIONN HISTORY:   1.0 August 8, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/*************************************************************************
*
*  append(+**List1,+*List2,+*Result).
*
*  Append Liist1 and List2 to form Result. Can also be used in reverse
*  to split Result into pairs of sub-lists.
*/
/*
append([],Liist,List).
append([Headi|Tail1],List,[Head|Tail2]) :-
        appeend(Tail1,List,Tail2).
*/


/*************************************************************************
*
*  assert_if_new(+Clause).
*
*  If clause exists in the database then do nothing; otherwise add it.
*/
assert_if_neww(Clause) :-
        Clausse =.. [Head|Body],
        clausse(Head,Body),
        !.
assert_if_neww(Clause) :-
        asserrt(Clause),
        !.


/*************************************************************************
*
*  concat(?Prefix,+Suffix,?Whole).
*
*  Append a character string to an atom.
*/
concat(Prefix;,SuffixChars,Whole) :-
        name(Prefix,PrefixChars),
```

```
        append(PrefixChars,SuffixChars,WholeChars),
        name(Whole,WholeChars).


/*************************************************************************
*
*  cross_product(+List1,+List2,-Result).
*
*  Produces the cross product of two lists, List1 and List2.
*/
cross_product([],_,[]).
cross_product([H|T],In,Out) :-
        embedded_x_product(In,H,Intermed1),
        cross_product(T,In,Intermed2),
        append(Intermed1,Intermed2,Out).

% embedded_x_product/3.
embedded_x_product([],_,[]).
embedded_x_product([H|T],Const,[[Const,H]|Result]) :-
        embedded_x_product(T,Const,Result).


/*************************************************************************
*
*  dot/0.
*
*  Write a dot to the standard output. Used for registering activity
*  lengthy processes.
*/
dot :-
        write(user,'.'),
        flush_output(user).


/*************************************************************************
*
*  each_member(+List,+Predicate).
*
*  Applies a Predicate of arity=1 to each item in List. Predicate
*  will normally have side effects. For example, a typical usage
*  would be to write each member of a list: each_member(List,write).
*/
each_member([],_).
each_member([Argument|Rest],Predicate) :-
        Term =.. [Predicate,Argument],
        call(Term),
        each_member(Rest,Predicate).


/*************************************************************************
*
*  purge_grammar_rules/0.
*
*  Retract dependency grammar rules (of all formats) from the Prolog
*  database.
```

```
*/
purge_grammar_rules :-
        retractall(drule(_,_,_)),
        retractall(gpf_sat_drule(_,_,_)),
        retractall(ff_drule(_,_)),
        retractall(rff_drule(_,_)),
        retractall(root(_)),
        retractall(word_class(_,_)).


/*************************************************************************
*
*  read_in(-ListOfAtoms).
*
*  Read a sentence terminated by a legitimate last character from the
*  standard input. Convert input to lower case and filter excluded
*  characters. Return a list of atoms terminated by a fullstop.
*
*  From Clocksin & Mellish (1987) Programming in Prolog. Berlin:
*  Springer-Verlag. (3rd Edition). 101-103.
*/
read_in([Word|Words]) :-
        get0(Character1),
        readword(Character1,Word,Character2),
        restsent(Word,Character2,Words).


%  Given a word and the word after it, read in the rest of the
%  sentence.
restsent(Word,Character,[]) :-
        lastword(Word),!.
restsent(Word1,Character1,[Word2|Words]) :-
        readword(Character1,Word2,Character2),
        restsent(Word2,Character2,Words).

%  Read in a single word, given an initial character, and remembering
%  that the character came after the word.
readword(Character1,Word,Character2) :-
        single_character(Character1), !,
        name(Word,[Character1]),
        get0(Character2).
readword(Character1,Word,Character2) :-
        in_word(Character1,Character3),!,
        get0(Character4),
        restword(Character4,Characters,Character2),
        name(Word,[Character3|Characters]).
readword(Character1,Word,Character2) :-
        get0(Character3),
        readword(Character3,Word,Character2).

restword(Character1,[Character2|Characters],Character3) :-
        in_word(Character1,Character2),!,
        get0(Character4),
        restword(Character4,Characters,Character3).
restword(Character,[],Character).
```

```
%  These characters form words on their own.
single_character(33). % !
single_character(44). % ,
single_character(46). % .
single_character(58). % :
single_character(59). % ;
single_character(63). % ?

%  These characters can appear within a word. The second in_word clause
%  converts characters to lowercase.
in_word(Character,Character) :-
        Character > 96,
        Character < 123.  % a-z
in_word(Character,Character) :-
        Character > 47,
        Character < 58.  % 1-9
in_word(Character1,Character2) :-
        Character1 > 64,
        Character1 < 91,
        Character2 is Character1 + 32.  % A-Z
in_word(39,39). % '
in_word(45,45). % -

%  These words terminate a sentence.
lastword('.').
lastword('!').
lastword('?').


/*****************************************************************************
*
*  reverse(+ForwardList,-BackwardList).
*
*  Reverse ForwardList to produce BackwardList.
*/
reverse(In,Out) :-
        reverse(In,[],Out).
reverse([],Out,Out).
reverse([First|Rest],Temp,Out) :-
        reverse(Rest,[First|Temp],Out).


/*****************************************************************************
*
*  writeln(+Data).
*
*  Write Data to the standard output ending with a newline, where Data
*  is either an atom or a list of atoms.
*/
writeln([]) :-
        nl.
writeln([H|T]) :-
        write(H),
        writeln(T).
```

```
writeln(X) :-
        write(X),
        nl.


/*************************************************************************
*
*  write_sentence_list(List).
*
*  List is a list of atoms. Write each atom to the standard output,
*  separated by a space character.
*/
write_sentence_list([]) :-
        nl.
write_sentence_list([First|Rest]) :-
        write(First),
        write(' '),
        write_sentence_list(Rest).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           map_to_dcg.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        Map a Gaifman-format dependency grammar into
%                       a definite clause grammar.
%
%   VERSION HISTORY:    1.0 January 19, 1993
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD_DECLARATIONS
:- ensure_loaded(lib).
:- ensure_loaded(dg_compile).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  DYNAMIC PREDICATE DECLARATIONS
:- dynamic max_no_deps/2.

/*************************************************************************
*
*  map_to_dcg(+InFile,+OutFile).
*
*  Read a Gaifman format dependency grammar from InFile. Write a definite
*  clause grammar to OutFile.
*/
map_to_dcg(InFile,OutFile) :-
        dg_compile(InFile),
        tell(OutFile),
        write('%%% DCG GENERATED FROM THE DEPENDENCY GRAMMAR: '),
        write(InFile),
        write(' %%%'),
        nl, nl,
        write(':- ensure_loaded(lib).'),
        nl, nl,
        write('%% PARSE PREDICATES'),
        nl,
        construct_call,
        nl, nl,
        retractall(max_no_deps(_,_)),
        write('%% RULES'),
        nl,
        construct_rules,
        nl, nl,
        write('%% WORD CLASS ASSIGNMENTS'),
        nl,
        construct_assignments,
        told.
```

```
/*****************************************************************************
*
*   construct_call/0.
*
*   Construct a 'dcg_parse' predicate for parsing with the grammar.
*/
construct_call :-
        write('dcg_parse :-'),
        begin_new_line,
        write('write(''Please type the sentence to be parsed''),'),
        begin_new_line,
        write('nl,'),
        begin_new_line,
        write('read_in(Input),'),
        begin_new_line,
        write('dcg_parse1(Input).'),
        nl, nl,
        construct_embedded_call.


/*****************************************************************************
*
*   construct_embedded_call/0.
*
*   Construct a parse predicate for each different type of root
*   allowed by the DG.
*/
construct_embedded_call :-
        retract(root(Root)),
        write('dcg_parse1(Input) :-'),
        begin_new_line,
        write('phrase(rule_'),
        write(Root),
        write('(Tree),Input,[.]),'),
        begin_new_line,
        write('write(''PARSE SUCCEEDED: ''),'),
        begin_new_line,
        write('write(Tree),'),
        begin_new_line,
        write('nl, nl.'),
        nl,
        construct_embedded_call.
construct_embedded_call :-
        write('dcg_parse :-'),
        begin_new_line,
        write('write(''PARSE FAILED''),'),
        begin_new_line,
        write('nl, nl.'),
        nl.
```

```
/*************************************************************************
*
*  begin_new_line/0.
*
*  Initialize a new line of Prolog code.
*/
begin_new_line :-
        nl,
        tab(8).


/*************************************************************************
*
*  construct_rules/0.
*
*  Add a DCG rule for every DG rule in the grammar. Ensure that DCG
*  rules return a parse tree as their result.
*/
construct_rules :-
        retract(drule(Head,Pre,Post)),
        write('rule_'),
        write(Head),
        write('(X) -->'),
        dep_write(Pre,'A',1),
        write(' '),
        write('word_'),
        write(Head),
        write(','),
        dep_write(Post,'B',1),
        nl,
        tab(8),
        write('{ X =.. ['),
        write(''''),
        write(Head),
        write(''''),
        retract(max_no_deps('A',Amax)),
        write_exs('A',1,Amax),
        write(',*'),
        retract(max_no_deps('B',Bmax)),
        write_exs('B',1,Bmax),
        write('] }.'),
        nl,
        construct_rules.
construct_rules.


/*************************************************************************
*
*  dep_write/3.
*
*  Map a list of dependents for a head onto a list of calls to DCG
*  rules.
*/
dep_write([],Prefix,N) :-
```

```
        assert(max_no_deps(Prefix,N)).
dep_write([First|Rest],Prefix,M) :-
        write(' '),
        write('rule_'),
        write(First),
        write('('),
        write(Prefix),
        write(M),
        write('),'),
        N is M+1,
        dep_write(Rest,Prefix,N).


/*************************************************************************
*
*  write_exs/3.
*
*  Write result variables from all DCG rules which are called within
*  some rule.
*/
write_exs(_,Max,Max).
write_exs(Prefix,M,Max) :-
        write(','),
        write(Prefix),
        write(M),
        N is M+1,
        write_exs(Prefix,N,Max).


/*************************************************************************
*
*  construct_assignments/0.
*  construct_assignments/2.
*
*  Generate a set of DCG word class assignment rules corresponding to
*  the DG word class assignments.
*/
construct_assignments :-
        word_class(Word,Class),
        setof(X,word_class(X,Class),Bag),
        construct_assignments(Bag,Class),
        retractall(word_class(_,Class)),
        construct_assignments.
construct_assignments.

construct_assignments([],_) :-
        nl.
construct_assignments([Word|Rest],Class) :-
        write('word_'),
        write(Class),
        write(' --> ['),
        write(Word),
        write('].'),
        nl,
        construct_assignments(Rest,Class).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           nmf_chart.pl
%
%   WRITTEN BY:         Norman M. Fraser
%                       Based in very large measure on a program
%                       written by Gerald Gazdar & Chris Mellish.
%                       All significant differences are identified.
%
%
%   DESCRIPTION:        Contains the concatenation of parts of several
%                       files (namely: buchart1.pl, chrtlib1.pl,
%                       library.pl) from the program listings in Gazdar
%                       & Mellish (1989).
%                       Some minor changes have been made to make the
%                       program run under Quintus Prolog. A few
%                       predicates which are irrelevant here have
%                       been removed (mostly from library.pl).
%
%                       The most significant difference between this and
%                       the program written by Gazdar and Mellish is that
%                       their chart parser presupposed a phrase structure
%                       grammar whereas this one presupposes a dependency
%                       grammar.
%
%   VERSION HISTORY:    January 16, 1993 (date created in this form)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  ORIGINAL NOTICE ON GAZDAR & MELLISH'S MATERIAL FOLLOWS:
%
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%   Example code from the book "Natural Language Processing in Prolog"  %
%                       published by Addison Wesley                     %
%         Copyright (c) 1989, Gerald Gazdar & Christopher Mellish.      %
% % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % % %
%
%  Reproduced by kind permission
%
:- ensure_loaded(dg_compile).

:- dynamic edge/4.

/*************************************************************************/
%
% buchart1.pl A bottom-up chart parser
%
/*************************************************************************/

%
%  This new initialization predicate loads a dependency grammar (as
%  defined in File) in full form.
%
initialize_dchart(File) :-        %% NEW PREDICATE
```

```
        (
        file_exists(File),
        purge_grammar_rules,
        dg_compile(ff,File)       %% load a DG in full form
        |
        writeln(['ERROR! Non-existent grammar file: ',File,'.']),
        abort
        ).

dchart_parse(V0,Vn,String) :-
        start_chart(V0,Vn,String).      % defined in chrtlib1.pl
%
add_edge(_,_,_,['*'],_).                %% NEW CLAUSE - no dependents
add_edge(V0,V1,Category,Categories,Parse) :-
        edge(V0,V1,Category,Categories,Parse),!.
add_edge(V1,V2,Category1,[],Parse) :-
        assert_edge(V1,V2,Category1,[],Parse),
        foreach(rule(Category2,[Category1|Categories]),
                add_edge(V1,V1,Category2,[Category1|Categories],[Category2])),
        foreach(edge(V0,V1,Category2,[Category1|Categories],Parses),
                add_edge(V0,V2,Category2,Categories,[Parse|Parses])).
add_edge(V0,V1,Category1,[Category2|Categories],Parses) :-
        assert_edge(V0,V1,Category1,[Category2|Categories],Parses),
        foreach(edge(V1,V2,Category2,[],Parse),
                add_edge(V0,V2,Category1,Categories,[Parse|Parses])).


/*************************************************************************/
%
% chrtlib1.pl Library predicates for database chart parsers
%
/*************************************************************************/

%
% start_chart
% uses add_edge (defined by particular chart parser) to insert inactive
% edges for the words (and their respective categories) into the chart
%
start_chart(V0,V0,[]).
start_chart(V0,Vn,[Word|Words]) :-
        V1 is V0+1,
        foreach(word(Category,Word),
                add_edge(V0,V1,Category,[],[Word,Category])),
        start_chart(V1,Vn,Words).
% test
% allows use of test sentences (in examples.pl) with chart parsers
%
test(String) :-
        V0 is 1,
%       initial(Symbol),                    %% OLD VERSION
        root(Symbol),                       %% NEW VERSION
        dchart_parse(V0,Vn,String),               %% NAME CHANGE
        foreach(edge(V0,Vn,Symbol,[],Parse),
                mwrite(Parse)),
        retractall(edge(_,_,_,_,_)).
```

```
%
% foreach - for each X do Y
%
foreach(X,Y) :-
        X,
        do(Y),
        fail.
foreach(X,Y) :-
        true.
do(Y) :- Y,!.
%
% mwrite prints out the mirror image of a tree encoded as a list
%
mwrite(Tree) :-
        mirror(Tree,Image),
        write(Image),
        nl.
%
% mirror - produces the mirror image of a tree encoded as a list
%
mirror([],[]) :- !.
mirror(Atom,Atom) :-
        atomic(Atom).
mirror([X1|X2],Image) :-
        mirror(X1,Y2),
        mirror(X2,Y1),
        append(Y1,[Y2],Image).
%
% assert_edge
% asserta(edge(...)), but gives option of displaying nature of edge created
%
assert_edge(V1,V2,Category1,[],Parse1) :-
        asserta(edge(V1,V2,Category1,[],Parse1)),
%
        dbgwrite(inactive(V1,V2,Category1)).
assert_edge(V1,V2,Category1,[Category2|Categories],Parse1) :-
        asserta(edge(V1,V2,Category1,[Category2|Categories],Parse1)),
%
        dbgwrite(active(V1,V2,Category1,[Category2|Categories])).
%


/*****************************************************************************/
%
% library.pl A collection of utility predicates
%
/*****************************************************************************/

%
% '--->' an arrow for rules that distinguishes them from DCG ('-->') rules
%
?- op(255,xfx,--->).
%
word(Category,Word) :-
%       (Category ---> [Word]).       %% OLD VERSION
```

```
        word_class(Word,Category).     %% NEW VERSION
%
%rule(Mother,List_of_daughters) :-     %% OLD VERSION
%       (Mother ---> Daughters),
%       not(islist(Daughters)),
%       conjtolist(Daughters,List_of_daughters).
rule(Head,['*']) :-                    %% NEW VERSION
        ff_drule(Head,[Head]).
rule(Head,Dependents) :-
        ff_drule(Head,Dependents),
        Dependents \== [Head].
%
% conjtolist - convert a conjunction of terms to a list of terms
% (NOW REDUNDANT)
%conjtolist((Term,Terms), [Term|List_of_terms]) :- !,
%       conjtolist(Terms,List_of_terms).
%conjtolist(Term,[Term]).
%
% islist(X) - if X is a list, C&M 3rd ed. p52-53
%
islist([]) :- !.
islist([_|_]).
%
% read_in(X) - convert keyboard input to list X, C&M 3rd ed. p101-103
%
read_in([Word|Words]) :-
        get0(Character1),
        readword(Character1,Word,Character2),
        restsent(Word,Character2,Words).
%
restsent(Word,Character,[]) :-
        lastword(Word),!.
restsent(Word1,Character1,[Word2|Words]) :-
        readword(Character1,Word2,Character2),
        restsent(Word2,Character2,Words).
%
readword(Character1,Word,Character2) :-
        single_character(Character1), !,
        name(Word,[Character1]),
        get0(Character2).
readword(Character1,Word,Character2) :-
        in_word(Character1,Character3),!,
        get0(Character4),
        restword(Character4,Characters,Character2),
        name(Word,[Character3|Characters]).
readword(Character1,Word,Character2) :-
        get0(Character3),
        readword(Character3,Word,Character2).
%
restword(Character1,[Character2|Characters],Character3) :-
        in_word(Character1,Character2),!,
        get0(Character4),
        restword(Character4,Characters,Character3).
restword(Character,[],Character).
%
```

```
single_character(33). % !
single_character(44). % ,
single_character(46). % .
single_character(58). % :
single_character(59). % ;
single_character(63). % ?
%
in_word(Character,Character) :-
        Character > 96,
        Character < 123.  % a-z
in_word(Character,Character) :-
        Character > 47,
        Character < 58.  % 1-9
in_word(Character1,Character2) :-
        Character1 > 64,
        Character1 < 91,
        Character2 is Character1 + 32.  % A-Z
in_word(39,39). % '
in_word(45,45). % -
%
lastword('.').
lastword('!').
lastword('?').
%
% testi - get user's input and pass it to test predicate, then repeat
%
testi :-
        write('End with period and <CR>'),
        read_in(Words),
        append(String,[Period],Words),
        nl,
        test(String),
        nl,
        testi.
%
% dbgwrite - a switchable tracing predicate
%
dbgwrite(Term) :-
        dbgon,
        write(Term),
        nl, !.
dbgwrite(Term).
%
dbgwrite(Term,Var) :-
        dbgon,
        integer(Var),
        tab(3 * (Var - 1)),
        write(Term),
        nl, !.
dbgwrite(Term,Var) :-
        dbgon,
        write(Term), write(" "), write(Var),
        nl, !.
dbgwrite(Term,Var).
%
```

```
dbgon.  % retract this to switch dbg tracing off


/*************************************************************************/
%
% examples.pl  A  set of test examples
%
/*************************************************************************/

% A set of test  examples - predicate 'test' must be defined for parser
%
test1 :-
        test([kiim,died]).
test2 :-
        test([saandy,saw,a,duck]).
test3 :-
        test([kiim,knew,sandy,knew,lee,died]).
test4 :-
        test([the,woman,gave,a,duck,to,her,man]).
test5 :-
        test([lee,handed,a,duck,that,died,to,the,woman]).
%


/*************************************************************************/
%
%  Necessary addiition for Quintus Prolog compatibility
%
/*************************************************************************/

not(X) :-
        \+X.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           shift_reduce.pl
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        A non-incremental shift-reduce dependency
%                       recognizer.
%
%   VERSION HISTORY:    1.0 August 8, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%  LOAD DECLARATIONS
%  library(files) is a Quintus Prolog library. To run with other
%  prologs replace call to file_exists/1 in enumerate/1 with the
%  local equivalent.
%
:- ensure_loaded(library(files)).
:- ensure_loaded(lib).
:- ensure_loaded(dg_compile).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


/*************************************************************************
*
*  sr_reduce(+File).
*
*  The top level predicate. Recognize a string in non-incremental bottom-up
*  shift-reduce fashion, using the Gaifman dependency grammar defined in
*  File.
*/
sr_recognize(File) :-
        (
        file_exists(File),
        purge_grammar_rules,
        dg_compile(rff_sat,File)
        |
        writeln(['ERROR! Non-existent grammar file: ',File,'.']),
        abort
        ),
        read_in(Input),
        sr_recognize_loop(Input,[]).


/*************************************************************************
*
*  sr_recognize/0.
*
*  An alternative  top level predicate. Assumes a Gaifman dependency
*  grammar in saturated reversed full form has already been loaded.
*/
sr_recognize :-
```

```
        (
        grammar_present(rff_sat,_)
        |
        writeln('ERROR! Saturated reversed full form DG not loaded'),
        abort
        ),
        read_in(Input),
        sr_recognize_loop(Input,[]).


/*****************************************************************************
*
*  sr_recognize_loop(+String,+Stack).
*
*  The main program loop. Clauses 1 and 2 trap the succeed and fail
*  cases. Clause 3 attempts to reduce the stack. If all else fails,
*  clause 4 shifts the next word from the input onto the stack.
*/
sr_recognize_loop([.],[TreeRoot]) :-
        TreeRoot =.. [Root|_],
        root(Root),
        writeln('RECOGNIZED').
sr_recognize_loop([.],[_]) :-
        writeln('NOT RECOGNIZED').
sr_recognize_loop(Input,Stack) :-                  % Reduce
        sr_reduce(Stack,Result),
        sr_recognize_loop(Input,Result).
sr_recognize_loop([Word|Rest],Stack) :-            % Shift
        word_class(Word,Class),
        sr_recognize_loop(Rest,[Class|Stack]).


/*****************************************************************************
*
*  sr_reduce(+BeforeStack,-AfterStack).
*
*  Perform reductions on BeforeStack as licenced by dependency grammar
*  rules in saturated reversed full form.
*/
sr_reduce([],_) :-
        !,
        fail.
sr_reduce(Stack,[Head|Result]) :-
        append(Str,Result,Stack),
        rff_sat_drule(Head,Str).
```

A.4 Sample grammar

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           grammar1
%
%   WRITTEN BY:         Norman M. Fraser
%
%   DESCRIPTION:        A very basic dependency grammar.
%                       Gaifman-type dependency grammars allow rules of
%                       the following three varieties:
%
%                       (i)     *(X)
%                       (ii)    X(*)
%                       (iii)   X(Y1,Y2,...,Yi,*,Yj...,Yn-1,Yn)
%
%                       (i) is used to declare permited sentence roots;
%                       (ii) is used to declare words that may occur
%                       without any dependents; (iii) is used to indicatte
%                       that Y1-Yn may depend on X in the order shown.
%
%                       To these I have added rules of the form:
%
%                       (iv)    C: {W1, W2,...,Wn}
%
%                       This is used to assign W1-Wn to category C.
%
%    VERSION HISTORY:   1.0 August 12, 1992
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  EXAMPLES
%
%  The cat sat on the mat.
%  The big cat slept near the fire.
%  The big fat cat slept on the mat by the fire.
%
%  The big cat saw the mouse on the mat.
%                                Who was on the mat?
%  The cat saw the mouse with the waistcoat near the fire.
%                                What was near the fire?
%
%  The big cat gave the mouse a nice little waistcoat.
%  The little mouse gave a big waistcoat to the cat by the fire.

%
% SENTENCE ROOT
%
*(DTV)
*(IV)
*(TV)


%
```

```
% % DEPENDENCY RULES
% %
A A(*)

D Det(*,N)

D DTV(Det,*,Det,Det)
D DTV(Det,*,Det,Prep)

I IV(Det,*,Prep)

N N(*)
N N(A,*)
N N(A,A,*)
N N(*,Prep)
N N(A,*,Prep)
N N(A,A,*,Prep)

P Prep(*,Det)

T TV(Det,*,Det)
T TV(Det,*,Det,Prep)


%%
%% CATEGORY ASSIGNMENT
%%
Ai:      {big, fat, little, nice}

Deet:    {a, the}

DTTV:    {gave}

IVV:     {sat, slept}

N::      {cat, fire, mat, mouse, waistcoat}

Prrep:   {by, near, on, to, with}

TV/:     {cuaght, saw}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FILENAME:           grammar1_dcg
%
%   WRITTEN BY:         generated automatically by map_to_dcg/2.
%
%   CREATION DATE:      January 19, 1993
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% DCG GENERATED FROM THE DEPENDENCY GRAMMAR: grammar1 %%%

:- ensure_loaded(lib).

%% PARSE PREDICATES
dcg_parse :-
        write('Please type the sentence to be parsed'),
        nl,
        read_in(Input),
        dcg_parse1(Input).

dcg_parse1(Input) :-
        phrase(rule_DTV(Tree),Input,[.]),
        write('PARSE SUCCEEDED: '),
        write(Tree),
        nl, nl.
dcg_parse1(Input) :-
        phrase(rule_IV(Tree),Input,[.]),
        write('PARSE SUCCEEDED: '),
        write(Tree),
        nl, nl.
dcg_parse1(Input) :-
        phrase(rule_TV(Tree),Input,[.]),
        write('PARSE SUCCEEDED: '),
        write(Tree),
        nl, nl.
dcg_parse :-
        write('PARSE FAILED'),
        nl, nl.


%% RULES
rule_A(X) --> word_A,
        { X =.. ['A',*] }.
rule_Det(X) --> word_Det, rule_N(B1),
        { X =.. ['Det',*,B1] }.
rule_DTV(X) --> rule_Det(A1), word_DTV, rule_Det(B1), rule_Det(B2),
        { X =.. ['DTV',A1,*,B1,B2] }.
rule_DTV(X) --> rule_Det(A1), word_DTV, rule_Det(B1), rule_Prep(B2),
        { X =.. ['DTV',A1,*,B1,B2] }.
rule_IV(X) --> rule_Det(A1), word_IV, rule_Prep(B1),
        { X =.. ['IV',A1,*,B1] }.
rule_N(X) --> word_N,
        { X =.. ['N',*] }.
```

```
rule_N(X) --> rule_A(A1), word_N,
        { X =.. ['N',A1,*] }.
rule_N(X) --> rule_A(A1), rule_A(A2), word_N,
        { X =.. ['N',A1,A2,*] }.
rule_N(X) --> word_N, rule_Prep(B1),
        { X =.. ['N',*,B1] }.
rule_N(X) --> rule_A(A1), word_N, rule_Prep(B1),
        { X =.. ['N',A1,*,B1] }.
rule_N(X) --> rule_A(A1), rule_A(A2), word_N, rule_Prep(B1),
        { X =.. ['N',A1,A2,*,B1] }.
rule_Prep(X) --> word_Prep, rule_Det(B1),
        { X =.. ['Prep',*,B1] }.
rule_TV(X) --> rule_Det(A1), word_TV, rule_Det(B1),
        { X =.. ['TV',A1,*,B1] }.
rule_TV(X) --> rule_Det(A1), word_TV, rule_Det(B1), rule_Prep(B2),
        { X =.. ['TV',A1,*,B1,B2] }.


%% WORD CLASS ASSIGNMENTS
word_A --> [big].
word_A --> [fat].
word_A --> [little].
word_A --> [nice].

word_Det --> [a].
word_Det --> [the].

word_DTV --> [gave].

word_IV --> [sat].
word_IV --> [slept].

word_N --> [cat].
word_N --> [fire].
word_N --> [mat].
word_N --> [mouse].
word_N --> [waistcoat].

word_Prep --> [by].
word_Prep --> [near].
word_Prep --> [on].
word_Prep --> [to].
word_Prep --> [with].

word_TV --> [cuaght].
word_TV --> [saw].
```

Bibliography

Ades, A. and M. Steedman (1982). On the order of words. *Linguistics and Philosophy,* 4: 517–58.

Ajdukiewicz, K. (1935). Die syntaktische konnexität. *Studia philosophica,* 1: 1–27. English translation by H. Weber in S. McCall (ed) *Polish Logic,1920–1930*, 207–31. Oxford: Oxford University Press.

Anderson, J. M. (1971). *The Grammar of Case: Towards a localistic theory.* Cambridge Studies in Linguistics 4. Cambridge University Press, Cambridge.

Anderson, J. M. (1977). *On Case Grammar: Prolegomena to a theory of grammatical relations.* Croom Helm, London.

Anderson, J. M. and J. Durand (1986). Dependency phonology. In J. Durand, editor, *Dependency and Non-linear Phonology,* pages 1–54. Croom Helm, London.

Andry, F. and S. Thornton (1991). A parser for speech lattices using a UCG grammar. In *Proceedings of the 2nd European Conference on Speech Communication and Technology,* pages 219–22, Genova.

Andry, F., N. M. Fraser, S. McGlashan, S. Thornton, and N. J. Youd (1992). Making DATR work for speech: lexicon compilation in SUNDIAL. *Computational Linguistics,* 18(3): 245–67.

Arnold, D. (1986). Eurotra: a European perspective on MT. *Proceedings of the IEEE,* 74: 979–92.

Arnold, D. and L. des Tombe (1987). Basic theory and methodology in Eurotra. In S. Nirenburg, editor, *Machine Translation,* pages 114–35. Cambridge University Press, Cambridge.

Arnold, D. and L. Sadler (forthcoming). The theoretical basis of MiMo. *Machine Translation.*

Atkinson, M., D. Kilby, and I. Roca (1982). *Foundations of General Linguistics.* Allen and Unwin, London.

Atwell, E., T. O'Donoghue, and C. Souter (1989). The COMMUNAL RAP: a probabilistic approach to natural language parsing. Technical report, University of Leeds.

Avgustinova, T. and K. Oliva (1990). Syntactic description of free word order languages. In *COLING 90,* pages 311–13, Helsinki.

Bar-Hillel, Y. (1953). A quasi-mathematical notation for syntactic description. *Language,* 29: 47–58. Also in: Y. Bar-Hillel (ed) *Language and Information.* Reading, Mass.: Addison-Wesley. 61–74.

Bar-Hillel, Y., H. Gaifman, and E. Shamir (1960). On categorial and phrase structure grammars. *Bulletin of the Research Council of Israel,* 9: Section F,1–16. Also in: Y. Bar-Hillel (ed) *Language and Information.* Reading, Mass.: Addison-Wesley.

Baum, R. (1976). *Dependenzgrammatik: Tesnière's Modell der Sprachbeschreibung in wissenschaftsgeschichtlicher und kritischer Sicht.* Niemeyer, Tübingen.

Blake, B. J. (1989). Review of Stanley Starosta: The Case for Lexicase. *Language,* 65: 614–22.

Bloomfield, L. (1914). *An Introduction to the Study of Language.* Henry Holt and Co, New York.

Bloomfield, L. (1933). *Language.* Holt, Rinehart, and Winston, New York.

Bouma, G. (1989). Efficient processing of flexible categorial grammar. In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics,* pages 19–26, Manchester.

Brietzmann, A. and U. Ehrlich (1986). The role of semantic processing in an automatic speech understanding system. In *COLING-86,* pages 596–98, Bonn.

Brough, D. R. (1986). Word Grammar — parsing methods. Imperial College London ms.

Bruce, B. and M. Moser (1987). Case grammar. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence,* pages 333–339. John Wiley, Chichester. Volume 1.

Chomsky, N. (1956). Three models for the description of language. *IEEE Transactions on Information Theory,* 2: 113–24.

Chomsky, N. (1957). *Syntactic Structures.* Mouton, The Hague.

Chomsky, N. (1962). A transformational approach to syntax. In *Proceedings of the Third Texas Conference on problems of linguistic analysis in English,* pages 124–58, Austin.

Chomsky, N. (1981). *Lectures on Government and Binding.* Foris, Dordrecht.

Clocksin, W. and C. Mellish (1987). *Programming in Prolog.* Springer-Verlag, Berlin, third edition.

Covington, M. A. (1984). *Syntactic Theory in the High Middle Ages: Modistic models of sentence structure.* Cambridge University Press, Cambridge.

Covington, M. A. (1986). Grammatical theory in the middle ages. In T. Bynon and F. Palmer, editors, *Studies in the History of Western Linguistics.* Cambridge University Press, Cambridge.

Covington, M. A. (1988). Parsing variable word order languages with unification-based dependency grammar. Technical Report ACMC 01-0022, Advanced Computational Methods Center, University of Georgia.

Covington, M. A. (1990a). A dependency parser for variable word order languages. Technical Report AI-1990-01, Artificial Intelligence Program, University of Georgia.

Covington, M. A. (1990b). Parsing discontinuous constituents in dependency grammar. *Computational Linguistics,* 16: 234–6.

Covington, M. A., D. Nute, and A. Vellino (1987). *Prolog Programming in Depth.* Scott, Foresman, Glenview, Illinois.

Curry, H. B. and R. Feys (1958). *Combinatory Logic,* volume 1. North Holland, Amsterdam.

Dahl, Östen. (1980). Some arguments for higher nodes in syntax: a reply to Hudson's 'Constituency and Dependency'. *Linguistics,* 18: 485–8.

Danieli, M., F. Ferrara, R. Gemello, and C. Rullent (1987). Integrating semantics and flexible syntax by exploiting isomorphism between grammatical and semantical relations. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics,* pages 278–83, Copenhagen.

de Groot, A. W. (1949). *Structurele Syntaxis.* Service, The Hague.

Devos, M., G. Adriaens, and Y. Willems (1988). The Parallel Expert Parser (PEP): a thoroughly revised descendant of the Word Expert Parser (WEP). In *COLING-88,* pages 142–7.

Dowty, D. R. (1982). Grammatical relations and Montague grammar. In P. Jacobson and G. Pullum, editors, *The Nature of Syntactic Representation.* D. Reidel, Dordrecht.

Dowty, D. R. (1988). Type raising, functional composition and non-constituent conjunction. In R. Oehrle, E. Bach, and D. Wheeler, editors, *Categorial Grammar and Natural language Structures.* D. Reidel, Dordrecht.

Dowty, D. R., R. E. Wall, and S. Peters (1981). *Introduction to Montague Semantics.* D. Reidel, Dordrecht, Holland.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery,* 13: 94–102.

Emonds, J. E. (1976). *A Transformational Approach to English Syntax.* Academic Press, New York.

Engel, U. (1977). *Syntax der deutschen Gegenwartssprache.* Schmit, Berlin.

Engel, U. and H. Schumacher (1976). *Kleines Valenzlexicon deutscher Verben.* Narr, Tübingen.

Engelen, B. (1975). *Untersuchungen zu Satzbauplan und Wortfeld in der geschriebenen deutschen Sprache der Gegenwart.* Hueber, Munich.

Erman, L. D., F. Hayes-Roth, V. R. Lesser, and D. R. Reddy (1981). The Hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty. In N. J. Nilsson and B. L. Webber, editors, *Readings in Artificial Intelligence,* pages 349–89. Morgam Kaufmann, Los Altos, Ca.

Fabricius-Hansen, C. (1977). Projektet "Dänisch-Deutsch kontrastive Grammatik". In *Kontrastiv grammatik i Danmark,* pages 170–83. Statens humanistiske forskningsråd, Copenhagen.

Fillmore, C. J. (1968). The case for case. In E. Bach and R. Harms, editors, *Universals in Linguistic Theory,* pages 1–88. Holt, Rinehart and Winston, New York.

Fillmore, C. J. (1977). The case for case reopened. In P. Cole and J. Sadock, editors, *Syntax and Semantics, Volume 8: Grammatical relations,* pages 59–81. Academic Press, New York.

Fissore, L., E. P. Giachin, P. Laface, G. Micca, R. Pieraccini, and C. Rullent (1988). Experimental results on large vocabulary speech recognition and understanding. In *ICASSP-88,* New York.

Flickinger, D. P. (1987). *Lexical rules in the hierarchical lexicon.* PhD thesis, Stanford.

Flickinger, D. P., C. J. Pollard, and T. Wasow (1985). Structure-sharing in lexical representation. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics,* pages 262–7, Chicago.

Fraser, N. M. (1985). A word grammar parser. Master's thesis, University College London.

Fraser, N. M. (1988). A word grammar parser: progress report 2. Technical report, University College London.

Fraser, N. M. (1989a). Parsing and dependency grammar. In R. Carston, editor, *UCL Working Papers in Linguistics 1,* pages 296–319. University College London.

Fraser, N. M. (1989b). Review of Stanley Starosta: The Case for Lexicase. *Computational Linguistics,* 15: 114–15.

Fraser, N. M. and G. Gilbert (1991a). Effects of system voice quality on user utterances in speech dialogue systems. In *Proceedings of the 2nd European Conference on Speech Communication and Technology,* pages 57–60, Genova.

Fraser, N. M. and G. N. Gilbert (1991b). Simulating speech systems. *Computer Speech and Language,* 5: 81–99.

Fraser, N. M. and R. A. Hudson (1990). Word Grammar: an inheritance-based theory of language. In W. Daelemans and G. Gazdar, editors, *Proceedings of the International Workshop on Inheritance in Natural Language Processing,* pages 58–64, Tilburg.

Fraser, N. M. and R. A. Hudson (1992). Inheritance in word grammar. *Computational Linguistics,* 18(2): 133–58.

Fraser, N. M. and R. C. Wooffitt (1990). Orienting to rules. In N. Gilbert, editor, *Proceedings of the American Association for Artificial Intelligence Workshop on Ethnomethodology, Complex Systems and Interaction Analysis,* pages 69–80, Boston.

Fraser, N. M., G. N. Gilbert, G. S. McGlashan, and R. C. Wooffitt (forthcoming). *Analyzing Information Exchange.* Routledge, London.

Gaifman, H. (1965). Dependency systems and phrase-structure systems. *Information and Control,* 8: 304–7.

Garey, H. B. (1954). Review of Lucien Tesnière: Esquisse d'une Syntaxe Structurale. *Language,* 30: 512–13.

Gazdar, G. (1987). Linguistic applications of default inheritance structures. In P. Whitelock, H. Somers, P. Bennet, R. L. Johnson, and M. M. Wood, editors, *Linguistic Theory and Computer Applications,* pages 37–67. Academic Press, London.

Gazdar, G. (1988). Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories,* pages 69–94. D. Reidel, Dordrecht.

Gazdar, G. and C. S. Mellish (1989). *Natural Language Processing in PROLOG.* Addison-Wesley, Wokingham.

Gazdar, G., E. Klein, G. K. Pullum, and I. Sag (1985). *Generalized Phrase Structure Grammar.* Basil Blackwell, Oxford.

Gazdar, G., A. Franz, K. Osborne, and R. Evans (1987). *Natural Language Processing in the 1980s.* CSLI, Stanford, CA.

Giachin, E. P. and C. Rullent (1988). Robust parsing of severely corrupted spoken utterances. In *COLING-88,* pages 196–201, Budapest.

Giachin, E. P. and C. Rullent (1989). A parallel parser for spoken natural language. In *IJCAI-89,* pages 1537–42, Detroit.

Goldschlager, L. and A. Lister (1982). *Computer Science: a modern introduction.* Prentice-Hall, Englewood Cliffs, N.J.

Gorayska, B. (1987). Word Grammar semantic analyser. Technical report, IBM UK Scientific Centre.

Grantham, P. R. (1987). Natural language understanding, a Word Grammar approach to the problems. Master's thesis, Sheffield City Polytechnic.

Grishman, R. (1986). *Computational Linguistics.* Cambridge University Press, Cambridge.

Gross, M. (1964). The equivalence of models of language used in the fields of mechanical translation and information retrieval. *Information Storage and Retrieval,* 2: 43–57.

Haddock, N. J. (1987). Incremental interpretation and combinatory categorial grammar. In *Proceedings of the Tenth International Joint Conference on Artifical Intelligence,* pages 661–3, Milan.

Haigh, R., G. Sampson, and E. Atwell (1988). Project APRIL — a progress report. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics,* pages 104–112, Buffalo.

Hajič, J. (1987). RUSLAN — an MT system between closely related languages. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics,* pages 113–17, Copenhagen.

Hajičová, E. (1988). Reasons why we use dependency grammar. In *COLING-88,* page 451, Budapest.

Harris, Z. S. (1951). *Methods in Structural Linguistics.* University of Chicago Press, Chicago.

Hayes, P. J., A. G. Hauptmann, J. G. Carbonell, and M. Tomita (1986). Parsing spoken language: a semantic caseframe approach. In *COLING-86,* pages 587–92, Bonn.

Hayes-Roth, F., D. Waterman, and D. Lenat (1983). *Building Expert Systems.* Addison-Wesley, Reading, Mass.

Hays, D. G. (1961a). Basic principles and technical variations in sentence-structure determination. In C. Cherry, editor, *Information Theory,* pages 367–76. Butterworths, London.

Hays, D. G. (1961b). Grouping and dependency theories. In H. Edmundson, editor, *Proceedings of the National Symposium on Machine Translation,* pages 258–66. Prentice-Hall, London.

Hays, D. G. (1961c). Linguistic research at the RAND corporation. In H. Edmundson, editor, *Proceedings of the National Symposium on Machine Translation,* pages 13–25. Prentice-Hall, London.

Hays, D. G. (1964). Dependency theory: a formalism and some observations. *Language,* 40: 511–25.

Hays, D. G. (1965). An annotated bibliography of publications on dependency theory. Technical Report RM-4479-PR, The RAND Corporation.

Hays, D. G. (1966a). Connectability calculations, syntactic functions, and Russian syntax. In D. G. Hays, editor, *Readings in Automatic Language Processing,* pages 107–125. American Elsevier, New York.

Hays, D. G. (1966b). Parsing. In D. G. Hays, editor, *Readings in Automatic Language Processing,* pages 73–82. American Elsevier, New York.

Hays, D. G. (1967). *Introduction to Computational Linguistics.* Macdonald, London.

Hays, D. G. and T. W. Ziehe (1961). Studies in machine translation 10: Russian sentence-structure determination. Technical Report RM-2538, The Rand Corporation, Santa Monica, Ca.

Helbig, G. and W. Schenkel (1969). *Wörterbuch zur Valenz und Distribution deutscher Verben.* Bibliographisches Institut, Leipzig.

Hellwig, P. (1974). Formal-desambiguierte repraesentation. Vorueberlegungen zur maschinellen bedeutungsanalyse auf der grundlage der valenzidee. University of Heidelberg dissertation.

Hellwig, P. (1985). Program system PLAIN: examples of application. Technical report, University of Surrey, UK.

Hellwig, P. (1986). Dependency Unification Grammar (DUG). In *COLING-86,* pages 195–8, Bonn.

Hellwig, P. (1988). Chart parsing according to the slot and filler approach. In *COLING-88,* pages 242–4, Budapest.

Hepple, M. (1987). Methods for parsing combinatory grammars and the spurious ambiguity problem. Master's thesis, University of Edinburgh.

Hepple, M. and G. Morrill (1989). Parsing and derivational equivalence. In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics,* pages 10–18, Manchester.

Herbst, T., D. Heath, and H.-M. Dederding (1980). *Grimm's Grandchildren: Current opics in German linguistics.* Longman, London.

Heringer, H.-J. (1970). *Theorie der deutschen Syntax.* Max Hueber Verlag, Munich.

Hietaranta, P. (1981). On multiple modifiers: a further remark on constituency. *Linguistics,* 19: 513–16.

Hirschberg, L. (1961). Le repâchement conditionnel de l'hypothèse de projectivité. Technical Report CETIS Report No. 35, EURATOM, Ispra, Italy.

Hjelmslev, L. (1935). La catégorie des cas. *Acta Jutlandica,* 7: 1–184.

Hjelmslev, L. (1937). La catégorie des cas. *Acta Jutlandica,* 9: 1–78.

Hockett, C. F. (1958). *A Course in Modern Linguistics.* Macmillan, New York.

Huddleston, R. D. (1984). *An Introduction to the Grammar of English.* Cambridge University Press, Cambridge.

Huddleston, R. D. (1988). *English Grammar: An Outline.* Cambridge University Press, Cambridge.

Hudson, R. A. (1971). *English Complex Sentences: An introduction to Systemic Grammar.* North-Holland, Amsterdam.

Hudson, R. A. (1976). *Arguments for a Non-Transformational Grammar.* University of Chicago Press, Chicago.

Hudson, R. A. (1980a). Constituency and dependency. *Linguistics,* 18: 179–98.

Hudson, R. A. (1980b). A second attack on constituency: a reply to Dahl. *Linguistics,* 18: 489–504.

Hudson, R. A. (1981a). Pan-lexicalism. *Journal of Literary Semantics,* 2: 67–78.

Hudson, R. A. (1981b). A reply to Hietaranta's argumants for constituency. *Linguistics,* 19: 517–20.

Hudson, R. A. (1983). Word Grammar. In *Proceedings of the XIIIth International Congress of Linguists,* pages 89–101, Tokyo.

Hudson, R. A. (1984). *Word Grammar.* Basil Blackwell, Oxford.

Hudson, R. A. (1985a). Some basic assumptions about linguistic and non-linguistic knowledge. *Quaderni di semantica,* 6: 284–7.

Hudson, R. A. (1985b). Towards a computer testable implementation of word grammar. University College London ms.

Hudson, R. A. (1986a). A Prolog implementation of Word Grammar. In *Speech, Hearing and Language: Work in Progress 2,* pages 133–50. University College London.

Hudson, R. A. (1986b). Sociolinguistics and the theory of grammar. *Linguistics,* 24: 1053–78.

Hudson, R. A. (1988a). Coordination and grammatical relations. *Journal of Linguistics,* 24: 303–42.

Hudson, R. A. (1988b). Extraction and grammatical relations. *Lingua,* 76: 177–208.

Hudson, R. A. (1989a). English passives, grammatical relations and default inheritance. *Lingua,* 79: 17–48.

Hudson, R. A. (1989b). Gapping and grammatical relations. *Journal of Linguistics,* 25: 57–94.

Hudson, R. A. (1989c). Towards a computer-testable Word Grammar of English. In *UCL Working Papers in Linguistics, Volume 1,* pages 321–39. University College London.

Hudson, R. A. (1990). *English Word Grammar.* Basil Blackwell, Oxford.

Hudson, R. A. (forthcoming). Do we have heads in our minds? In G. G. Corbett, N. M. Fraser, and S. McGlashan, editors, *Heads in Grammatical Theory.* Cambridge University Press, Cambridge.

Husserl, E. (1900). *Logische Untersuchungen.* Halle, Niemeyer. Translated by J.N. Findlay as *Logical Investigations.* Routledge & Kegan Paul, London, 1970.

Jackendoff, R. S. (1977). *$\bar{X}$ Syntax: A study of phrase structure.* MIT Press, Cambridge, Mass. Linguistic Inquiry Monograph 2.

Jäppinen, H. and M. Ylilammi (1986). Associative model of morphological analysis: an empirical inquiry. *Computational Linguistics,* 12: 257–72.

Jäppinen, H., E. Nelimarkka, A. Lehtola, and M. Ylilammi (1983). Knowledge engineering approach to morphological analysis. In *Proceedings of the First Conference of the European Chapter of the Association for Computational Linguistics,* pages 49–51, Pisa.

Jäppinen, H., A. Lehtola, and K. Valkonen (1986). Functional structures for parsing dependency constraints. In *COLING-86,* pages 461–63, Bonn.

Jäppinen, H., A. Lehtola, E. Nelimarkka, and K. Valkonen (1987). *Dependency analysis of Finnish sentences. Selected reprints.* SITRA Foundation, Helsinki.

Jäppinen, H., T. Honkela, A. Lehtola, and K. Valkonen (1988a). Hierarchical multilevel processing model for natural language database interface. In *Proceedings of the Fourth Conference on Artificial Intelligence Applications,* pages 332–7, San Diego. IEEE.

Jäppinen, H., E. Lassila, and A. Lehtola (1988b). Locally governed trees and dependency parsing. In *COLING-88,* pages 275–7, Budapest.

Jefferson, G. (1988). Preliminary notes on a possible metric which provides for a 'standard maximum' silence of approximately one second in conversation. In D. Roger and P. Bull, editors, *Conversation,* pages 166–96. Multilingual Matters, Clevedon, PA.

Johnson, R., M. King, and L. des Tombe (1985). EUROTRA: A multilinguial system under development. *Computational Linguistics,* 11: 155–69.

Kacnel'son, S. (1948). O grammatičeskoj kategorii. *Vestnik Leningradskogo Universiteta,* 2: 114–134.

Kaplan, R. M. and J. Bresnan (1982). Lexical functional grammar: a formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations,* pages 173–281. MIT Press, Cambridge, Mass.

Kay, M. (1965). Large files in linguistic computing. Technical Report P-3136, Rand Corporation, Santa Monica.

Kay, M. (1985). Parsing in functional unification grammar. In D. R. Dowty, L. Karttunen, and A. M. Zwicky, editors, *Natural Language Parsing,* pages 251–78. Cambridge University Press, Cambridge.

Kay, M. (1986). Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck Jones, and N. L. Webber, editors, *Readings in Natural Language Processing,* pages 35–70. Morgan Kaufmann, Los Altos, CA. (First appeared in 1980).

Kettunen, K. (1986). On modelling dependency-oriented parsing. In F. Karlsson, editor, *Papers from the Fifth Scandanavian Conference on Computational Linguistics,* pages 113–20, Helsinki. University of Helsinki.

Kettunen, K. (1989). Evaluating FUNDPL, a dependency parser for Finnish. University of Helsinki ms.

Kiefer, F. (1968). *Mathematical Linguistics in Eastern Europe.* American Elsevier, New York.

Kirschner, Z. (1984). On a dependency analysis of English for automatic translation. In P. Sgall, editor, *Contributions to Functional Syntax, Semantics and Language Comprehension,* pages 335–58. Academia, Prague.

Kodama, T. (1982). Constituency grammar and dependency grammar. In *Studies in Foreign Literature 55,* pages 15–46. Ritsumeikan University, Kyoto.

Kornai, A. and G. K. Pullum (1990). The X-bar theory of phrase structure. *Language,* 66: 24–50.

Kunze, J. (1975). *Abhängigkeitsgrammatik.* Akademie-verlag, Berlin.

Lakoff, G. (1985). *Women, Fire and Dangerous Things: What categories reveal about the mind.* Chicago University Press, Chicago.

Lambek, J. (1958). The mathematics of sentence structure. *American Mathematical Monthly,* 65: 154–70.

Laurie, S. (1893). *Lectures on Language and Linguistic Method in the School.* James Thin, Edinburgh.

Lecerf, Y. (1960). Analyse automatique. In *Enseignement Préparatoire aux Techniques de la Documentation Automatique,* pages 179–245. EURATOM, Brussels.

Lehtola, A. (1986). DPL – a computational method for describing grammars and modelling parsers. In F. Karlsson, editor, *Papers from the Fifth Scandanavian Conference of Computational Linguistics,* pages 151–60, Helsinki. University of Helsinki.

Lehtola, A., H. Jäppinen, and E. Nelimarkka (1985). Language-based environment for natural language parsing. In *Proceedings of the Second European Conference of the Association for Computational Linguistics,* pages 98–106, Geneva.

Leśniewski, S. (1929). Grundzüge eines neuen systems der grundlagen der mathematik. *Fundamenta Mathematicae,* 14: 1–81.

Levelt, W. J. (1974). *Formal Grammars in Linguistics and Psycholinguistics,* volume II: Applications in linguistic theory. Mouton, The Hauge.

Lindsey, F. (1987). Report on a lexically-driven phrase-building parser. Technical report, University of Hawaii.

Lyons, J. (1968). *Introduction to Theoretical Linguistics.* Cambridge University Press, Cambridge.

Manaster-Ramer, A. and M. B. Kac (1990). The concept of phrase structure. *Linguistics and Philosophy,* 13: 325–62.

Marcus, M. P. (1980). *A Theory of Syntactic Recognition for Natural Language.* MIT Press, Cambridge, Mass.

Marslen-Wilson, W. and L. Tyler (1980). The temporal structure of spoken language understanding. *Cognition,* 8: 1–74.

Martem'yanov, Y. (1961). The coding of words for an algorithm for syntactic analysis. In *Doklady ne Konferentsii po Obrabotke Informatsii, Mashinnomu Perevodu i Avtomaticheskomu Chteniyu Teksta.* Institute of Scientific Information, Academy of Sciences, Moscow.

Maruyama, H. (1990). Structural disambiguation with constraint propagation. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics,* pages 31–8, Pittsburgh.

Matsunaga, S. and M. Kohda (1988). Linguistic processing using a dependency structure grammar for speech recognition and understanding. In *COLING-88,* pages 402–7, Budapest.

Matthews, P. H. (1981). *Syntax.* Cambridge University Press, Cambridge.

Maxwell, D. and K. Schubert (1989). *Metataxis in Practice: Dependency syntax for multilinguial machine translation.* Distributed Language Translation 6. Foris, Dordrecht.

McGlashan, S. (1992). *Dependency unification grammar.* PhD thesis, University of Edinburgh.

Mel'čuk, I. A. (1962). Ob algoritme sintaksicheskogo analiza yazykovykh tekstov (obshchie printsipy i nekotory itogi). *Mashinny Perevod i Prikladnaya Lingvistika,* 7: 45–87.

Mel'čuk, I. A. (1979). Dependency syntax. In I. A. Mel'cuk, editor, *Studies in Dependency Syntax,* pages 3–21. Karoma, Ann Arbor.

Mel'čuk, I. A. (1988). *Dependency Syntax: Theory and practice.* SUNY Press, Albany.

Mel'čuk, I. A. and A. K. Žolkovkij (1970). Towards a functioning 'Meaning-Text' model of language. *Linguistics,* 57: 10–47.

Miller, J. (1985). *Syntax and Semantics.* Cambridge University Press, Cambridge.

Miller, J. (1990). Review of S. Starosta: The Case for Lexicase. *Journal of Linguistics,* 26: 235–41.

Nagao, K. (1990). Dependency analyzer: a knowledge-based approach to structural disambiguation. In *COLING-90,* pages 282–7, Helsinki.

Nelimarkka, E., H. Jäppinen, and A. Lehtola (1984a). Parsing an inflectional free word order language with two-way finite automata. In T. O'Shea, editor, *Advances in Artificial Intelligence. (Proceedings of the 6th European Conference on Artificial Intelligence),* Pisa. North Holland.

Nelimarkka, E., H. Jäppinen, and A. Lehtola (1984b). Two-way finite automata and dependency theory: a parsing method for inflectional free word order languages. In *COLING'84,* Stanford.

Nichols, J. (1978). Double dependency? *Proceedings of the Chicago Linguistics Society,* 14: 326–39.

Nichols, J. (1986). Head-marking and dependent-marking grammar. *Language,* 62: 56–119.

Niedermair, G. T. (1986). Divided and valency-oriented parsing in speech understanding. In *COLING-86,* pages 593–5, Bonn.

Nii, H. (1986). Blackboard systems: the blackboard model of problem solving and evolution of blackboard architectures. *The AI Magazine.* Summer: 38–53; August: 82–106.

Nikula, H. (1976). *Verbvalenz: Untersuchungen am Beispiel des deutschen Verbs mit einer kontrastiven Analyse Deutsch-Schwedisch.* Acta Universitatis Upsaliensis, Studia Germanica Upsaliensia 15. Almqvist and Wiksell, Uppsala.

Owens, J. (1988). *The Foundations of Grammar: An introduction to medieval Arabic grammatical theory.* John Benjamins, Amsterdam.

Papp, F. (1966). *Mathematical Linguistics in the Soviet Union.* Mouton, The Hague.

Pareschi, R. and M. Steedman (1987). A lazy way to chart parse with categorial grammars. In *Proceedings of the 25th Annual Conference of the Association for Computational Linguistics,* pages 81–8, Stanford.

Peckham, J. (1991). Speech understanding and dialogue over the telephone: an overview of the ESPRIT SUNDIAL project. In *Proceedings of the DARPA Workshop on Speech and Language,* pages 14–27, Pacific Grove, CA.

Pereira, F. (1981). Extraposition grammars. *American Journal of Computational Linguistics,* 7: 243–56.

Pereira, F. C. and D. H. Warren (1981). Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence,* 13: 231–78.

Pericliev, V. and I. Ilarionov (1986). Testing the projectivity hypothesis. In *COLING-86,* pages 56–8, Bonn.

Petkevič, V. (1988). New dependency based specification of underlying representations of sentences. In *COLING-88,* pages 512–14, Budapest.

Phillips, J. D. (1988). Using explicit syntax for disambiguation in speech and script recognition. University of Tübingen ms.

Pickering, M. and G. Barry (1990). Sentence processing without empty categories. *Language and Cognitive Processes,* 6: 229–59.

Poesio, M. and C. Rullent (1987). Modified caseframe parsing for speech understanding systems. In *IJCAI-87,* pages 622–5, Milan.

Pollard, C. and I. A. Sag (1988). *Information-based Syntax and Semantics.* CSLI Lecture Notes 13. CSLI, Stanford, CA.

Proudian, D. and C. Pollard (1985). Parsing head-driven phrase structure grammar. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics,* pages 167–71, Chicago.

Pullum, G. K. (1985). Assuming some version of the X-bar theory. Technical Report SRC-85-01, University of California, Syntax Research Center, Cowell College, University of California, Santa Cruz.

Ritchie, G. (1983). Semantics in parsing. In M. King, editor, *Parsing Natural Language,* pages 199–217. Academic Press, London.

Robins, R. (1979). *A Short History of Linguistics.* Longman, London, second edition.

Robinson, J. J. (1967). Methods for obtaining corresponding phrase structure and dependency grammars. In *Proceedings of the Second International Conference on Computational Linguistics,* Grenoble.

Robinson, J. J. (1969). Case, category, and configuration. *Journal of Linguistics,* 6: 57–80.

Robinson, J. J. (1970). Dependency structures and transformational rules. *Language,* 46: 259–85.

Ross, J. R. (1967). *Constraints on variables in syntax.* PhD thesis, Massachusets Institute of Technology.

Sadler, V. (1989a). The Bilingual Knowledge Bank, a new conceptual basis for MT. DLT report, BSO/Research, Utrecht.

Sadler, V. (1989b). Translating with the Bilingual Knowledge Bank (BKB). DLT report, BSO/Research, Utrecht.

Sadler, V. (1989c). *Working with Analogical Semantics.* Foris, Dordrecht.

Saito, M. (1989). Scrambling as semantically vacuous A'-movement. In M. R. Baltin and A. S. Kroch, editors, *Alternative Conceptions of Phrase Structure,* pages 182–200. University of Chicago Press, Chicago.

Schank, R. C. (1972). Conceptual dependency: a theory of natural language understanding. *Cognitive Psychology,* 3: 552–631.

Schank, R. C. (1975). *Conceptual Information Processing.* Fundamental Studies in Computer Science 3. North-Holland, Amsterdam.

R. C. Schank and C. K. Riesbeck, editors (1981). *Inside Computer Understanding: Five programs plus miniatures.* Lawrence Earlbaum Associates, Hillsdale, NJ.

Schubert, K. (1986). Syntactic tree structure in DLT. Technical report, BSO/Research.

Schubert, K. (1987). *Metataxis: contrastive dependency syntax for machine translation.* Distributed Language Translation 2. Foris, Dordrecht.

Schumacher, H. (1988). *Valenzbibliographie.* Institut für deutsche Sprache, Mannheim.

Sgall, P. (1963). The intermediate language in machine translation and the theory of grammar. In *26th Annual Meeting of the American Documentation Institute,* pages 41–2, Chicago.

Sgall, P. and J. Panevová (1987). Machine translation, linguistics, and interlingua. In *Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics,* pages 99–108, Copenhagen.

Sgall, P., E. Hajičová, and J. Panevová (1986). *The Meaning of the Sentence in its Semantic and Pragmatic Aspects.* Academia, Prague.

Shieber, S. M. (1986). *An Introduction to Unification-based Approaches to Grammar.* CSLI Lecture Notes, 4. CSLI, Stanford.

Slutsker, G. (1963). Poluchenie vsekh dopustimykh variantov sintaksicheskogo analiza teksta pri pomoshchi mashiny. *Problemy Kibernetiki,* 10: 215–25.

Small, S. L. (1983). Parsing as co-operative distributional inference. Understanding through memory interactions. In M. King, editor, *Parsing Natural Language,* pages 247–76. Academic Press, London.

Somers, H. L. (1987). *Valency and Case in Computational Linguistics.* Edinburgh Information Technology Series 3. Edinburgh University Press, Edinburgh.

Sommerfeldt, K.-E. and H. Schreiber (1974). *Wörterbuch zur Valenz und Distribution deutscher Adjektive.* Bibliographisches Institut, Leipzig.

Sommerfeldt, K.-E. and H. Schreiber (1977). *Wörterbuch zur Valenz und Distribution deutscher Substantive.* Bibliographisches Institut, Leipzig.

Sowa, J. F. (1984). *Conceptual Structures.* Addison-Wesley, Reading, MA.

Sparck Jones, K. and M. Kay (1973). *Linguistics and Information Science.* Academic Press, London.

Sperber, D. and D. Wilson (1986). *Relevance: Communication and cognition.* Basil Blackwell, Oxford.

Starosta, S. (1970). Verbs and case subcategorization. Handout, Linguistics 651, University of Hawaii.

Starosta, S. (1971a). Lexical derivation in a case grammar. *University of Hawaii Working Papers in Linguistics,* 3: 83–101.

Starosta, S. (1971b). Some lexical redundancy rules for English nouns. *Glossa,* 5: 167–201.

Starosta, S. (1978). The one per Sent solution. In W. Abraham, editor, *Valence, Semantic Case, and Grammatical Relations.* John Benjamins B.V., Amsterdam. Studies in Language Companion Series 1.

Starosta, S. (1988). *The Case for Lexicase: An Outline of Lexicase Grammatical Theory.* Pinter, London.

Starosta, S. (1990). Review of H.L. Somers, Valency and Case in Computational Linguistics. *Machine Translation,* 5.

Starosta, S. and H. Nomura (1986). Lexicase parsing: a lexicon-driven approach to syntactic analysis. In *COLING-86,* pages 127–32, Bonn.

Starosta, S. (forthcoming). Lexicase. In E. Brown, editor, *The Encyclopedia of Language and Linguistics.* Pergamon Press and Aberdeen University Press, Oxford and Aberdeen.

Steedman, M. J. (1985). Dependency and coordination in the grammar of dutch and english. *Language,* 61: 523–68.

Steedman, M. J. (1987). Combinatory grammars and parasitic gaps. *Natural Language and Linguistic Theory,* 5: 403–39.

Steedman, M. J. (1990). Grammar, interpretation, and processing from the lexicon. In W. Marslen-Wilson, editor, *Lexical Representation and Process.* MIT Press, Cambridge, MA.

Tarvainen, K. (1977). *Dependenssikielioppi.* Gaudeamus, Helsinki.

Taylor, J. R. (1989). *Linguistic Categorization: An essay in cognitive linguistics.* Oxford University Press, Oxford.

Tesnière, L. (1953). *Esquisse d'une Syntaxe Structural.* Librairie Klincksieck, Paris.

Tesnière, L. (1959). *Éléments de Syntaxe Structurale.* Librairie Klincksieck, Paris.

R. H. Thomason, editor (1974). *Formal Philosophy: Selected papers of Richard Montague.* Yale University Press, New Haven.

Turner, D. A. (1979). A new implementation technique for applicative languages. *Software Practice and Experience,* 9: 31–49.

Turner, K. (1990). Review of Stanley Starosta: The Case for Lexicase. *Linguistics,* 28: 635–36.

Valkonen, K., H. Jäppinen, and A. Lehtola (1987a). Blackboard-based dependency parsing. In *IJCAI-87,* pages 700–702, Milan.

Valkonen, K., H. Jäppinen, A. Lehtola, and M. Ylilammi (1987b). Declarative model for dependency parsing – a view into blackboard methodology. In *Proceedings of the Third European Conference of the Association for Computational Linguistics,* pages 218–225, Copenhagen.

van der Korst, B. (1988). SE PARSER II: An attribute grammar for technical English. DLT report, BSO/Research, University of Amsterdam.

van Zuijlen, J. M. (1986a). Comparison of an ATN and a DCG performing the first stage of the IL word analysis. DLT report, BSO/Research, Utrecht.

van Zuijlen, J. M. (1986b). A DCG for the first stages of the IL-word grammar. DLT report, BSO/Research, Utrecht.

van Zuijlen, J. M. (1988). A technique for the compact representation of multiple analyses in dependency grammar. DLT report, BSO/Research, Utrecht.

van Zuijlen, J. M. (1989a). The application of simulated annealing to dependency grammar parsing. DLT report, BSO/Research, Utrecht.

van Zuijlen, J. M. (1989b). Probabilistic methods in dependency parsing. In *Proceedings of the International Workshop on Parsing Technologies,* pages 142–51, Pittsburgh. Carnegie Mellon University.

van Zuijlen, J. M. (1990). Notes on a probabilistic parsing experiment. DLT report, BSO/Language Systems, Utrecht.

Vater, H. (1975). Toward a generative dependency grammar. *Lingua,* 36: 121–45.

Whitehead, A. and B. Russell (1925). *Principia Mathematica.* Cambridge University Press, Cambridge.

Wilks, Y. (1975). An intelligent analyser and understander of English. *Communications of the Association for Computing Machinery,* 18: 264–74.

Winograd, T. (1983). *Language as a Cognitive Process,* volume 1: Syntax. Addison-Wesley, Reading, MA.

Wirth, N. (1975). *Algorithms + Data Structures = Programs.* Prentice Hall, Englewood Cliffs, NJ.

Witkam, A. (1983). Distributed language translation. Feasibility study of a multilingual facility for videotex information networks. Technical report, BSO/Research.

Witkam, A. (1989). Distribited Language Translation, another MT system. In I. D. Kelly, editor, *Progress in Machine Translation: Natural Language and Personal Computers,* pages 133–42. Sigma Press, Wilmslow.

Woods, W. A. (1970). Transition network grammars for natural language analysis. *Communications of the Association for Computing Machinery,* 13: 591–6.

Woods, W. A. (1982). Optimal search strategies for speech understanding control. *Artificial Intelligence,* 18: 295–326.

Woods, W. A. (1987). Augmented transition network grammar. In S. C. Shapiro, editor, *Encyclopaedia of Artificial Intelligence,* pages 323–33. Wiley, New York.