Methods for Parallelizing Search Paths in Parsing

by

Carl Gustave de Marcken

S.B. Electrical Engineering and Computer Science, Massachussetts Institute of Technology
(1990)

S.B. Mathematics, Massachussetts Institute of Technology
(1991)

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

June, 1993

Signature of Author ____________________
Department of Electrical Engineering and Computer Science
May 5, 1993

Certified by ____________________
Robert C. Berwick
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ____________________
Campbell L. Searle
Chair, Department Committee on Graduate Students

Methods for Parallelizing Search Paths in Parsing

by
Carl G. de Marcken

Submitted to the Department of Electrical Engineering and Computer Science
on May 5, 1993, in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Many search problems are commonly solved with simple combinatoric algorithms that unnecessarily duplicate and serialize work at considerable computational expense. There are a number of techniques available that can eliminate redundant computations and perform remaining operations in parallel, effectively reducing the branching factors of these algorithms. This thesis investigates the application of these techniques to the problem of parsing natural language into grammatical representations. The result is a useful and efficient programming language and compiler that can reduce some of the combinatoric expense commonly associated with principle-based parsing and other generate-and-test search problems. The programming language is used to implement and test some natural language parsers, and the improvements are compared to those that result from implementing more deterministic theories of language processing.

Thesis Supervisor: Robert C. Berwick
Title: Professor of Computer Science and Engineering

Acknowledgments

Among the people who have provided valuable suggestions and criticism regarding this research, I would especially like to thank my advisor Bob Berwick for his support and prodding. Conversations with Dave Baggett and Eric Ristad helped clarify my positions on a variety of topics.

This work has been supported by an Air Force Graduate Laboratory Fellowship.

Contents

Chapter 1

Introduction

Many search problems are commonly solved with simple combinatoric algorithms that unnecessarily duplicate and serialize work at considerable computational expense. There are a number of techniques available that can eliminate redundant computations and perform remaining operations in parallel, effectively reducing the branching factors of these algorithms. This thesis investigates the application of these techniques to the problem of parsing natural language into grammatical representations. The result is a useful and efficient programming language and compiler that can reduce some of the combinatoric expense commonly associated with principle-based parsing and other generate-and-test search problems. The programming language is used to implement and test some natural language parsers, and the improvements are compared to those that result from implementing more deterministic theories of language processing.

1.1 Parsing as a Search Problem

Modern linguistic theory has shifted from rule-based accounts of language to generative *principle-and-parameters* theories that rely on a small set of language-universal principles to explain and predict human linguistic capacity. In these theories cross-linguistic variation is accounted for by differing lexicons and simple parameters in the principles, while the basic innate principles (the *universal grammar*) remain constant. This approach has greater descriptive adequacy, results in a more succinct grammatical representation, and has more plausible learnability requirements for children that grammars built out of thousands of construct-specific rules.

Chapter 2 provides a deeper introduction to principle-based theories, but we present a simple illustration here. Imagine a parser (a computer program that builds grammatical representations from input strings) analyzing the sentence (1).

(1) What book did John see him give Bill?

Among the facts that the parser must be capable of deriving about the sentence are that

(2) a. John did some seeing.
    b. The sentence is a question about the book.
    c. *Him* does not refer to the same person as John or Bill, but is otherwise free to vary.

In one particular principle-based theory, *Government-and-Binding Theory*, a principle named *Theta Criterion* is used to account for (2a); a principle named *Move-α* accounts for (2b) and why *book* appears at the beginning of the sentence even though it appears at the end in *John saw him give Bill the book*; and a principle named *Binding Theory* explains (2c). These and other principles are largely independent of one another but can interact in subtle ways to explain some very complex linguistic phenomena.

In the past most parsing was accomplished using context-free grammars, where the problem of finding a representation for an entire sentence could be divided into the problem of finding representations for subparts of the sentence. This divide-and-conquer approach led to efficient polynomial-time algorithms for parsing. But principle-based theories lead to grammars that seem significantly more complex than simple CFGs; current principle-based generative theories of language take the form of parameterized filters over essentially arbitrary structures. In effect the modern problem of parsing a sentence reduces to finding any representation that can meet about 20 different linguistic criteria. See [4], [5] for discussions of issues in parsing with these theories and several examples of implemented parsers.

The only known viable approach to parsing with current linguistic theories is through generate-and-test methods. Structures are enumerated and passed through filters that represent principles; any structure that passes all of the filters is considered a valid interpretation of the sentence. In order to keep this sort of scheme feasible great effort is usually expended to ensure that the number of structures enumerated is finite, indeed small. This is usually accomplished by carefully precompiling theorems derived from the filters into the generation process, theorems that would eventually eliminate most of the structures later in the process. Since the production of these theorems has not been automated (and probably never will be, since linguists are not bound to any particular representation for their axioms), most systems require significant modification to efficiently handle small changes in the linguistic theory.

Although current theories can seem far removed from earlier models, they nevertheless permit a myriad of optimization techniques. Many of the generators used in parsing (such as the assignment of case and thematic roles to noun phrases) are local and compositional. Thus much of the generation process and be decomposed and structure shared. Shared structure representations when combined with memoized filters allow the filtering of many search paths at once. And most of the filters and generators linguists have proposed are dependent on only a small subset of all the modules in the system. This property means that much of the generation and filtering process can be highly parallelized, eliminating some of the combinatorial explosion inherent in serializing generators.

This is not to say that parsing is not still inherently combinatorial. No small variant of the current linguistic theory is known to be parsable with any method other than exponential-time search. But there are techniques that can be applied to greatly reduce the potential costs of small variations in the theory, and that would allow systems to be built with less effort put into optimization work.

1.2 Generate-and-Test Optimization Techniques

There are a wide variety of problem-specific optimizations that could be used to speed up many parsers, but this thesis will concentrate on building a compiler that can optimize through two fairly general techniques that should apply to any search problem that can be expressed as a dependent network of generators and filters, eliminating redundant computations and performing operations in parallel.

1.2.1 Eliminating Redundant Computations

Many search problems like parsing can be optimized because much of the work done by conventional techniques during the search is repetitious. Generators and filters are being applied to identical arguments several times, or to arguments that differ only in respects irrelevant to the process at hand. This is the result of one generator (or choice point) being local and therefore not influencing the performance of some other group of filters and generators. The most basic approach to eliminating such redundant computation involves a combination of memoization and a restructuring of the searcher dependencies. Instead of performing a depth-first search through each of the generators as most implementations do, a generator is only executed once for each element of the cross product of the sets of elements generated by its parents in the module dependency graph. This eliminates a great deal of computation but can only be done if the results of a generator's execution are memoizable (which is not always easy to guarantee, since any side effects involved in its computation must be reproducible).

1.2.2 Performing Operations in Parallel

Many parsing systems are difficult to adapt to parallel computers because they use side-effects in their generators, which would lead to inconsistencies if two of their output values were utilized concurrently. But some of the same methods that can be used to solve the problem of memoizing generators in an effort to eliminate redundant computations can also be used to replace side-effecting operations with functional ones.

1.2.3 Examples of Optimization Opportunities in a Parser

The current state of the art in linguistically motivated parsers is Sandiway Fong's *Principle and Parameters Parser* [14]. He organizes his parser into modules of generators and filters. Each generator takes in structures and amends them nondeterministically, usually outputting several structures for every one taken in. Filters do not alter structure but may rule out some of their inputs. There is a partial ordering among the filters and generators that reflects module dependencies: filters can not be applied until generators have created the requisite structures, and some generators modify structures that other generators create. Figure 1.1 shows the partial ordering among the modules of Fong's parser.

When Fong's Prolog implementation is actually run on a sentence it chooses a total ordering of modules that obeys the partial ordering and uses it to serialize the modules into a depth-first search. The result is

Figure 1.1: The partial ordering of Fong's parser's modules. Dark rectangles represent generators and light ones represent filters.

that each generator unnecessarily multiplies the resource expenditure of the entire process. For example, while Generator 1 and Generator 2 of Fong's parser both depend on Generator 0 neither depends on the other. If Generator 1 is run before Generator 2 and produces m structures for every one it takes in, then Generator 2 will operate m times in an identical fashion. Since there are filters that depend on the cross-product of the modifications of Generators 1 and 2 some multiplicative factor is conceivably necessary. But if a lot of computation is involved in *deciding what to modify* rather than actually performing the modifications, then each generator would do better to memoize the effects of its call rather than to continually recompute them.

A greater optimization comes from completely separating the computation of truly independent modules. Although Generators 3 and 6 help determine which of the structures produced by Generators 0 and 2 eventually lead to valid parses, the particular branching factors associated with them do not need to cause other generators to do *any* extra work, because no module depends on 3 or 6 and any other module except Generators 0 and 2. To be more concrete, suppose the serialization of Generators 0 and 2 produces n structures. Generator 6 can be run on each of the n structures and if it branches significantly may produce $a \cdot n$ new structures. Some serialization of generators 4, 5 and 7 may also have to be run for each of the n structures (producing $b \cdot n$ new structures), but not each of the $a \cdot n$ ones. The $a \cdot b \cdot n$ different valid parses can be represented and computed at an expense only proportional to $a \cdot n + b \cdot n$. If a and b are large, this is a significant savings over Fong's actual implementation.

1.3 Research Motivations

An important question is *why* we are concerned with improving the search efficiency of principle-based parsers. The current generation of such parsers is not prohibitively slow for many applications, and with expected improvements in computing power any seeming laggardliness will disappear. It is important to realize that all existing parsers implement only a small subset of linguistic theory. As more and more language processes are introduced into the parsing procedure there is going to be a decided loss of efficiency. This is especially true if these processes are non-deterministic. It is important to ensure that the cost of parsing does not increase exponentially as more principles are added to the process.

1.4 Deterministic Generators

Many of the generators involved in the parsing process are non-deterministic. For instance, in the sentence *John said that Bill saw him* the generator that builds structure representing the referent of *him* is going to have to allow for both the possibility that *John* and *him* are coreferent, and the possibility that *him* refers to some extra-sentential entity. The ambiguities the generators must cope with create most of the complexity in the parsing process. By introducing theories with more determinism much of the inefficiency associated with generate-and-test search can be reduced. For instance, by deterministically choosing the antecedent of *him* using some context-based heuristic the ambiguity is eliminated.

Several deterministic theories for different aspects of language are substituted for their more traditional non-deterministic counterparts in some of the experiments described in this thesis, and the results are compared with the efficiency gains that result from improvements in search strategies and techniques. Chapter 5 contains a discussion about the merits of these deterministic variations.

1.5 Thesis Outline

Chapter 2 is an introduction to a form of principles-and-parameters linguistic theory. It contains a discussion of how such theories can be translated into implementations and just what the complexity of those implementations will be. Chapter 3 presents the new search programming language, starting with a simple example search program. It discusses the rationale for various features and how the language is implemented. Chapter 4 then presents a subset of a linguistic theory and a detailed discussion of how some parsers using this theory are implemented in the search language. A variety of features in the search language are profiled on these test parsers and the results are tabulated. The chapter concludes with a qualitative summary of the efficacy of each language feature. Finally, chapter 5 concludes with generalizations from these test results and their implications for generate-and-test implementations of linguistic theories.

Chapter 2

The Computational Nature of Principle-Based Parsing

Most modern generative linguistics has shifted from rule-based transformational theories [6] to more principled and modular accounts of language competence[1], as exemplified by Chomsky's revolutionary *Lectures on Government and Binding* [7]. Following other sciences, it is believed that the observable complexities of syntax are the result of interactions between a small number of innate modules. In this view, simple parameters in the modules or lexicon account for the variations in syntax across languages. This *principle-and-parameters* approach results in more compact, language universal theories with considerably more realistic learnability requirements.

Despite the success of principle-and-parameter theories, the natural-language processing community has been slow to adopt them as language models. Almost all parsers still use thousands of construct-specific rules that can not account for a variety of seemingly complex phenomena. But recently a strong effort has begun to demonstrate the computational viability of the current linguistic approach (see [1], [4], [5], [14] among others).

2.1 Chapter Outline

This chapter examines the computational nature of current principle-and-parameter theories. In particular, section 2.2 gives examples of principles and section 2.3 looks at the basis for the generate-and-test searches that currently seem the most reasonable approach to parsing with principle-based theories.

---

[1] Competence theories seek to describe *what* a person knows, in contrast to performance theories that deal with the more computational issue of *how* a person utilizes that knowledge. Section 2.3 discusses this in more detail.

2.2 Examples of Principles

The principles-and-parameters view of language actually encompasses a broad range of theories. For instance, Generalized Phrase Structure Grammar (GPSG, see [15]) is a computationally motivated theory of language that describes syntax through context-free rules, but ones that are automatically generated via principled "metarules". The enormous size of the resulting rule set brings the claims of GPSG's computational viability into question (see [2] for more details) but the notion that the interaction between a small number of metarules and feature-passing mechanisms results in the diversity of phenomena found in syntax is clearly reminiscent of Chomsky's approach. However, this thesis will concentrate on principles that are less well computationally specified than those in GPSG; the so-called Government-and-Binding (GB) principles that have evolved from [7] and later work have more explanatory power than GPSG does but their computational implementation is not so intuitive. It is these sorts of principles that we will look at, under the assumption that it is more fruitful to bring the domain of linguistics to computer science than the other way around. See [28] for an introduction to GB theory.

Most GB principles are declarative in nature (*Noun phrases receive case*, not *Check that each noun phrase has received case*). They are specified as filters over essentially arbitrary structures. Any structure which passes through each of the filters unscathed is considered a viable interpretation of a sentence. Sentences which have structures that violate a small subset of the principles may be understandable but ungrammatical.

To get a feel for the nature of principles, we present simplified versions of 3 different modules found in a variation of GB theory. It is important to realize that there is no "standard" set of principles and that most descriptions of principles found in the literature contain minor or even major inconsistencies and inadequacies, as would naturally be expected in an active field of research. It is a great challenge to compile a set of internally consistent or complete principles for use in a parser.

2.2.1 Binding Theory

Intuitively, binding theory tries to explain when phrases may or must be coreferent. Possible antecedents of pronouns (*him*, *them*) and anaphors (*herself*, *themselves*) can be deduced from binding theory, as can various properties of *empty categories*, unspoken (phonetically null) elements which nevertheless have full syntactic properties. The examples of (1) should make much of this clear.

(1) a. John thought Bill saw *him*.
    b. John thought Bill saw *himself*.

In (1a) the pronoun *him* may co-refer with *John* or some extrasentential person, but not with *Bill*. However the anaphor *himself* in (1b) has just the opposite property: it must refer to *Bill*. Most versions of binding theory make the distinction through locality conditions. For instance, the theory presented in [7] contains three conditions:

- Condition A: An anaphor must be bound in its governing category.

- Condition B: An pronoun must be free in its governing category.
- Condition C: An R-expression must be free.

For the time being, we can take *bound* to mean coreferent and *free* to mean non-coreferent, and the *governing category* of a phrase to be the smallest clause containing the phrase. *R-expressions* are similar to proper names or other expressions with inherent reference.

The governing category for *him* in (1a) is *Bill saw him.* Condition B dictates that *him* must be free (not bound) inside the governing category, so *him* can not refer to *Bill.* In (1b) Condition A makes just the opposite restriction, that *himself* must refer to *Bill.*

(2) a. Bill was seen *e*.
b. – was seen Bill.

In the account in [7], sentence (2a) contains an unpronounced empty category, denoted by *e*. Empty categories function as markers for argument positions that have had their words "moved" in the observable surface representation. For instance, it is believed that in (2a) the underlying representation is that found in (2b). Empty categories may have many of the same properties that anaphors and pronouns do. In fact, Chomsky's binding theory predicts that the empty category in (2a) is an anaphor. Condition A then explains why it must refer to *Bill* (*i.e.*, why the object of *see* is *Bill*).

In its full glory binding theory is capable of explaining a wide variety of extremely complex phenomena. Figure 2.1 contains most of the key clauses in the binding theory presented in [7]. It is not intended to be enough information to implement or test the theory, only to give a feeling for the declarative nature of the definitions used by linguists.

2.2.2 The Case Filter

Case theory was developed to explain the differences between sentences such as those in (3) (discussion follows [19], section 1.4):

(3) a. It is likely John will leave.
b. * It is likely John to leave.
c. John is likely to leave.

(The asterisk, or star, is a common linguistic notation for expressing an ungrammatical sentence.) (3b) is ungrammatical, which we can take to mean that it violates some linguistic condition. (3a) and (3b) differ only in the tense of the subordinate clause. The grammatical relations assigned are identical in both sentences, and in the semantically identical (3c) the *to leave* causes no difficulties. So why can't *John* appear in the subject position of a tenseless clause?

Assign numerical indices freely to all noun phrases, subject to Conditions A, B and C. Any noun phrases with the same index are coreferent.

- **Condition A:** An anaphor must be bound in its governing category.
- **Condition B:** An pronoun must be free in its governing category.
- **Condition C:** An R-expression must be free.
- **Binding:** α binds β iff α c-commands β and α and β are coindexed. If α is not bound, it is free.
- **C-Command:** For α, β nodes in a tree, α c-commands β iff every branching node dominating α dominates β and neither α nor β dominates the other.
- **Coindexing:** α and β are coindexed iff they bear the same numerical index.
- **Governing Category:** β is a governing category for α iff β is the minimal category containing α, a governor for α, and a SUBJECT accessible to α.
- **Government:** α governs β iff α c-commands β, α is either N, V, A or P and no maximal projection dominates β that does not dominate α. (see section 2.2.3).
- **SUBJECT:** Subjects and agreement are SUBJECTs.
- **Accessibility:** A SUBJECT α is accessible to β if α c-commands β and assignment to β of the index of α would not result in a violation of the i-within-i filter.
- **i-within-i:** Any node with index i that dominates another node with index i is ungrammatical.
- **Agreement:** Agreement and subject are coindexed.

Figure 2.1: A version of Binding Theory.

The hypothesized solution is a condition, the *Case Filter*, requiring that overt (pronounced) noun phrases receive an abstract property called *Case*. Tensed verbs assign case (in English, nominative case) to their subjects. Prepositions and transitive verbs assign oblique and accusative case to their objects. In possessive constructs, the possessor receives genitive case. No noun phrase may be pronounced in any position that does not receive case through one of these mechanisms. In the case of example (3a) *John* receives case from the tensed verb *will leave*, and the pleonastic *it* receives case from the copula *is*. But in (3b) the infinitive *to leave* does not assign case to *John*, and the caseless *John* violates the Case Filter. This problem is rectified in (3c), where *John*, which still bears the semantic role of subject of *to leave*, is moved to a position where it can receive case from *is*.

Like binding theory, the Case Filter is used to uniformly account for a great number of seemingly disparate phenomena. For instance, in

(4) a. Who did John say *e* is clever.
b. * Who did John say *e* to be clever.

(4) the grammaticality contrast seems to be the same as in (3). The structural location where *who* receives its grammatical role, in the subject position of *is clever*, is not assigned case in (4b) but is in (4a). So it seems that some empty categories function in the same way as noun phrases with respect to the Case Filter, and indeed this generalization explains many grammaticality judgements.

The motivation for the Case Filter is contested. Some say it is a strictly phonological condition, others an ingredient in a more involved process leading to semantic interpretation. Regardless of interpretation, it functions as a simple filter on the allowed output of the generation process, just as the binding conditions act as filters on the interpretation of anaphoric relations.

2.2.3 X-Bar Theory

X-Bar Theory forms the frame that many other parts of linguistic theory are built on. It describes grammatical sentences at the tree structure level, through a small set of context-free grammar (CFG) like rules. Slighly simplified, these recursive rules are

- $\overline{\overline{X}} \Rightarrow$ SPECIFIER $\overline{X}$
- $\overline{X} \Rightarrow X$ COMPLEMENT
- $\overline{X} \Rightarrow \overline{X}$ ADJUNCT

Here, $\overline{\overline{X}}$ indicates a phrasal (maximal) level category (SPECIFIERs, COMPLEMENTs and ADJUNCTs are all phrasal level categories) and $\overline{X}$ indicates a *bar* level category. X ranges over all basic category types, such as nouns ($\overline{\overline{N}}$, $\overline{N}$, N), verbs, adjectives, determiners, inflection and complementizers (N,V,A,D,I,C).

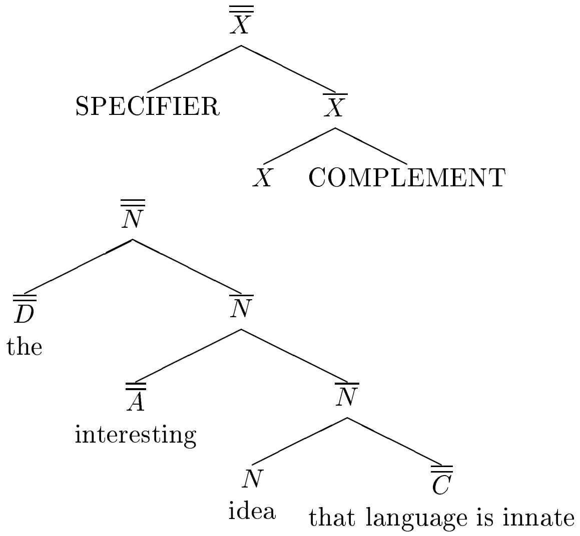
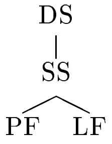


Figure 2.2: The basic X-Bar frame and a sample instantiation.

The right hand sides of these rules are ordered, but their order is parameterized and can very across languages. For instance, English is a right-branching language in which the order is essentially that given. In other languages, the rule ordering may vary by category. In Japanese the objects in verb phrases occur after the subject but before the verb (SOV), unlike the SVO order of English. This is accounted for by the $\overline{V} \Rightarrow$ COMPLEMENT V rule of Japanese. Figure 2.2 presents the basic X-Bar frame and an example of how a sample phrase might be structured.

2.2.4 Move-α

Move-α is a very succinctly phrased portion of the Governement-Binding theory that has extraordinary consequences. It states: *move anything anywhere*! To understand just what this implies, it is necessary to understand the different linguistic levels at which linguistic structure is used. Currently 4 levels are hypothesized: logical form (LF), phonetic form (PF), deep structure (DS), and surface structure (SS). Logical form is the level at which interpretation takes place, and certain relations such as scope among quantifiers is specified. Phonetic form is the level at which an utterance is pronounced. Deep structure is the level at which argument relations between objects are set out. These levels are related to each other in the following way



The relation between structure at the different levels is not arbitrary. For the present purposes let us just look at DS and SS. The deep structure for a question like *who did John see?* might be that found in (5a) where as the surface structure that in (5b).

(5) a. John did see who.
b. Who$_1$ did$_2$ John e_2 see e_1.

Looking at the differences between (5a) and (5b) it is plain that both *who* and *did* are in different positions, and that (5b) now contains two empty categories. Move-α, which allows arbitrary movement, permits the relocation of *who* and *did* in SS to positions other than those they are in in DS. However, *Did who John see* is still ungrammatical at SS. This is because the extraordinary power seemingly granted by move-α is constrained by a number of principles. *Did who John see* is ruled out because there is no position to the left of *who* for *did* to move to. Other constraints are that movement must leave an empty category behind (a *trace* of the movement); it must only move Xs and $\overline{\overline{X}}$s and leave traces of the same category; and movement can not be to an occupied position.

All together the constraints specified by linguists temper the unlimited movement possibilities down into a small number of relatively simple patterns. Among these are *head movement* (movement of *did* in (5)), which is short distance movement of basic lexical categories like verbs and explains such phenomena as the verb-second/verb-final alteration in German; *A-movement*, the short distance movement of $\overline{\overline{X}}$s to argument positions, which explains how objects get to subject positions in passive sentences; and *$\overline{A}$-movement*, the much longer distance movement of $\overline{\overline{X}}$s to non-argument positions exemplified in questions like (5b).

2.2.5 Uniformity of Principles

As should be clear from the principles superficially described above, linguists do not necessarily express their theories in any well-specified formal language, and may casually use terms with far-reaching computational implications. Indeed, this freedom has allowed linguistic theory to undergo radical changes in a very short time period, and it appears to be continuing its evolution (see [9] for significant recent shift) with fundamental new characteristics introduced every few years. It is not an easy thing for a computational linguist to pin down the field long enough to evaluate the state of affairs. This makes it difficult to develop any computational models of language with long-term relevance. Chapter 5 expands upon this difficulty.

2.3 Parsing with Principles

The innate principles governing language use, as described by linguists, do not explicitly state what computational mechanism implements them. This is the distinction between *competence* theories and *performance* theories. For many of the principles, such as move-α, it is not at all clear how the principles affect either parsing or generation of language from a computational perspective. There are some recent principle-based linguistic theories which seem on the surface to be more computationally well specified (see, for instance, [21], who tries to enumerate exactly all the elements of her theory) but these theories are often not as rich or as descriptively adequate as others. How can one translate a generative linguist's description of language competence into a form suitable for use in a computer parser?

2.3.1 Language Complexity

First of all, it is important to look at human languages to see how complex they really are. For instance, it is conceivable that the principles are in extension identical to a simple context-free grammar. Were this the case, then it would presumably be more expeditious to use such an equivalent grammar to describe language for computational purposes than the principles as they are written. This is much of the motivation behind GPSG. It does not in fact seem that human language is context-free, or at least not for any reasonably small grammar ([3]). Let us look, as [3] did, at the intricacies of human language from another perspective, that of computational complexity. We can ask whether it is likely that there are any simple polynomial time algorithms for parsing sentences. Many principles are stated in terms of non-deterministic search (*e.g.*, the *assign numerical indices freely to all noun phrases* of binding theory). Is inefficient non-deterministic search really necessary for deciphering utterances, or is it merely the derivative of a notational convenience?

Very few results about the computational complexity of language exist. Most results are dependent on a particular theory of language. But recently Ristad ([25]) has presented a proof that binding theory is NP-complete[2] that does not depend on any formalism. He uses only basic facts about the possible antecedents of pronouns and anaphors in English to show that one can translate the NP-complete 3-SAT problem into a question of whether or not a sentence permits a valid assignment of antecedents for all the pronouns and anaphors in it. Since such antecedent computation is an incontestable part of our language faculty, this seems to indicate that language is at least NP-complete, and therefore that efficient deterministic algorithms for parsing can not exist. In other words, search is necessary. But Ristad's proof is not convincing. Without going into Ristad's impressive argument, it is possible to point out the general flaw in his argument.

Ristad presents a variety of sentences containing pronouns and anaphors, and points out the restrictions on co-reference between them (the facts that binding theory seeks to capture). For instance, he presents

(6) Before Mark, Phil and Hal were friends, he$_1$ wanted him$_2$ to introduce him$_3$ to him$_4$.

Ristad makes the point that (6) is not easy to understand (NP-complete problems are not easy to solve), and that certain restrictions hold between the various pronouns. For instance, $him_1 \neq him_2$, $him_2 \neq him_3$, $him_2 \neq him_4$ and $him_3 \neq him_4$. These facts can be verified by attempting to read the sentence with the various pronouns bound to different combinations of the three names. By combining simple sentences with clear linguistic judgements to produce more complex sentences, Ristad is able to encode any 3-SAT problem as the problem of determining whether or not a sentence has a valid set of interpretations for

---

[2] *NP-complete* problems are a class of problems for which no known polynomial time algorithms exist, and correspond to problems in which a solution can be guessed and efficiently verified to be correct. These problems are all equivalent to each other within a constant factor of processing complexity, and showing that an NP-complete problem can be translated into another form is a proof that that form is also at least as hard as NP-complete problems.

all of its pronouns. But is this in fact a computation a human is capable of performing? Let us look at one of Ristad's examples.

(7) *Romeo* wanted *Juliet* to love *him* before *e* wanting *himself* to.

In English, *Romeo* can not refer to *Juliet*; *Juliet* can not refer to the same thing as *him*; *him* can not refer to the same thing as *himself*; *Romeo* refers to the same thing as the empty category *e*; and *e* refers to the same thing as *himself*. Ristad makes the point that these facts imply by transitivity that *Romeo* can not refer to the same thing as *him*. This is interesting, because in the prefix sentence *Romeo wanted Juliet to live him* the interpretation of *him* and *Romeo* coreferent is natural.

The flaw in Ristad's argument stems from the fact that he assumes English speakers are capable of calculating the complete antecedent relationships for sentences such as (7). But most listeners in fact initially interpret *him* and *Romeo* as coreferent and then become confused or find the sentence ungrammatical. It is true that if one reads the sentence knowing the correct assignments then one can verify its correctness and the sentence is processable. But most speakers on initial attempt will read up to *him*, assign the antecedent *Romeo* to it, and get confused later in the sentence when that possibility is precluded. There is no indication that English speakers are capable of searching the entire space to find a valid interpretation after this initial failure.

Just what is the implication of this? It is likely that when English speakers come upon a pronoun they use some sort of heuristic for picking its antecedent ([18], [16]) and then verify that all binding theory conditions are satisfied. A computer model without the proper heuristic algorithm (and the exact heuristic would likely require semantic or phonological information unavailable to current computer implementations) will have to perform search to find a proper antecedent assignment for pronouns, and as Ristad proved this is going to be an NP-complete task. It is not the case that a reasonable theory of language needs to be NP-complete, because people can not understand sentences like (7). Human language processing undoubtedly involves heuristics that need to be part of a linguistic theory if the theory is to adequately model human performance. So we can expect that given the current state of knowledge, search will be necessary for an ignorant computer to process binding relationships (and other parts of language), but that it is completely reasonable to expect that with more understanding an efficient computer model of human language could be built, one that did not involve search.

2.3.2 Generate and Test

The previous section makes the point that current computer implementations of language capabilities are likely to require search. Linguists seem to make this assumption when they write descriptions of language. For instance, binding theory is expressed in terms of filters on antecedent relationships. Implicitly or explicitly, some statement must be made of how these relationships are hypothesized. Usually this is in terms of some statement such as *assign numerical indices freely to all noun phrases*. Move-α states *move anything anywhere*. These formulations follow the traditional computer science paradigm of *generate-and-test*: generate potential solutions, and later test them to make sure they are correct. If potential solutions can be efficiently written down and efficiently tested, then this formulation is equivalent to that of NP-complete problems, no more or less powerful than any other techniques sufficiently powerful for search. So generate-and-test implementations are a natural form for computer implementations of linguistic theories to take. This research (see [14], [4] for discussions) takes this approach to parsing, non-deterministically generating possible parses and then efficiently applying the tests (filters) of linguistic theory to verify correctness.

2.3.3 The Computational Nature of Generators

The intuitive notion of a generator is a procedure for producing structure that represents a solution to a problem. In the case of parsing, generators produce linguistic structures that represent the important relationships between words in an utterance. A generator need not produce a complete structure. By following the natural modularity proposed by linguists, a binding theory generator might produce only indices for noun phrases, given a list of these phrases, and leave the generation of quantifier scoping to some other generator.

In the generate-and-test paradigm it is usually assumed that generators are non-deterministic (they produce many possible structures for a given input), but the notion of generator can be useful for even deterministic parts of a theory. For instance, the Case Filter applies to noun phrases after case assignment. Case assignment is usually assumed to be unambiguous and deterministic. For a given phrase in a given structural relationship with a case-assigner, either a specific case is assigned or it isn't, and this can be computed exactly without guessing. But it is still a convenient abstraction to have a case generator that assigns case and a case filter that applies to the result, even if the two could be merged together or even combined with the original phrase-structure generation.

The branching factor of generators is the single most important indication of the computational cost of parsing with a linguistic theory. Deterministic generators such as case assignment can be implemented quite efficiently, and if necessary could be merged with other modules. But non-deterministic generators such as the one that assigns binding indices tend to multiply work done by other modules and are also more dependent on input length. For instance, doubling the number of noun phrases in a sentences is only going to cause a case assignment generator to do twice as much work, but the number of possible assignments of numerical indices to noun phrases grows exponentially with the number of noun phrases in a sentence. For this reason when one examines the efficiency of a theory of language from a computational viewpoint, one should concentrate on those modules that require non-deterministic guessing and not worry about the total number of filters or generators.

2.3.4 The Computational Nature of Filters

Filters are conditions on structures built by generators. They are applied to the output of one or more generators and if they fail, the structures built by those generators are rejected. The number of filters found in a linguistic theory, like the number of generators, is somewhat arbitrary, since most conditions on structures can be divided up into a number of special case filters or combined into a single broad filter with substructure. Computationally the particular modularization of filters is much less important than the structure of generators in a parser. One important exception to this generalization is that if a filter can be made dependent on as few generators as possible, then it can be used early in the process of generation to eliminate unnecessary work by other modules.

A filter that is dependent on only one generator can always be merged with the generator. This can lead to important efficiency gains, because it is possible that a filter can rule out a generator's output part-way through the generation process. For instance, if the Case Filter is only dependent on the deterministic case assigner, which in turn is dependent only on phrase structure, then the Case Filter can be *interleaved* with the process of generating phrase structure to rule out *John to sleep in a bed* before the process of generating phrase structures for the sentence is complete. As each phrase is created, case criteria can be checked on its components, and if the Case Filter is violated then the phrase is rejected.

Filters with multiple dependencies on non-deterministic generators require a multiplicative amount of work and are therefore far more fundamental to the computational complexity of a theory than ones with single dependencies. If there were no module (filter or generator) that depended on each of two non-deterministic generators, then the two generators could be executed completely independently and no combinatorial amount of work would result. But with a single multiply-dependent filter the multiplicative cross-product of results must be computed.

2.3.5 Derived Principles

It is not at all obvious that linguistic theory would be so cleanly divided into modules were that not a principle goal of researchers. It is difficult to produce psycholinguistic or neurological evidence that shows the brain is divided on these lines. Although the divisions permit a reasonably parsimonious and clear description of part of language competence, it is always possible that the processes the brain uses to compute language cross these divisions. It might be that by "compiling" the effects of different principles together a single, efficient procedure for computing language results. Correa ([10]) has taken this approach and produced efficient, deterministic algorithms for computing an approximation of the effects of move-α and several other modules. Obviously if it could be proven that these procedures were identical in extension to the sum of the original linguistic modules, then they would be a great vindication of the computational tractability of linguistic theory and would show that little or no search is involved in language.

Fong ([14]) argues that the obvious attractions of these derived principles of Correa's are overshadowed by their shortcomings. Specifically, he claims that Correa's algorithms are descriptively inadequate (they *are not* equivalent to the original theory formulations, let alone human competence) and that in general the effort and complexity involved in compiling these derived principles prevents them from being understood, proved accurate, or kept current with improvements in linguistic theory. It is certainly true that, for instance, Correa's move-α algorithm is descriptively inadequate. It cannot handle sentences with multiple gaps, such as *Which book did you file e without reading e?*, which the original linguistic theory does explain. But no linguistic theory is perfect, or the field would have ceased to exist. And it would be strange indeed if the subset of language actually used by people did not have an efficient implementation (though it may be that some search is involved).

As an example of a derived principle, let us look at Correa's *structural determination* algorithm for empty category typology. Section 2.2.1 briefly mentions that empty categories have some of the same properties as anaphors and pronouns. The following sentences contain 4 different empty categories, or positions where noun phrases seem to receive their semantic roles but are not pronounced.

(8) a. I asked about *e* seeing John.
b. John was seen *e*.
c. *e* saw John.
d. Who did John see *e*?.

The empty positions marked in (8) by *e* need not be interpreted as full syntactic entities (though Government-Binding theory treats them as such). For our purposes what is important is that regardless of the theory there is some relationship between these argument positions and unspoken noun phrases. Sentence (8a) has the interpretation that the subject role of *see* is arbitrarily interpreted. This is reflected by there being an independent empty category in the subject position of *seeing*. In (8b) *John* receives the semantic role it would normally get in an active sentence from being in the position that *e* is in. Sentence (8c) is ungrammatical in English, but in a language like Spanish or Italian it can mean *I saw John*, where the reference of the empty subject can be determined by the morphology of the verb. And in (8d) *e* is in the position that *who* gets its semantic interpretation from. These 4 empty categories represent 4 different ways that a noun phrase can receive semantic interpretation in a location different from where it is lexically realized. What is particularly interesting is that many of the restrictions on the relationships between the empty positions and the noun phrases can be explained by assuming that there are 4 different types of empty categories, two which are pronominal and two which are anaphoric: {`+anaphor`, `+pronominal`} (8a); {`+anaphor`, `-pronominal`} (8b); {`-anaphor`, `+pronominal`} (8c); {`-anaphor`, `-pronominal`} (8d). For instance, in passive sentences like (8b) the fact that the moved phrase must be close to the position it receives its thematic role from is explained by the fact that as an anaphor, binding theory requires its empty category to have a local binder. The longer distance relationship in (8d) is permitted because the empty category, being neither anaphoric nor pronominal, is not restricted by binding theory. The untensed nature of the embedded sentence in (8a) is the result of a complex interaction of principles that apply to a category that is both anaphoric and pronominal.

An important issue is how an empty category has its anaphoric and pronominal properties determined. A natural answer is that it is by function- any empty category used in a question relationship (as in (8d)) must be `-anaphoric`, `-pronominal` by stipulation. This idea, proposed by Chomsky in [8], is called *functional determination*. Another possibility is that the typology is *freely determined*: any possibility is possible, but all but the correct will be ruled out by some principle or other. For instance, in (8b) the empty category can not be pronominal, because then it would be locally bound by *John*, in violation of binding theory. These two possibilities have great implications for parsing. In the functional determination case the exact relationships the category enters into must be calculated before the empty category typology can be computed. This can be computationally expensive, since many filters can not be applied before the typology is known. In free determination the typology can be non-deterministically guessed, which allows the filters to be applied earlier but results in overgeneration early on. The two possibilities have different empirical predictions.

Correa has created an algorithm derived from a number of different principles that uses clues about the structural position an empty category is in to determine its typology. This algorithm states that an empty category is an anaphor if it is in a position that can receive a thematic role and is either not assigned case or not in a particular structural relation (*government*) with a verb or preposition. An empty category is a pronoun if it is not in the same government relation with a verb or preposition. These conditions can be computed easily without knowledge of the thematic or binding relations that the empty category enters into, and the algorithm greatly cuts down search later in the parsing process. According to Fong, the algorithm is not always as correct as the original formulation, and it does not explain, for instance, how it might be relaxed to handle a slightly ungrammatical sentence.[3]

We explore Correa's deterministic algorithms in our tests (see Chapter 4) because of the great potential they and other derived principles have.

2.4 Variability within the Generate-and-Test framework

The generate-and-test framework permits many possible implementations of a given linguistic theory, even following the linguists' language closely. For instance, nothing has been said about whether principles have to be applied to entire sentences at once. From an implementors vantage it may be simpler to apply generators and filters a sentence at a time, but psychologically this is implausible. People are capable of interpreting sentences and detecting problems in sentences early on as they read them word by word. This is an indication that principles should be applied step by step. Crocker [11] makes this explicit in a framework motivated by psycholinguistic results that forces every module to apply incrementally. Fong also discusses how some principles can be interleaved with the initial phrase-structure generation process, and the potential gains in efficiency that result. In the experiments in this thesis no effort is made to implement linguistic theory in what seems to be a psychologically plausible manner, since these experiments are concerned with making principle-based parsing efficient and not in making psychological predictions.

---

[3] One possibility is that the algorithm acts as a search ordering heuristic rather than an absolutism. The setting that the deterministic algorithm chooses is searched before the alternate. If the search is halted after a single solution has been found, this produces an algorithm that can handle any sentence that free determination or functional determination can, and is much more efficient in the general case.

Chapter 3

A Programming Language for Search Problems

Implementing principle-based parsers is difficult without a proper substrate, one that permits non-deterministic generate-and-test search in a rich programming environment. In this chapter we present a search language specifically designed for making modularized search problems easy to write and efficient to execute. It includes a variety of efficiency-motivated programming constructs that are particularly useful for processing linguistic structure.

3.1 Chapter Outline

The chapter starts off with a crossword puzzle example that motivates many of the programming language features and provides code from a sample search program that solves the puzzle. This section provides brief overviews of major language features. Then section 3.3 discusses internal representations used during search. Explicit definitions of how generators, filters and search strategies are defined in the language are found in sections 3.4 and 3.5. Finally, other language features are discussed, such as failure propagation (3.6), memoization (3.7) and concurrency (3.8.)

3.2 A Crossword Puzzle Example

Imagine trying to solve a simple crossword puzzle, such as that shown in figure 3.1. Essentially a search needs to be done, iterating through a word list to find a set of four different words that fit in the spaces provided, under the constraint that certain letters of each word be identical . There are a variety of traditional techniques for solving the problem. These include

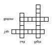


Figure 3.1: A simple crossword puzzle.

- *Depth First Search:* Pick a possible word for one of the spaces, say the top row. Then go through the word list again looking for a suitable candidate for the left column, then the right column and bottom row. At every step (or at the end) verify that all the constraints have been satisfied, and if no word can be found to fit in a space, backtrack and change a previous word.
- *Dynamic Depth First Search (Constraint Satisfaction Search):* Perform a similar search, but dynamically order the selection of spaces according to various heuristics such as how many words are available for the space. Instead of backtracking to the most recent space, pick the most recent space which is directly responsible for current failures.

Both of these techniques will most likely perform unnecessary work when trying to find all solutions to this crossword puzzle. Imagine a depth first search that first selects the word *aardvark* for the top row. It moves on to select the left column and searches the lengthy word list for all words 6 letters long, the second of which must be *r*. For each of these words it performs a similar search on the right column, and then proceeds with the final bottom row. If there are 342 possible 6 letter words with *r* as a second letter, then it will search the right column 342 times and find the same set of words every time, since *aardvark* hasn't changed.[1]

What most search methods lack that is relevant to this problem is the ability to encode the notion that the left and right columns of the crossword puzzle may be searched independently, and the results need not be combined until the bottom row is searched. The programming language discussed here is expressly designed to make such (in)dependencies explicit, in a generate and test approach to search.

3.2.1 An Implementation

Let us look at how this crossword puzzle problem might be expressed in our programming language. First of all, *generators* must be defined for all of the spaces (see figure 3.2). Then *filters* must be defined

---

[1] In the constraint satisfaction literature the method known as arc-consistency is in part designed to alleviate this problem, by permanently pruning items from a node's search space, but it would not work in this case because all computation on the right and left columns is dependent on *aardvark*.

```
(defgenerator TOP-WORD ()
  (return-result (a-member-of *word-list*)))

(defgenerator LEFT-WORD (top-word)
  (let ((possible-left-word (a-member-of *word-list*)))
    (when (eq (second possible-left-word) (third top-word))
      (return-result possible-left-word))))

(defgenerator RIGHT-WORD (top-word)
  (let ((possible-right-word (a-member-of *word-list*)))
    (when (eq (second possible-right-word) (seventh top-word))
      (return-result possible-right-word))))

(defgenerator BOTTOM-WORD (left-word right-word)
  (let ((possible-bottom-word (a-member-of *word-list*)))
    (when (and (eq (first possible-bottom-word) (fifth left-word))
               (eq (fifth possible-bottom-word) (fifth right-word)))
      (return-result possible-bottom-word))))
```

Figure 3.2: Generators for the four crossword puzzle spaces.

(figure 3.3) and an explicit search strategy mapped out (figure 3.4). In this particular problem the exact division between generators and filters is somewhat arbitrary. A given generator could return all words, all words that match up with previously hypothesized words for other spaces in the puzzle, or all words that match up and are of the requisite length. We choose to have the generators pay attention to previously hypothesized words, but to use explicit filters to ensure that words are of the proper length for the space they must fit in.

Generators

The role of generators is to produce any number of structures that will be used by other generators and filters. In this simple example the generators are essentially themselves just filters on the words contained in `*word-list*`, though in a more complex situation they could utilize side effects and return locally constructed data structures.

In more detail, for every environment (collection of input arguments) in which the generator is executed a data structure called a *set* is created, and all the values resulting from the generator's execution are stored in this set, each with a list of closures that represents all side-effects executed to produce the value. This set can later be used to enumerate the structures for filtering or the execution of other generators. Sections 3.3.1 and 3.3.2 contain a further elaboration of this process.

The programming language uses as a base SCREAMER([26]), an extension of COMMON LISP. SCREAMER's compiler automatically CPS[2] converts programs, thereby allowing non-deterministic functions. Thus the

---

[2] CPS (Continuation Passing Style) conversion involves rewriting programs so that after each result is computed, a

```
(deffilter TOP-WORD-LENGTH (top-word)
  (unless (= (length top-word) 8)
    (reject)))

(deffilter LEFT-WORD-LENGTH (left-word)
  (unless (= (length left-word) 6)
    (reject)))

(deffilter RIGHT-WORD-LENGTH (right-word)
  (unless (= (length right-word) 8)
    (reject)))

(deffilter BOTTOM-WORD-LENGTH (bottom-word)
  (unless (= (length bottom-word) 7)
    (reject))
  (succeed))
```

Figure 3.3: Filters for the crossword puzzle problem.

effect of the non-deterministic function `a-member-of` in the `TOP-WORD` generator is essentially that of

```
(defgenerator TOP-WORD ()
  (dolist (temp *word-list*)
    (return-result temp)))
```

Filters

Filters are used to selectively delete values from sets produced by generators. In the example, each value is a list of letters. To permanently delete a value, the function `reject` is called inside the filter. If `reject` is not called the value remains in the set.

Dependency Declarations and Search Strategies

Because it is not always possible for the language to infer all module dependencies from the input arguments to each generator and filter, dependencies between modules must be explicitly declared with the function `define-tree-positions`. One way to look at these dependencies is that each generator must have an input stream and an output stream. The output stream is named with the generator's name. The input stream can be the output stream from a single generator if there is only one dependency,

---

function (the *continuation*) that represents "how the result is used" is called with it. Calling a continuation more than once results in exhaustive non-deterministic programs.

```
(define-tree-positions
  '((top TOP-WORD)
    (generate top-word LEFT-WORD)
    (generate top-word RIGHT-WORD)
    (cross left-word right-word LEFT-AND-RIGHT)
    (generate left-and-right BOTTOM-WORD)))

(defsearcher CROSSWORD-PUZZLE
  ((generate TOP-WORD
     (filter TOP-WORD-LENGTH top-word)
     (generate LEFT-WORD
       (filter LEFT-WORD-LENGTH left-word))
     (generate RIGHT-WORD
       (filter RIGHT-WORD-LENGTH right-word))
     (generate LEFT-AND-RIGHT)
     (generate BOTTOM-WORD
       (filter BOTTOM-WORD-LENGTH bottom-word)))))
```

Figure 3.4: A search strategy for the crossword puzzle problem.

or a new stream can be created by *crossing* (in the sense of *cross-product*) two others if a generator has multiple dependencies. In the example (figure 3.4), the `TOP-WORD` generator is declared to be the *top* generator, with no input dependencies (no input arguments). The `LEFT-WORD` and `RIGHT-WORD` generators take their input from the `TOP-WORD` generator's output stream. The output streams from `LEFT-WORD` and `RIGHT-WORD` are crossed to produce a new stream, `LEFT-AND-RIGHT`, which is used as input to the `BOTTOM-WORD` generator. It is not necessary to declare the input dependencies of filters.

Even with the generator dependencies declared, the precise ordering of the search is underspecified. A specific search program is created with the `defsearcher` macro. In the example found in figure 3.4 the `CROSSWORD-PUZZLE` program performs the search as follows: first the `TOP-WORD` generator is executed, and for each of the values generated, the `TOP-WORD-LENGTH` filter is applied. So long as the filter does not reject the value generated by `TOP-WORD` the `LEFT-WORD` generator is executed on the value the `TOP-WORD` generator produced. Its results are also filtered, and the set of values that results is stored away. The same process occurs with the `RIGHT-WORD` generator and `RIGHT-WORD-LENGTH` filter. At this point there exists a set of left-words and a set of right-words each associated with a given top-word. The command `(generate LEFT-AND-RIGHT)` forms the cross product of these two sets. For each member of this cross product set, the `BOTTOM-WORD` generator and `BOTTOM-WORD-LENGTH` filter is applied. Any value that is not rejected by this filter, when combined with the associated top, left and right words, constitutes a solution to the crossword puzzle.

Notice that while the left and right word sets were computed and filtered independently, they were stored permanently so that the cross product could be rapidly enumerated for the bottom word generator. A slight change in the program, to

```
(defsearcher CROSSWORD-PUZZLE
```

```
((generate TOP-WORD
   (filter TOP-WORD-LENGTH top-word)
   (generate LEFT-WORD
     (filter LEFT-WORD-LENGTH left-word)
     (generate RIGHT-WORD
       (filter RIGHT-WORD-LENGTH right-word)
       (generate LEFT-AND-RIGHT
         (generate BOTTOM-WORD
           (filter BOTTOM-WORD-LENGTH bottom-word)))))))
```

would have resulted in a standard depth first search.

3.3 Internal Representations

One reason simple search strategies are so effective is that they have little computational overhead associated with them- a depth first search needs only to maintain a simple stack. More sophisticated search strategies require the building of complex internal representations. This language must permanently record all results produced by generators, and must maintain a dependency structure between those values in order for them to be efficiently re-enumerable, and for certain types of search failures to be detected. This section describes the internal representations built by search programs written in the language, and how those representations are manipulated.

3.3.1 Value Pairs and Side Effects

Each generator executes some actions, which may include performing destructive side-effects, and returns a series of values. Because other generators and filters will make reference to these values, and because it is important for the values to be efficiently regeneratable, the generator must essentially be memoized: the resultant values and their associated side effects must be stored. We'll call the aggregate structure of a value and a sequence of side-effects a *value pair*. Value pairs are created every time `return-result` is called inside of a generator, as done in the crossword example figure 3.2.

Fortunately, SCREAMER's implementation provides convenient access to side effects. SCREAMER uses backtracking to simulate non-determinism. Each side effect (usually a `setq` form) is replaced by a combination of the usual destructive operator and a push of a closure onto a global stack called `*trail*`. The closure, when executed, effectively undoes the side-effect. When SCREAMER backtracks it executes these closures as it unravels the stack and thus computations do indeed seem completely non-destructive. We extend this mechanism by providing two alternative destructive operators into COMMON LISP, `set!` and `set!-local`. `set!-local` pushes two closures onto a second stack, `*effects*`. The first closure is the same undo closure that SCREAMER uses; the second closure, when executed, performs the original side-effect. In short, the `*effect*` stack contains not only a record of how to undo all side-effects executed during the current thread of non-determinism, but also an efficient encoding of how to perform the side-effects themselves. When a generator outputs a value, the current value of `*effects*` (a list) is joined with the value to produce a value pair. `set!` is the same as `set!-local` except that the undo

```
(defgenerator TEST ()
  (let ((simple-list (list 'empty 'empty)))
    (set! (first simple-list) (either 1 2))
    (set!-local (second simple-list) (either 'A 'B))
    (return-result simple-list)))
```

In the following table, $\mathcal{X}$ represents a particular list structure originally created by `(list 'empty 'empty)` that is modified during the execution of the `TEST` generator, and $\mathcal{Y}$ represents the second cons cell in that list.

Value Pair #	Value	Effect Closures	
1	$\mathcal{X}$	To Do:	`((lambda () (rplaca` $\mathcal{Y}$ `'A)) (lambda () (rplaca` $\mathcal{X}$ `1)))`
		To Undo:	`((lambda () (rplaca` $\mathcal{Y}$ `'empty)) NIL)`
2	$\mathcal{X}$	To Do:	`((lambda () (rplaca` $\mathcal{Y}$ `'B)) (lambda () (rplaca` $\mathcal{X}$ `1)))`
		To Undo:	`((lambda () (rplaca` $\mathcal{Y}$ `'empty)) NIL)`
3	$\mathcal{X}$	To Do:	`((lambda () (rplaca` $\mathcal{Y}$ `'A)) (lambda () (rplaca` $\mathcal{X}$ `2)))`
		To Undo:	`((lambda () (rplaca` $\mathcal{Y}$ `'empty)) NIL)`
4	$\mathcal{X}$	To Do:	`((lambda () (rplaca` $\mathcal{Y}$ `'B)) (lambda () (rplaca` $\mathcal{X}$ `2)))`
		To Undo:	`((lambda () (rplaca` $\mathcal{Y}$ `'empty)) NIL)`

Figure 3.5: A simple generator and the value pairs it produces.

closure is not created. This is more efficient in cases where it is not necessary to undo effects upon backtracking. SCREAMER offers a similar distinction.

To illustrate how the effect-recording system works, figure 3.5 contains an example of a simple generator and the value pairs it produces. `either` is a SCREAMER function that non-deterministically returns one of its arguments. Notice that because one of the destructive operations uses `set!` and not `set!-local`, there are fewer undo closures than do closures. The language also defines many other common destructive operators, such as `push!`, `push!-local`, `pop!`, `pop!-local`, `inc!`, `inc!-local`, etc.

The overhead of creating closures and adding them to the `*effects*` stack for every simple side-effect can be significant for a program, and because of this for many search problems an extremely simple control mechanism such as that found in depth first searches may well be a more efficient mechanism. But if a generator needs to be executed many times, it is very likely that the low cost of regenerating these memoized values will outweigh the initial overhead. This is particularly true for generators that perform complex calculations but a relatively small number of side-effects.

To use a value found in a value pair, the side effecting closures must first be executed in reverse order (from the bottom of the `*effects*` stack to the top, to recreate the proper temporal sequence). At this point the structure found in value portion of the value pair is in an identical state to when it was returned from the generator originally. Before another value from a different value pair may be examined, the undo closures from the original value pair must be executed.

The issues that arise with the interaction between side-effects and concurrent execution are discussed in section 3.8.

3.3.2 Sets, Elements, Dependencies and Enumeration

All the value pairs produced by a generator must be stored in some sort of aggregate if they are to be re-enumerated. This aggregate is called a *set*. Whenever a generator function is applied to some input arguments, a new set is created. That set will contain all the value pairs produced by the generator's application. In addition the set includes dependency information pointing back to the generator's input arguments, and information concerning the state of the generation process.

To encapsulate the arguments that a generator takes, a data structure called an *element* is used. Generators get their input from the output of zero or more other generators. For example, the `TOP-WORD` generator in figure 3.2 has no input dependencies; `LEFT-WORD` gets its arguments from one other generator; and `BOTTOM-WORD` from two other generators. In the case of no input arguments a special element type exists, a *topmost-element*. For the simple case where one generator feeds directly into another, the *generated-element* type is used, and when the outputs of several different generators must be combined to produce a single element, the *crossed-element* is used.

When a generator is executed its value pairs are encapsulated in *generated-elements*, and then these are placed in a *set*. Other single-argument generators can get their input arguments from these generated-elements. But in the case of a generator like `BOTTOM-WORD` that takes several input arguments, generated-elements must be combined into *crossed-elements*. Crossed-elements are built up in a binary tree fashion: a crossed element can be created from any combination of two generated-elements or crossed-elements. The purpose of this crossing procedure is to create a single aggregate element that represents the results of several different generators. When a generator is executed and creates a new *set* of value pairs, that *set* contains information pointing back to the single *element* that represents all of the generator's input arguments. In this way dependency information between all value pairs is maintained.

Figure 3.6 depicts the internal representation of the search in the crossword example.

A generator like `BOTTOM-WORD` needs input elements that have multiple dependencies. This is done by creating a sort of virtual generator, `LEFT-AND-RIGHT` in the example. This virtual generator produces a set of elements, just like a real generator, only the elements are members of two sets, not just one, and hence the element has dependencies from two generators. One of the element's parent sets is associated with the `LEFT-WORD` generator and the other with the `RIGHT-WORD` generator. This can be seen by looking at the `LEFT-AND-RIGHT` sets in figure 3.6.

We can now provide explicit definitions of sets and elements. This is done in figure 3.7. Let us examine these definitions in more detail.

- *Sets: Sets* hold the list of values (*elements*) produced by a generator. The *elements* slot points to a list of all the elements contained in the set. The *tree-position* slot contains information about what generator produced the set, primarily for debugging purposes. The *parent-element* is the element that contained the input arguments for the generator that produced this set, in most cases. For

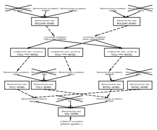


Figure 3.6: Some objects constructed during the crossword search. Each *element* is drawn with a descriptive name and its type, each *set* in a box with the name of its position in the dependency tree and its type. Crossed out elements have been deleted by filters.

indexing sets (explained below) the slot is used in a slightly different manner. *fully-generated?* is true if the generation process is complete. *deleted?* is true if the set has been deleted (see section 3.6).

- *Generated Sets:* Generated sets are just a subclass of sets that are directly created by a generator, as opposed to indexing sets.
- *Indexing Sets:* Indexing sets are built when the results of several generators need to be combined. When a crossed position is generated, an *indexing-set-from-A* is built for each element of one parent generator, and an *indexing-set-from-B* for each element of the other.
- *Simple Elements: Elements* are used to represent the input arguments for generators and to hold value produce by generators.. Simple elements are the parents of all other elements. The *deleted?* slot is true if the element has been deleted. The *daughter-sets* slot contains all the sets that have been generated from the element.
- *Topmost Elements:* Topmost elements are elements with no dependencies and no associated value pairs, used to represent the dependencies of a top level generator.
- *Generated Elements:* Generated elements are created when a generator is applied to its input arguments. The *value-pair* slot contains the resulting value pair, and the *associated-set* is the set that the element is put into.
- *Crossed Elements:* Crossed elements represent a dependency on several different generators, and hence have two associated sets, *index-set-A* and *index-set-B*.

Enumeration

The representation built up during the search (figure 3.6) can be used to efficiently enumerate partial and complete solutions. Let us look at a search program for the crossword puzzle again:

```
(defsearcher CROSSWORD-PUZZLE
  ((generate TOP-WORD
     (filter TOP-WORD-LENGTH top-word)
     (generate LEFT-WORD
       (filter LEFT-WORD-LENGTH left-word))
     (generate RIGHT-WORD
       (filter RIGHT-WORD-LENGTH right-word))
     (generate LEFT-AND-RIGHT)
     (generate BOTTOM-WORD
       (filter BOTTOM-WORD-LENGTH bottom-word)))))
```

Generating the `TOP-WORD` position is easy, because it has no input arguments. But look at the generation of `BOTTOM-WORD`. The generator function must be called on every possible pair of left and right words, which are in turn dependent on the top word. So before `BOTTOM-WORD` can be generated, `TOP-WORD` must be re-enumerated. For every one of the elements produced by this enumeration, a set for the `LEFT-WORD` position and a set for the `RIGHT-WORD` position exists. Either of these can be used to further enumerate

```
;;;
;;; SETs
;;;

(defclass set ()
  ((elements :accessor set-elements :initform nil :type list)
   (tree-position :accessor set-tree-position :initarg :tree-position :type symbol)
   (parent-element :reader parent-element :initarg :parent-element :type simple-element)
   (fully-generated? :accessor fully-generated? :initform nil :type symbol)
   (deleted? :accessor deleted? :initform nil :type symbol)))

(defclass generated-set (set) ())
(defclass indexing-set (set) ())
(defclass indexing-set-from-A (indexing-set) ())
(defclass indexing-set-from-B (indexing-set) ())

;;;
;;; ELEMENTs
;;;

(defclass simple-element ()
  ((deleted? :accessor deleted? :initform nil :type symbol)
   (daughter-sets :accessor daughter-sets :initform nil :type (or cons nil))))

(defclass topmost-element (simple-element) ())

(defclass generated-element (simple-element)
  ((value-pair :reader element-value-pair :initarg :value :type value-pair)
   (associated-set :accessor associated-set :initarg :associated-set :type set)))

(defclass crossed-element (simple-element)
  ((index-set-A :reader index-set-A :initarg :index-set-A :type indexing-set)
   (index-set-B :reader index-set-B :initarg :index-set-B :type indexing-set)))
```

Figure 3.7: The definitions of sets and elements.

the tree. Say the `LEFT-WORD` position is chosen. Then it is enumerated, producing a number of elements. For each of these elements there exists a *indexing-set-from-A* set that represents the `LEFT-AND-RIGHT` tree position. This is then enumerated, and the elements produced contain backpointers that allows the `RIGHT-WORD` position to be set. Once all of these tree positions are set, then the `BOTTOM-WORD` generator can be applied.

In general enumeration can be complex. The precise ordering of tree positions in the enumeration process can affect the efficiency of a search.

3.4 Specification of Generators and Filters

The crossword example generator and filter definitions in figures 3.2 and 3.3 are fairly self-explanatory, but we provide a slightly more precise description of the search language syntax here.

Generators are defined with the `defgenerator` macro, which takes the form `(defgenerator name (&rest positions) &rest body)`. `positions` is a list of the generator's input arguments, which must be the names of tree positions, or a list of a variable name to bind locally and a tree position name. For instance,

```
(defgenerator CHAIN-FORMATION (PHRASE-STRUCTURE)
  ;; The result of CHAIN-FORMATION is a set (list) of chains, each a list
  ;; of noun phrases linked by movement.
  ;;
  ;; Compute all chains.
  (let ((chain-state (do-chain-formation phrase-structure)))
    ;; Unless there are either incomplete chains...
    (unless (or (chain-state-partial-chains chain-state)
                ;; Or noun phrases that have not been incorporated into chains...
                (chain-state-free-phrases chain-state))
      ;; then return the list of completed chains.
      (return-result (chain-state-completed-chains chain-state)))))

(defgenerator FREE-INDEXING ((list-of-chains CHAIN-FORMATION))
  ;; Return a list of chain sets, that looks like
  ;;
  ;; ((REFERENTIAL-INDEX-1 CHAIN-1-1 CHAIN-1-2 ...)
  ;;  (REFERENTIAL-INDEX-2 CHAIN-2-1 ...)
  ;;  )
  ;;
  ;; where each referential index is an integer and each chain is a list
  ;; of phrases.  The chains paired with a referential index all co-refer.
  ;;
  (when list-of-chains
    (return-result
     (loop for integer-set in (freely-index (length list-of-chains))
           for referential-index from 1
           collect (cons referential-index
```

```
                         (mapcar #'(lambda (i) (nth i list-of-chains))
                                 integer-set))))))
```

At any time during the execution of a generator the function `return-result` can be called to output a value. Depending on the particular search strategy, the execution of the generator function may or may not be temporarily halted while other aspects of the search process are performed. The body of a generator may be non-deterministic.

Filters are defined with the `deffilter` macro, which looks very much like `defgenerator`: `(deffilter name (&rest positions) &rest body)`. A filter rejects the current search path if and only if somewhere in its dynamic scope it executes the `reject` function. For instance,

```
(deffilter CONDITION-A ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING)
                        ANAPHOR?)
  ;; For each phrase in the phrase-structure tree TREE...
  (map-up-phrase-structure tree (phrase)
    ;; If that phrase is an anphoric noun phrase...
    (when (and (np?  phrase) (is-anaphor?  phrase))
      ;; Find the governing category for the phrase...
      (let ((gc (governing-category phrase)))
        ;; If there is a gov.  cat.  and no binder for the phrase within it...
        (unless (or (null gc) (find-binders phrase gc indices))
           Reject this parse.
           (reject))))))
```

```
(deffilter subjacency ((chains CHAIN-FORMATION))
  (dolist (chain chains) ;; For every chain...
    (mapl #'(lambda (chain-part)
              (let ((phrase1 (first chain-part))
                    (phrase2 (second chain-part)))
                ;; Phrase1 and Phrase2 are consecutive (movement slots) in the chain.
                (when (and phrase1 phrase2)
                  ;; If they are not subjactent, reject the chains.
                  (unless (subjacent phrase1 phrase2)
                    (reject)))))
          chain)))
```

Because it may be useful to signal a halt to the search process after a single solution is found, the special function `succeed`, when executed inside the dynamic scope of a generator or filter, halts the search process.

3.5 Specification of Dependencies and Search Strategies

3.5.1 Dependency Declarations

The definitions of generators and filters are not sufficient to determine the search dependencies. For instance, a filter may not explicitly need the value that a certain generator produces while it still relies on that generator's side-effects having taken place. Binding theory Condition A, presented in section 3.4 relies on previous side-effects to have set the anaphoric property of noun phrases, though no reference needs to be made in the argument list to the `ANAPHOR?` generator. Therefore the dependencies between various generators and filters must be explicitly declared. This is done by explicitly declaring the entire dependency graph. In the example, for instance,

```
(define-tree-positions
  '((top TOP-WORD)
    (generate top-word LEFT-WORD)
    (generate top-word RIGHT-WORD)
    (cross left-word right-word LEFT-AND-RIGHT)
    (generate left-and-right BOTTOM-WORD)))
```

Every module (generator or filter) must take its input arguments from a single named position in the dependency graph. Positions are named either by generators or, in the case of multiple dependencies, by explicitly creating a crossed position. All of this is done through the `define-tree-positions` function, which takes a list of clauses, each clause which defines a single position in the dependency graph. The possible position definition clauses are:

- `(top` *generator-name*`)`: Define the new position *generator-name* that represents the output of generator *generator-name*, which has no dependencies.
- `(generate` *parent-position generator-name*`)`: Define the new position *generator-name*. The generator *generator-name* is dependent on the position *parent-position* and its ancestors.
- `(cross` *parent-position-1 parent-position-2 new-position-name*`)`: Define the new position *new-position-name*, which combines all the information from *parent-position-1* and *parent-position-2*. No ordering is guaranteed, so there should be no interactions between the side-effects of the generators in the ancestors of the two parent positions.

It is often necessary to explicitly create one position which combines the results of all generators, so that there is a single element created that represents the solution to the global search. This is because there is no explicit mechanism in the language for outputting results. The most basic way to get at the results of a search is to execute on the cross-product of all generators a single filter that prints (or otherwise outputs) the value of each generator position, after the seach process is complete.

Notice that filters do not need to be declared anywhere here. This is because modules are not expected to be dependent on whether or not particular filters have been applied. If they are, then either the filter can be guised as a generator that either outputs its input or doesn't, or the filter dependency can simply be incorporated into the search strategy.

```
(defsearcher CROSSWORD-PUZZLE
  ((generate TOP-WORD
     (filter TOP-WORD-LENGTH top-word)
     (generate LEFT-WORD
       (filter LEFT-WORD-LENGTH left-word))
     (generate RIGHT-WORD
       (filter RIGHT-WORD-LENGTH right-word))
     (generate LEFT-AND-RIGHT)
     (generate BOTTOM-WORD
       (filter BOTTOM-WORD-LENGTH bottom-word)))))

(defsearcher CROSSWORD-PUZZLE
  ((generate TOP-WORD
     (filter TOP-WORD-LENGTH top-word)
     (generate LEFT-WORD
       (filter LEFT-WORD-LENGTH left-word)
       (generate RIGHT-WORD
         (filter RIGHT-WORD-LENGTH right-word)
         (generate LEFT-AND-RIGHT
           (generate BOTTOM-WORD
             (filter BOTTOM-WORD-LENGTH bottom-word))))))))
```

Figure 3.8: Two search programs for the crossword puzzle example.

3.5.2 Search Programs

Search programs are built with the `defsearcher` macro. The specification of the program orders the application of generators and filters, specifies concurrency, and declares whether one process has scope over another, or whether they are executed completely separately. As examples, the two search programs provided for the crossword puzzle are reprinted in figure 3.8.

The search programs must include a generation step for each tree position. But once generated, a position need not be regenerated. So, for instance, in the first program in figure 3.8 the lines

```
(generate RIGHT-WORD
  (filter RIGHT-WORD-LENGTH right-word))
(generate LEFT-AND-RIGHT)
```

first cause the `RIGHT-WORD` generator to run. For each of the resulting values, the `RIGHT-WORD-LENGTH` filter is applied. Then after this initial generation is complete, the `LEFT-AND-RIGHT` crossed position is generated. This process is dependent on the `RIGHT-WORD` position, but since that position has already been generated and stored away, it can be re-enumerated quickly to create the input arguments for the second generation process. In the second program this overhead is dispensed with:

```
(generate RIGHT-WORD
  (filter RIGHT-WORD-LENGTH right-word)
  (generate LEFT-AND-RIGHT
```

Here, after each element is generated by `RIGHT-WORD` the filter is executed, and then the `LEFT-AND-RIGHT` generator is applied. It only generates those crossed elements which include the one `RIGHT-WORD` currently being generated. This saves on some small overhead associated with re-enumerating `RIGHT-WORD` but introduces other potential inefficiencies, as is discussed in section 3.6.

There are a number of different clauses that can be used to construct search programs. The `defsearcher` macro functions as a compiler. It creates a function of no arguments that executes the search. All input and output must be made by individual generators and filters through special mechanisms, such as global variables. `defsearcher` takes the following form: `(defsearcher program-name body)`, where `body` is a list of clauses. Each clause is executed in sequence. The possible clause constructors are:

- `(generate` *position body*`)`: Enumerate any unbound parent positions, and execute the generator associated with *position*. As each result is returned, bind the position to that value and execute `body`.
- `(pgenerate` *position body*`)`: Similar to `generate`, except that as each value is generated, a separate process is created for it, and *body* is executed concurrently in each process. When all are complete, execution of the `pgenerate` halts. See section 3.8 for more information.
- `(filter` *filter-name parent-position*`)`: Enumerate any unbound parent positions, and apply the filter function. If it `rejects` the input, delete the element containing the input arguments and advance its enumerator. See section 3.6 for more information.
- `(cobegin` *body*`)`: Start separate processes for each of the clauses in *body*, and execute them concurrently. When all are complete, the `cobegin` halts.
- `(with` *position body* `)`: Like `generate`, except that it assumes *position* has already been generated and only needs to be enumerated.
- `(pwith` *position body* `)`: See `pgenerate` and `with`.

3.6 Failure Propagation

Propagation of failure is a classic problem in the search literature. In certain searches detecting the proper branch point to retreat to after a failure can greatly improve search efficiency, but in others the overhead associated with this computation turns out to be far more costly than the search itself. There are a few special considerations that need to be thought about for the search mechanisms used by this language.

First of all, recall that all value pairs produced by generators are encapsulated in elements and stored in sets. When a filter is executed, its function is applied to a particular element that represents its input arguments. If it *rejects* its input, then that element is deleted from its set, and a future enumeration of the set will not produce that element. Because a complete representation has been built of all the values produced during the search, deletion can be propagated beyond the local filter.

Imagine the following case related to the crossword puzzle example: suppose that the `TOP-WORD` generator produces the word *sequoias*. The next generator that will be executed is the `LEFT-WORD` generator, which will have to produce a word with q as its second letter. If the word list does not contain any 6 letter words with q in second position, then at the end of the `LEFT-WORD` generation process there will be no elements remaining in the new set (either none will have been produced, or those few produced will have been deleted by the `LEFT-WORD-FILTER`). This is an indication that the word *sequoias* is at fault for the failures, and can be deleted. If it is not, then the `RIGHT-WORD` generator will be applied needlessly to *sequoias*, producing values that can never be used.

In general, whenever a set that has been fully generated has its last element deleted, then the parent element to that set can also be deleted. It is not always trivial to ensure that a set has indeed been fully generated though. If we look at the following searcher

```
(generate A
  (generate B)
  (generate A-CROSS-B)
  (filter foo-filter A-CROSS-B)))
```

one might expect that the crossed position `A-CROSS-B` has been fully generated, and therefore that if the filter deletes the last element from one of the indexing sets, the parent to that indexing set can be deleted too. But that is not the case here. An indexing set that points back to an element of `A` will indeed have been fully enumerated. But an indexing set that points back to `B` (and contains one element for each $a \in$ `A`) will not be complete, because `A` is still being generated. For this reason it is not always advantageous to nest generation as much as possible.

3.6.1 Non-Local Exits

When a failure has been encountered, either because a filter has rejected its input arguments, or because a generator has produced no values, some exit must be performed back to get a new value for some antecedent tree position. In most cases it will be to the tree position that is being fed into the filter or generator. But if the failure has propagated beyond its starting point then the exit may be to a position further up in the dependency graph that must have its value changed.

A common strategy in searches is to backtrack to the closest (dynamic) superior branch point when a failure occurs. This is often because it is difficult or computationally expensive to determine the nearest branch point that is actually at fault in a failure. But since the search configuration is statically defined in this language it is very easy to compute the proper branch point to backtrack to. Thus in a depth first search defined with

```
(defsearcher CROSSWORD-PUZZLE
  ((generate TOP-WORD
```

```
    (filter TOP-WORD-LENGTH top-word)
    (generate LEFT-WORD
      (filter LEFT-WORD-LENGTH left-word)
      (generate RIGHT-WORD
        (filter RIGHT-WORD-LENGTH right-word)
        ...)))))
```

if the `RIGHT-WORD` generator fails to generate any values, then the search does not backtrack to the closest dynamic branch point (the `LEFT-WORD` generator) but instead the closest branch point that `RIGHT-WORD` is actually dependent on, `TOP-WORD`.

3.6.2 User-Declared Data Dependencies

It is common in natural language applications that there is shared substructure in the values produced by a generator. For instance, a phrase-structure generator may produce many parse trees that share a common phrase. In the ambiguous sentence *I saw a man with a telescope* the attachment of the prepositional phrase *with a telescope* varies but the word *I* is still unambiguously parsed as a pronoun. Imagine a filter being applied to the two possible phrase structure trees for this sentence. If it is concerned with the prepositional phrase then indeed it will do different work on the two values output by the generator, but if it, for instance, ruled out sentences with accusative pronouns in subject position, then it would redundantly apply to *I* twice. This overhead can essentially be eliminated with *memoization*, see section 3.7. But look at the sentence *Me saw a man with a telescope*; in this case the filter would rule out the sentence twice. Depending on the particular ordering of generators and filters it is quite conceivable that a significant amount of unnecessary processing will be done on the second value in between the time that the first value is ruled out by the filter and the time that the filter rules out the second value for the same reason.

For this reason a mechanism has been built that allows the user to define data dependencies among values, so that when a substructure in one value is rejected all other values with the same substructure are also deleted at the same time. It is used by defining a class for the substructure using a special macro that adds slots for dependency information and links to value pairs. Any object which is an instantiation of a class defined with this macro can be made dependent on any other such object, and any value pair dependent on any number of such objects. When any one of these objects is explicitly rejected (using an alternative form of `reject` then all elements with value pairs dependent on the object are deleted.

Although the definitions of dependent object classes is done with the `defmstructure` macro which will not be fully defined until section 3.7, the `:allow-dependencies t` section of the following definition declare that the `phrase` class allows dependency declarations.

```
(defmstructure (phrase :allow-dependencies t)
    ((category :initarg :category :accessor phrase-category :type symbol)
     (daughters :initarg :daughters :accessor phrase-daughters :type list)
     (parent ... )))
```

Inside any generator or filter one phrase can now be declared to be dependent on another with `(is-part-of` *substructure dependent-structure*):

```
(let ((phrase (make-instance 'phrase
                             :category category
                             :daughters daughters)))
  (dolist (daughter-phrase daughters)
    (is-part-of daughter-phrase phrase)))
```

Any value returned with `return-result` that is a dependency structure will be deleted if one of its substructures is rejected. And, whenever one dependency substructure is rejected (with a command like `(reject phrase)`), then all objects dependent on that structure are also rejected.

Unfortunately, this mechanism is not always as useful as it might seem, because often it is not a particular object that is at fault, but an object in conjunction with a variety of other side effecting operations or other objects. For instance, in *Me saw a man with a telescope* it is not *me* per se that is at fault, but *me* in subject position. In the sentence *Everyone but me saw a man with a telescope*, *me* could not be deleted. Unless this combination of an object and its sentential position are somehow encapsulated in a greater dependency object, this mechanism would not be useful. Section 4.2.1 presents some macros for performing this encapsulation that are used in an actual parser implementation.

3.7 Memoization

Natural language problems seem characterized by large numbers of local ambiguities, which are often expressed by building data structures that combine a small set of objects in a variety of different ways. Many of the same operations are performed on the same objects in each structure variation, and thus it is usually a significant optimization to *memoize* ([23], [13]), or store the results of an operation for future reuse. This is a variation of the technique used to allow generators to efficiently reproduce their results. The language provides a variety of macros that define memoized functions that maintain a table of the results of their application.

Although memoization of simple functions is easily added to COMMON LISP(see [24]), some extra care was taken to allow memoizable non-deterministic functions and efficient argument indexing. One of the common sources of inefficiency in memoized functions is the lookup of function arguments. At the start of a function call, the table pairing input arguments and results must be checked to see if the current arguments have been used before. For some classes of functions, such as those that take a single small integer argument, it is possible to efficiently store the table in the form of an array or a hash table. But for many functions that occur in natural language applications the input arguments are more likely to be complex data structures that are not easily indexed.

One solution is reserve a slot in a data structure for a given memoized function. When passed an argument structure, quickly checking a slot on that structure is sufficient to find a previously computed function value, if it exists. This efficient memoization technique is provided in the language through memoizing structures, or *mstructures*. The `defmstructure` macro allows one to declare classes of objects that can memoize. For example, in figure 3.9 the *phrase* class is defined to memoize 3 functions,

```
(defmstructure (phrase :allow-dependencies t)
    ((category :initarg :category :accessor phrase-category :type symbol)
     (daughters :initarg :daughters :accessor phrase-daughters :type list)
     ...)
  (ENUMERATE-PHRASE ANNOTATE-PHRASE-WITH-CASE DO-CHAIN-FORMATION))

(defndm do-chain-formation ((phrase) phrase)
  (case (length (phrase-daughters phrase))
    (0 ...)
    ...))

(defm! annotate-phrase-with-case ((phrase) phrase)
  ...)
```

Figure 3.9: An example illustrating the use of memoizing structures.

`enumerate-phrase`, `annotate-phrase-with-case` and `do-chain-formation`.

The four macros available for defining memoized functions that use mstructures for efficiency are `defm`, `defndm`, `defm!` and `defndm!`. `defndm` and `defndm!` are for non-deterministic functions, and `defm!` and `defndm!` record and reproduce side effects. Instead of the standard argument list found in a function definition, both an argument list and the name of the single argument that holds the mstructure must be provided. The similar macros `defmemo`, `defndmemo`, `defmemo!` and `defndmemo!` define memoizing functions that do not use mstructures, but store their results in ordinary hash tables.

Like the user-definable failure propagation mechanism, efficient memoization is not as useful as it might seem, because often a function's operations can not be easily specified just on the basis of its input arguments. The same structure may be passed to a function in two different contexts, with some side-effected slot containing a different value each time. The equality test on the arguments will not recognize the difference, and an (incorrect) memoized result will be returned at some point. Thus memoization must be used with some care.

An example of how care must be taken in memoization comes from binding theory. If we look at the definition of Condition A, from figure 2.1 it says that an anaphor must be bound in its governing category. Section 3.4 contains the `CONDITION-A` filter that implements this requirement. We might be tempted to memoize the computation of a phrase's governing category. But this won't work very well, because a single phrase can be shared between a large number of different tree structures, and its governing category may be different in each one. The memoization of the `c-commander` function (see Appendix B) gets around this problem by including in the function argument list the top level phrase of the tree structure. The c-commanders of a phrase are unique inside a given tree structure root.

3.8 Concurrency

In this search language, independent modules are made computationally independent. It is an obvious efficiency-motivated extension to make them computationally concurrent. If different modules can be executed at the same time, and even different search branches of a single module at the same time, then potentially the time cost of a complex search can be greatly reduced. Although the language has not actually been implemented on a concurrent computer system, the compiler can produce code with the necessary structure for concurrent execution, to the point that the code can be run in a simulated parallel environment such as LUCID COMMON LISP's processes.

Rather than go through the details of all the many additions and changes that must be made to the compiler to handle non-local exits, race-free modifications to the search state structure, process creation, and countless other minutiae, we will merely talk about some of the high level problems that must be dealt with when extending this language to handle concurrency in some form or other. Certainly the most important aspect that we will not touch upon is information passing between processes. The fine points of this are too dependent on the particulars of individual computer architectures and search programs.

3.8.1 Making COMMON LISP Functional

Concurrent computing is usually the domain of functional (non side-effecting) languages, and for a very good reason. If two different processes both effect a different change on the same memory location, then neither can be certain of its value. Given the existence of operations in this language like this one from example 3.5,

```
(set! (first simple-list) (either 1 2))
```

it should be obvious that it will be non-trivial to execute the two branches to this statement concurrently, or even to have two different modules operate on the two different results at the same time.

The solution to this problem is to turn COMMON LISP's side-effecting operations into non-destructive operators. Instead of a given writable location containing a readable value, it will contain a unique index. That index can be used to look back in the list of side-effects maintained for backtracking purposes (see section 3.3.1) to find the value the last side-effecting operation assigned to the location. For efficiency reasons not all side-effects use this mechanism, but only those explicitly declared to be potential conflicts under concurrent execution; it is possible that a sufficiently rich type and effect system (see [20], [27]) could deduce such conflicts automatically.

Looking at figure 3.10, to declare a memory location to be a potential conflict point for concurrency it must be a slot in an *mstructure*. In addition to the normal slot definition arguments, the `:nondestructive` keyword can be given. This declares that all read and write operations to this slot should be nondestructive. The argument given to the keyword is a list of all the generators that can potentially modify the slot, and this information is used to prune the search of the side-effects list. Once

```
(defmstructure (phrase :allow-dependencies t)
    ((category :initarg :category :accessor phrase-category :type symbol)
     (daughters :initarg :daughters :accessor phrase-daughters :type list)
     (parent :initform nil :accessor phrase-parent :NONDESTRUCTIVE (PHRASE-STRUCTURE))
     ...)
  (...))

(defgenerator phrase-structure (...)
  ...
  (set! (phrase-parent phrase) (either parent-A parent-B))
  ...)

(deffilter filter-parent (...)
  ...
  (if (predicate? (phrase-parent phrase))
      ...)
  ...)
```

Figure 3.10: An example illustrating how non-destructive side-effectors are declared and used.

this declaration has been made, the slot of a particular phrase will actually contain an index symbol such as `:parent-119` and a write operation such as the `set!` in the `phrase-structure` generator in figure 3.10 is translated to

```
(push (cons (phrase-parent phrase)      ;; The index.
            (either parent-A parent-B)) ;; The value.
      *effects*)
```

and a read like the `(phrase-parent phrase)` in the filter in figure 3.10 translated to something along the lines of

```
(cdr (assoc (phrase-parent phrase) ;; The index
            *effects*))
```

Here the overhead of a given read or write can be significant if many of them are performed and the list of effects grows large, but this mechanism does allow incompatible values to exist in different processes concurrently.

3.8.2 Specifying Concurrency

Section 3.5.2 enumerates the possible constructors for search programs, which include `pgenerate`, `pwith` and `cobegin`.

The `pgenerate` constructor executes a generator, then takes and runs its body on each generated element in parallel. The actual gain of this is hard to judge- it certainly would not reduce the time complexity of a search, since the generation process is still linear and whatever mechanism is used by the body to retrieve information from the ancestors of the generator is likely to be time dependent on the number of processes requesting information. But most likely some significant savings could result, especially in problems with little communication needs. `pwith` is very similar to `pgenerate`.

`cobegin` executes each statement of its body in parallel, and serves as a mechanism for running several modules concurrently. So, for instance, after a generator has been executed several different filters could be run over its results instead of just one at a time.

3.8.3 Overheads and Testing

It is important to realize that the high level concurrency constructs in this language are not practical for many applications, and have not been tested in any kind of representative environment. Although they can be used to the point of verifying correctness, the particular environment they have been tested in (interleaved processes in LUCID COMMON LISP) provides communication mechanisms transparently and otherwise allows many important aspects of concurrent computing to be glossed over. Furthermore, the high overhead associated with creating new processes swamps out the cost of the computation actually involved with the problem, and therefore renders empirical timing results meaningless.

Chapter 4

Tests and Results

The goal of this research was not to produce a particularly efficient, complete or psychologically plausible implementation of any particular human language competence or performance theory, but to create a convenient programming environment for experimenting with natural language problems and to test the viability of various efficiency-motivated search techniques. To this end, several relatively complex (though linguistically incomplete) principle-and-parameters based parsers were implemented in the language and tested under various search options.

4.1 Chapter Outline

This chapter introduces the parsers and their relevant computational properties, by describing each module one by one in section 4.2. To illustrate exactly how the modules work, a sample parse of a simple sentence is presented in section 4.2.2. Section 4.3 then describes the tests performed on the parsers and presents numerical results for a large number of different parser/parameter combinations. Finally, section 4.4 summarizes significant qualitative features found in these results.

4.2 The Test Parsers

The parsers are all search programs that seek to derive annotated parse trees (deep structures) for surface sentences, given as input a list of words. As with most GB-parsers (see [14], [12], [17]) they do this by starting with a covering context-free grammar for surface-structure, one that encompasses all legal transformations of a deep structure by the move-αprinciple. After deriving the surface structure of an entire sentence using standard CFG techniques, they reconstruct the movement sequence and also derive other ancillary information, such as antecedence relations between pronouns. This is not by any means all that a practical parser must do, and the parsers do not exhibit particularly broad linguistic coverage. For instance, the parsers do not derive important scoping information for quantifiers and are not able to derive the structure of questions about adjuncts. These and many other failings could be remedied relatively easily were the primary goal of this research to implement a more comprehensive language module.

There are only minor variations between the parsers, stemming from variations in the linguistic theory implemented. The changes were designed to test the effects of deterministic theories on the search. Because the majority of the modules in the parsers are identical, the parsers will be described together, module by module. Actual code for the parsers is provided in Appendix B, and section 4.2.2 provides a detailed sample execution of a parser.

See [28] for a more comprehensive introduction to the linguistic theories represented by these modules.

4.2.1 The Modules

The generators and filters that make up the test parsers are described in this section. For each one, a short description of the part of linguistic theory it implements is described, along with the intermodule dependencies and a description of any special language features used to implement the theory. First, though, some macros used by several modules and specific to linguistic structure are described.

Tree Structure Macros

Many of the modules map over phrase structure trees built by the first module of the parser. A variety of macros are defined to facilitate these mappings. Among these are `map-up-phrase-structure` and `map-up-phrase-structure-nd` which are used to apply a body of code to each phrase in a tree. For instance,

```
(deffilter CONDITION-A ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING)
                        ANAPHOR?)
  ;; For each phrase in the phrase-structure tree TREE...
  (map-up-phrase-structure tree (phrase)
    ;; If that phrase is an anphoric noun phrase...
    (when (and (np?  phrase) (is-anaphor?  phrase))
      ;; Find the governing category for the phrase...
      (let ((gc (governing-category phrase)))
        ;; If there is a gov.  cat.  and no binder for the phrase within it...
        (unless (or (null gc) (find-binders phrase gc indices))
           Reject this parse.
           (reject))))))
```

Another useful set of macros deal with the problem of memoization over combinations of phrases and side-effects. Imagine trying to memoize the Case Filter on phrases. Since case theory depends on whether phrases are anaphoric or not, any case-checking function will need the anaphoric property of a phrase to be an argument. But the anaphoric nature of empty categories is set destructively by a non-deterministic module, and therefore can not easily be referenced as an argument. The solution is to build new structures that encapsulate the anaphoric side-effect with the phrase object, and memoize on that new object. Obviously there is an overhead associated with this, but the new object can also be used for user-declared failures (see section 3.6.2), and the process can be built into macros that hide the details of the operations. These same macros can be used for other theories. The new structures built are called *virtual trees*. The following code illustrates the use of virtual trees to provide a single rejectable object that represents phrases and their anaphoric setting.

```
(defgenerator ANAPHOR? ((tree OPERATOR-ASSIGNMENT))
  (return-result (annotate-empty-categories-with-anaphor-feature tree)))

(defndm! annotate-empty-categories-with-anaphor-feature ((phrase) phrase)
 (virtual-map-up-phrase-structure
     (phrase annotate-empty-categories-with-anaphor-feature)
   (when (np? phrase)
     (if (empty? phrase)
         (unless (phrase-anaphor? phrase)
           (set!-local (phrase-anaphor? phrase) (either '+ '-)))
         (if (word? phrase)
           (set!-local (phrase-anaphor? phrase) (feature-value phrase 'anaphor)))))))

(deffilter EMPTY-CASE-FILTER ((virtual-tree ANAPHOR?) CASE-ASSIGNMENT)
  ;; Filter that says +A ecs must not get case, -As must (if in A positions).
  (virtual-phrase-map virtual-tree (phrase :category category)
   (when (and (eq category 'n2) (empty? phrase) (a-position? phrase))
     (let ((ana (phrase-anaphor? phrase))
           (case (phrase-assigned-case phrase)))
       (when (or (and (eq '+ ana) case)
                 (and (eq '- ana) (not case)))
         ;; It is not the empty category itself that is responsible
         ;; for the failure, but whatever domain that should contain
         ;; a case assigner.  This is usually the parent phrase of
         ;; the empty category, though if ECM were handled we would
         ;; have to be more careful.  Reject up one level of the tree
         ;; from PHRASE.
         (virtual-reject 1))))))
```

Now we present the modules that make up the parsers.

<u>`PHRASE-STRUCTURE`</u> *generator*

The `phrase-structure` generator gets a sequence of words from a global variable (it has no formal dependencies) and uses a tabular CFG parser and a small lexicon and grammar to non-deterministically produce phrase structure trees of that sentence. The grammar the parsers use is a subset of a covering grammar for the surface-structure of English, and includes empty categories. The parser inserts empty categories without any particular movement clues (this contrasts with parsers that only insert empty categories, for example, to the right of question words like *who* that indicate probable movement). This leads to overproduction, so that a sentence such as *John likes to bicycle* generates 24 different parse trees, including

```
(C2
  (C1 (C0)
      (I2 (N2 JOHN)
          (I1 (I0)
              (V2
                (V1 (V1 (V0 LIKES) (N2))
                    (P2
                      (P1 (P0 TO)
                          (C2
                            (C1 (C0)
                                (I2 (N2)
                                    (I1 (I0)
                                        (V2 (V1 (V0 BICYCLE)
                                                (N2)))))))))))))))

(C2
  (C1 (C0)
      (I2 (N2 JOHN)
          (I1 (I0)
              (V2
                (V1 (V0 LIKES)
                    (C2
                      (C1 (C0)
                          (I2 (N2)
                              (I1 (I0 TO)
                                  (V2 (V1 (V0 BICYCLE))))))))))))
```

The second tree is correct for the sentence, but the first illustrates some of the problems that result from not using enough information in the initial parsing process, such as incorrect argument structure for verbs and prepositions and a surfeit of empty categories. A more psychologically plausible parser would utilize this information early in the construction of sentential structure (see [11], [14]), in some sort of interleaved approach.

The `phrase-structure` generator outputs any number of tree structures, each in the form of a root phrase. Each phrase may have any number of daughter phrases, and some phrases may be shared between trees. Each phrase is a structure that also contains slots for information produced by other modules.

Language Facilities

The `phrase-structure` module uses the user declared data dependency feature to make each phrase a fundamental part of the phrases that contain it. This way, if any filter can rule out a particular sub-phrase many different parse trees can be eliminated at once. Most of the memoization involved in context-free parsing is handled explicitly by the chart parser, but one interesting function is

```
(defm!  set-phrase-parents ((phrase) phrase)
  (mapc #'(lambda (daughter)
           (set-phrase-parents daughter)
           (set!-local (phrase-parent daughter) phrase))
```

Since phrases are shared, it is not possible for them to have a single slot always pointing to their parents. The `set-phrase-parents` function is applied to the root node of parse trees as they are enumerated. It destructively sets the parent slots of all phrases within the current parse tree. The side-effects are stored along with the root node. The function can be memoized over each phrase, which speeds up the initial process of producing the value pair. The memoization is performed using mstructures (see section 3.7).

<u>`THETA-ROLE-ASSIGNMENT`</u> *generator*

For the purposes here, theta (short for *thematic*) roles can be thought of as the deep structure argument relations assigned by verbs to their subjects and objects. Different verbs assign different thematic roles, and therefore can take different combinations of arguments. For instance, one form of *bicycle* assigns an *agent* role to its subject (*John bicycled*) and another assigns a role to an object also (*John bicycled the Schwinn*), but no form assigns the clausal role that a verb like *think* does. This generator deterministically assigns thematic roles from verbs, prepositions and other words to phrases in particular structural relations with them, such as subjects. It fails when the assigning word is in the wrong sort of structural relation to assign the role (such as the verb *bicycle* with a clausal argument). The assignment is done by storing a theta-role symbol directly on the phrase that is the recipient of the role.

Structural positions in which thematic roles *can* be assigned (not must) are called *argument positions*, or *A-positions*. These include the classical subject position, and the complement position of verbs and prepositions. Non-argument positions are called *Abar-positions*.

Since theta-role assignment is on the basis of an inherent lexical property of a word and a structural relation, the generator need only be dependent on the `phrase-structure` generator. The `theta-criterion` filter works to filter out structures based on theta-role assignment and movement relations.

Language Facilities

Theta assignment is on the phrasal level under particular structural relationships, and can therefore be memoized on a phrase. This means that assignment applies only once to a phrase, regardless of how many times that phrase might appear in different parses. The only restriction one must be wary of is that it must be memoized on a phrase at least as high as both the assigner and assignee. The mstructures described in section 3.7 are used so that argument lookup during memoization is particularly efficient.

<u>`CASE-ASSIGNMENT`</u> *generator*

Case assignment, like theta-role assignment, is on the basis of structural relations between a word with certain properties and a phrase. Just as in theta-role assignment, a symbol indicating the type of case assigned is deterministically stored on the phrase that receives the case.

Language Facilities

Case assignment is very similar to theta-role assignment, and many of the same optimizations are used. But although case-assignment in these parsers is deterministic, the code allows for a non-deterministic theory. This necessitates a few changes, such as the use of virtual trees so that case is encapsulated with phrases.

```
(defgenerator CASE-ASSIGNMENT ((tree PHRASE-STRUCTURE))
  (if (eq (phrase-category tree) 'n2) ;; Top-level NP
      (assign-case tree 'NOM))
  (return-result (annotate-phrase-with-case tree)))

(defm! annotate-phrase-with-case ((phrase) phrase)
  (virtual-map-up-phrase-structure (phrase annotate-phrase-with-case)
    (when (eq phrase (core-phrase phrase))
      (cond ((p1? phrase)
             (when (and (complement phrase) (np? (complement phrase)))
               (assign-case (complement phrase) 'acc)))
            ...
```

<u>`LEXICAL-CASE-FILTER`</u> *filter*

The Case Filter, described in section 2.2.2, filters out some noun phrases that have not been assigned case. These noun phrases can be divided into two categories, lexically realized and empty categories. Empty categories do not necessarily have to receive case; this depends on their particular type. Lexically realized noun phrases must always receive case. The Case Filter has been divided into two parts, so that it can be applied early in the search process to lexically realized noun phrases and later to empty categories whose typology is not determined at the beginning of the search.

The `lexical-case-filter` is naturally dependent on the `case-assignment` generator.

Language Facilities

The Case Filter could be memoized over the virtual tree structure produced by `case-assignment`, though it is not. It rejects the specific data structure representing a combination of case and phrase structure so that all parse trees combining the combination are eliminated.

<u>`OPERATOR-ASSIGNMENT`</u> *generator*

Certain empty categories do not easily fit into the standard typology of being either anaphors, pronominals, neither or both (see section 2.3.5). These include *operators*, which only appear in certain unambiguous structural positions. The `operator-assignment` generator deterministically sets the type of empty categories that can be unambiguously determined by their structural location; it is dependent only on the structural forms generated by `phrase-structure`.

`ANAPHOR?` *generator*

As discussed in section 2.3.5, empty categories are usually assumed to have features associated with them. In particular, most empty categories fall into the typology of being either ±anaphoric and ±pronominal. These features can not be determined from the surface realization the empty phrase (there is no surface realization), but various theories differ as to how the features are set.

One possibility is to non-deterministically guess the value of each of these features. Call this the *free determination* of empty categories. Another is to ([8]) determine the particular functional relationships the empty category enters into and work backwards, specifying that, for instance, it can not be a pronominal if it is locally bound. This is called the *functional determination* of empty categories. A third possibility ([10] is to use structural information about the location of the empty category in the phrase-structure tree, combined with lexical information such as case, to determine the type of the empty category. This is called *structural determination* of empty categories.

It is not entirely clear exactly what empirical differences arise from the three different approaches. [14] cites an example sentence that structural determination might predict as grammatical when in fact it isn't, a mistake that the functional determination theory of [8] doesn't make, but it is not clear that there aren't other mechanisms that could explain the ungrammaticality. Free determination, which would examine all possibilities, would certainly make the same (possibly incorrect) prediction about Fong's sentence. It seems that free determination overgenerates in several cases, though that may reflect the lack of a different constraint.

All three of these different mechanisms have been implemented, and all have different dependencies. The free determination hypothesis merely guesses whether an empty category is an anaphor and thus needs not rely on any other information, so under this hypothesis `anaphor?` is dependent only on `operator-assignment` (and `operator-assignment`'s parent, `phrase-structure`).

Functional determination requires knowledge of movement and binding relationships. These can be computed before the type of an empty category is known, but only at a significant computational cost that arises from doing non-deterministic search on many structures that would be ruled out by filters that rely on empty category typology. The solution to this problem taken here is to implement functional determination as a filter that applies only after movement and binding relations have been computed on empty categories determined by free determination.

Structural determination relies only on structural and lexical information to determine empty category types. In Correa's system, `anaphor?` is dependent on both `operator-assignment` and `case-assignment` while `pronominal?` is dependent only on `operator-assignment`.

Language Facilities

For free determination and functional determination, the `anaphor?` generator sets the anaphoric and pronominal properties of empty categories without checking any structural relationships. This process can be memoized on the empty category itself. For structural determination this is not possible, since some structural context needs to be captured. Virtual trees are used in either case to capture a combination of anaphoric nature and phrase, which is used in the `empty-case-filter` for efficient error propagation.

```
(defgenerator ANAPHOR? ((tree OPERATOR-ASSIGNMENT))
  (return-result (annotate-empty-categories-with-anaphor-feature tree)))

(defndm! annotate-empty-categories-with-anaphor-feature ((phrase) phrase)
  (virtual-map-up-phrase-structure (phrase annotate-empty-categories-with-anaphor-feature)
    (when (np? phrase)
      (if (empty? phrase)
          (unless (phrase-anaphor? phrase)
            (set!-local (phrase-anaphor? phrase) (either '+ '-)))
          ...
```

`PRONOMINAL?` *generator*

See `anaphor?` above.

`EMPTY-CASE-FILTER` *filter*

As discussed in the section on the `lexical-case-filter`, the case filter is divided into two parts. The `empty-case-filter` ensures that empty categories which are anaphoric do not received case, and that non-anaphoric empty categories in certain positions do receive case. It is therefore dependent on both the `case-assignment` generator that assigns case and the `anaphor?` generator that determines the anaphoric nature of empty categories.

Language Facilities

The `empty-case-filter` is presented in section 4.2.1. It maps over the virtual trees created by `anaphor?`, which allows it to delete several parses at once if functional determination is in effect. Structural determination does not use memoization to share anaphoric structure, so for structural determination this filter does not gain in efficiency by using virtual trees.

`CHAIN-FORMATION` *generator*

Chain formation is the process by which movement histories (chains) between D-structure and S-structure are derived. Empty categories inserted by the `phrase-structure` generator into its parse trees are linked with phrases that have moved to positions different from their D-structure locations. The output is a list of chains of phrases, where each chain is a list of phrases that represents a set of phrases intimately linked by movement. These phrases are assumed to corefer, and bear the same referential index with respect to binding theory.

Two different theories of chain-formation have been implemented, the first based loosely on the non-deterministic chain-formation mechanism in [14] and the second on the deterministic algorithm in [10]. Both build chains phrase by phrase, working up from the bottom of the phrase structure trees and combining chains at each node. Fong's algorithm essentially builds all possible chains of noun phrases in which one phrase c-commands the next in a chain, where the definition of c-command is (as in figure 2.1): *For α, β nodes in a tree, α c-commands β iff every branching node dominating α dominates β and neither α nor β dominates the other.* Correa's algorithm deterministically builds chains by compiling in filters that use information about the exact position each phrase is in. The result is a very efficient algorithm, with more dependencies (though only on deterministic generators) but poorer empirical judgements: Correa's chain-formation algorithm can not explain parasitic gap sentences. Parasitic gap sentences involve movement that creates more than one empty position, such as *which book did you file e without reading e?*. Without any mechanism for merging chains or binding an empty category that is not part of a chain, at some point his algorithm will fail on this sentence. Despite the current deficiency, it is quite possible that a modified version of his chain formation algorithm could be made to work with parasitic gap sentences.

The non-deterministic chain-formation algorithm is dependent only on the `phrase-structure` generator. Correa's deterministic algorithm is dependent on `phrase-structure`, `theta-role-assignment`, `anaphor?` and `pronominal?`.

Language Facilities

Both chain formation algorithms work bottom-up, creating subchains for every daughter phrase before using that information to create the chains for the parent. The non-deterministic implementation of chain formation does not use information beyond the level of the parent phrase, and thus the algorithm can be memoized by phrase, saving work when multiple parses share phrase structure. But Correa's deterministic algorithm makes reference to theta-roles and other information from beyond the immediate locality, so memoization is not used.

`SUBJACENCY` *filter*

Subjacency is a condition on movement chains that ensures successive stages of movement are not separated by more than a small distance, defined in terms of the number of intervening nodes in the phrase-structure tree. This notion of locality is captured in many different ways in different linguistic theories, but here is implemented in a fairly traditional manner. The `subjacency` filter is dependent only on the `chain-formation` generator and of course, the phrase structure trees themselves.

`THETA-CRITERION` *filter*

One of the most fundamental (and long-lived) principles in the Government-Binding theories is the theta (thematic) criterion, which states that every noun phrase must receive one and only one thematic role.

Thematic roles are semantic roles such as agent and patient assigned by verbs and prepositions. The notion has been extended to state that every movement chain must receive one and only one role.

The `theta-criterion` filter is dependent on both the `theta-assignment` and `chain-formation` generators, and checks that each movement chain produced by `chain-formation` contains exactly one noun phrase that has received a theta-role from `theta-assignment`.

<u>`FREE-INDEXING`</u> *generator*

In figure 2.1 a version of binding theory is presented that states: *assign numerical indices freely to all noun phrases, subject to Conditions A, B and C.* The `free-indexing` generator implements the *assign numerical indices* portion of binding theory, by non-deterministically partitioning the chains produced by the `chain-formation` algorithm into different sets (each set corresponds to a different numerical index).

Language Facilities

The free indexing procedure really only needs to know the number of different chains, not the exact nature of each of them, to compute the possible partitions. And there is a simple compositional algorithm for computing the possible partitions for n chains given the possible partitions for $n-1$ chains. This algorithm can be memoized on n for efficiency.

```
(defndmemo freely-index ((n) () eql)
  ;; Nondeterministically return a partition on the integers 0 through
  ;; N-1.  Each partition is a list of integers.  For example, the
  ;; five values returned by (freely-index 3) are
  ;;
  ;;    ((2) (1) (0))
  ;;    ((2 1) (0))
  ;;    ((2 0) (1))
  ;;    ((2) (1 0))
  ;;    ((2 1 0))
  ;;
  (unless (zerop n)
    (let ((indexing (freely-index (1- n))))
      (either
       ;; Put integer into its own partition.
       `((,(1- n)) ,@indexing)
       ;; Put integer into some existing partition.
       (let ((index-set-to-merge-with (a-member-of indexing)))
         `((,(1- n) ,@index-set-to-merge-with)
           ,@(remove index-set-to-merge-with indexing)))))))
```

CONDITION-C-REXP *filter*

Binding theory condition C (section 2.2.1) states that R-expressions (names, for our purposes) must not be bound. This translates into a requirement that no phrase c-commands an R-expression with the same numerical index (see `free-indexing`). The theory treats certain types of empty categories as R-expressions, and therefore, as with the Case Filter, condition C is divided into two parts. The `condition-c-rexp` filter applies only to overt noun phrases and not empty categories, and can thus be applied before the typology of empty categories is determined.

`condition-c-rexp` is dependent on the indices assigned by `free-indexing` and, of course, the structural relations inherent in `phrase-structure`.

CONDITION-C-VAR *filter*

The `condition-c-var` filter acts as `condition-c-rexp` except that it applies only to empty categories that are non-anaphoric and non-pronominal. As such, it applies only after `anaphor?`, `pronominal?`, and `free-indexing` have been generated.

COINDEX-OPERATOR *filter*

Operators (see the `operator-assignment` generator) are a particular type of empty category used to represent scoping information. In this parser, relative clause constructions such as *the man that John saw* are assumed to have the following structure

[[NP the man][CP [OP *e*] [that John saw *e*]]]

The empty object of *saw* bears the same reference as *the man*, and has somehow moved from its base position. This is captured by the object moving (being part of the same chain) to the position of the operator [OP *e*]. The operator is then coindexed with the noun phrase *the man.* The `coindex-operator` filter checks that whatever index is assigned by the `free-indexing` generator to the noun phrase *the man* has also been assigned to the operator, and hence (since all members of a chain bear the same reference) to the object of *saw.*

I-WITHIN-I *filter*

The `i-within-i` filter captures a fairly basic condition that one phrase not bear the same referential index as another phrase it contains. This rules out phrases like

[his$_i$ friend]$_i$

where *his* and *friend* are coindexed. The filter is dependent on `free-indexing`, which labels chains with indices.

`CONDITION-A` *filter*

Condition A of binding theory (section 2.2.1) states that anaphors (words like *himself* and *themselves*) must have an antecedent within a local domain. It applies to all noun phrases that have anaphoric properties, including some empty categories. For each noun phrase `condition-a` computes the local domain, then checks that there is another noun phrase within the domain that has the same referential index.

The `condition-a` filter can only be applied after both the anaphoric property and referential index of a noun phrase has been computed, and therefore it is dependent on `free-indexing` and `anaphor?`.

`CONDITION-B` *filter*

The `condition-b` filter parallels `condition-a`, except that it verifies pronominal noun phrases do not have local binders, and is dependent on the `pronominal?` generator rather than `anaphor?`.

`LICENSE-CHAINS` *filter*

There are a number of conditions on movement histories (chains) that are not directly enforced by the `chain-formation` generator, in part because they derive from different aspects of the linguistic theory and in part because they depend on information not available at the time chains are produced. The `license-chains` filter checks that chains produced by `chain-formation` indeed satisfy other criteria. Among the conditions enforced:

- Each non-trivial chain includes at least one trace (`-pronominal` empty category).
- Operators must bind variables (`-pronominal`, `-anaphor`) empty categories.
- Operators must head (complete) a chain they are included in.
- English does not permit `+pronominal`, `-anaphor` empty categories.
- The empty category resulting from the movement of a question word must be licensed by an operator in the same movement chain.
- Only lexical items, operators, variables, and `+pronominal` empty categories may head chains.

Obviously, the eclectic nature of this filter necessitates dependencies on a variety of generators: `anaphor?`, `pronominal?` and `chain-formation`.

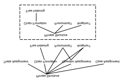


Figure 4.1: The dependencies between generators in the parser used for the example. The box contains the dependency graph among non-deterministic generators.

<u>`FUNCTIONAL-DETERMINATION`</u> *filter*

Functional determination is described above in the section on the `anaphor?` generator. It is a theory of how the typology of empty categories is determined. In particular, it states that the anaphoric and pronominal nature of an empty category is determined by the functional relationships the empty category enters in to. This is implemented by non-deterministically guessing the nature of the empty category and later (in the `functional-determination` filter) checking that the values were guessed correctly.

The filter uses information on structural positions, binders, and chains. It is dependent on the generators `chain-formation`, `anaphor?`, `pronominal?` and `free-indexing` in addition to basic `phrase-structure`.

4.2.2 An Example Parse

The general function of the above modules should be clearer after seeing an example of their execution. This section presents a sample parse of the sentence *Who did John say that he saw?*, by a parser using functional determination and the non-deterministic chain-formation algorithm. The dependency structure of the generators in the parser is shown in figure 4.1. The search strategy separates the execution of conceptually independent modules.

The first module to be executed is the `phrase-structure` generator, which non-deterministically parses the input sentence into phrase structure trees. It looks each word up in its small dictionary to determine its part of speech and special features (for instance, that *he* is a pronoun), then uses a simple context-free chart parser with a small grammar to produce the following four phrase structure trees:

```
(C2 (N2 WHO)
    (C1 (I0 DID)
        (I2 (N2 JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2 (N2)
                          (C1 (C0 THAT)
                              (I2 (N2 HE)
                                  (I1 (I0)
                                      (V2 (V1 (V0 SAW)))))))))))))
(C2 (N2 WHO)
    (C1 (I0 DID)
        (I2 (N2 JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2 (N2)
                          (C1 (C0 THAT)
                              (I2 (N2 HE)
                                  (I1 (I0)
                                      (V2 (V1 (V0 SAW) (N2)))))))))))))
(C2 (N2 WHO)
    (C1 (I0 DID)
        (I2 (N2 JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2
                        (C1 (C0 THAT)
                            (I2 (N2 HE)
                                (I1 (I0)
                                    (V2 (V1 (V0 SAW)))))))))))))
(C2 (N2 WHO)
    (C1 (I0 DID)
        (I2 (N2 JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2
                        (C1 (C0 THAT)
                            (I2 (N2 HE)
                                (I1 (I0)
                                    (V2 (V1 (V0 SAW) (N2)))))))))))))
```

The variation between the 4 trees is slight. The only differences are whether or not there is an empty object to the verb *saw* and whether or not there is an empty category in the specifier position of the embedded clause *that he saw.*

Then `theta-role-assignment` applies. Since the dictionary includes the information that *saw* assigns a role to an object noun phrase or clause, the two phrase structures with no phrase in complement position of *saw* fail and are discarded. For the remaining two structures, *John* receives a theta-role from *say*, *he* and the empty category in object position from *saw.*

`Operator-assignment` applies to the two phrase structures. Since neither have an empty category in an operator position, the generator outputs two unchanged structures.

`Case-assignment` assigns case from the two finite verbs in the sentence, nominative case to *John* and *he* and accusative case to the empty category in object position. At the end of this process, the two structures look something like:

```
(C2 (N2 WHO)
    (C1 (I0 DID)
        (I2 (N2 SUBJECT NOM JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2 (C1 (C0 THAT)
                              (I2 (N2 SUBJECT NOM HE)
                                  (I1 (I0)
                                      (V2 (V1 (V0 SAW)
                                              (N2 OBJECT ACC))))))))))))
(C2 (N2 WHO)
    (C1 (I0 DID)
        (I2 (N2 SUBJECT NOM JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2 (N2)
                          (C1 (C0 THAT)
                              (I2 (N2 SUBJECT NOM HE)
                                  (I1 (I0)
                                      (V2 (V1 (V0 SAW)
                                              (N2 OBJECT ACC))))))))))))
```

Here the actual theta-roles *see* and *say* would have assigned have for convenience been replaced with `SUBJECT` and `OBJECT`. The `lexical-case-filter` then checks that the lexically realized noun phrases in argument positions (*who* is not in an argument position) have been assigned case. In both structures *John* and *he* have received case, so both structures pass.

The `anaphor?` generator is now executed, which non-deterministically assigns the values `+` and `-` to the anaphoric property of empty categories not in the position of specifier of embedded clauses. In other words, the values are assigned to the empty category in object position of *saw*, but not to the empty category immediately preceding *that.* Since this process applies to both structures, the two input phrase structures produce four different phrase structures annotated with anaphoric features. Once the anaphoric properties have been determined, the `empty-case-filter` can be applied to empty noun phrases in argument positions (the empty category that is the complement of *saw*). It requires that `+anaphor` empty categories be caseless. Since two of the annotated phrase structures label the complement of *saw* as being anaphoric, and since *saw* has assigned it case, these two paths are filtered out and the empty category complement of *saw* is unambiguously non-anaphoric. In fact, the failure propagation mechanism actually rules out both after the first one has been analyzed, so the filter is actually only applied once.

Now the non-deterministic `chain-formation` generator is applied to the two phrase structures. The algorithm has a high branching factor, and 14 different chain possibilities result. 11 of these movement chains are for the structure with two empty categories, 3 for the structure with one empty category. Looking just at the 3,

```
((<Phrase N2: WHO>) (<Phrase N2: JOHN> <Phrase N2:>) (<Phrase N2: HE>))
((<Phrase N2: WHO>) (<Phrase N2: JOHN>) (<Phrase N2: HE> <Phrase N2:>))
((<Phrase N2: WHO>) (<Phrase N2: JOHN>) (<Phrase N2: HE>) (<Phrase N2:>))
```

Of the 3 movement chain sets produced, the first indicates that *John* moved from the position of the empty category, the second has *he* moving from that position, and in the third all four noun phrases are parts of separate chains (no movement took place). None of these or the other 11 chains are ruled out by the `subjacency` filter that is applied to them. But a much stronger requirement is then applied, the `theta-criterion`. This requires that every chain receive exactly one theta-role. In the three chains listed, the chain including *who* never receives a theta-role and thus all are filtered out. In fact, only one chain set of 14 survives this filter:

```
((<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>)
 (<Phrase N2: JOHN>)
 (<Phrase N2: HE>))
```

Here *John* and *he* are in their own chains, and the chain containing *who* and two empty categories receives its theta-role by virtue of the empty category in complement position of *saw*. Since all of the 3 chain sets for the one phrase structure are filtered, the phrase structure is also deleted.

Now the `free-indexing` generator applies, to the one remaining chain set produced by `chain-formation`. It partitions the chains into different sets with the same referential index. There are 3 chains in the one chain set that survived the theta criterion, and the five partitions `free-indexing` produces look like:

```
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>)] [(<Phrase N2: JOHN>)] [(<Phrase N2: HE>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: JOHN>)] [(<Phrase N2: HE>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>)] [(<Phrase N2: JOHN>) (<Phrase N2: HE>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: HE>)] [(<Phrase N2: JOHN>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: JOHN>) (<Phrase N2: HE>)]
```

In the first partition the reference of *he*, *John* and *who* is disjoint, and in the last partition all noun phrases are coreferential. Now four different filters apply to these five partitions without whittling down their numbers. `condition-c-rexp` verifies that *John* is not bound by a noun phrase in an argument position but the only potential binder is *who* and that is not in an argument position. `coindex-operator` has no effect because there are no operators. `i-within-i` does not apply to any case here, and `condition-a` applies only to anaphoric elements, which are nonexistent in this structure.

The `pronominal?` generator is now applied to the phrase structures. Note that it is not applied to the five values produced by `free-indexing` but only to the one remaining valid phrase structure. It non-deterministically assigns the values `+` and `-` to the pronominal property of the empty category that is the complement of *saw*, producing two output values. Now the `condition-b` filter is applied. It is dependent on both `free-indexing` and `pronominal?`. Since the one phrase structure has 5 indexings and 2 different pronominal values, the result is a cross product of ten different inputs to the filter. Of these, binding theory condition B rules out the two in which the empty category is a pronoun and there is local binder for that pronoun (*he*, in fact). These correspond to the following two `free-indexing` values:

```
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: HE>)] [(<Phrase N2: JOHN>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: JOHN>) (<Phrase N2: HE>)]
```

The pronoun *he* escapes condition B because there are no potential binders within its locality domain, the clause *that he saw*.

The `license-chains` filter is applied and is dependent on chains and the anaphoric and pronominal properties of empty categories. The one remaining phrase structure has only one chain, only one possible assignment of anaphoric features, and two possible assignments of pronominality to the empty category in complement position of *saw*. `license-chains` rules out the `+pronominal` possibility on the grounds that English does not permit empty categories that are pronominal but not anaphoric. At this stage the single phrase structure has unambiguous empty category typology, one possible chain, and still 5 possible indexings. The `condition-c-var` rules out

```
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: JOHN>)] [(<Phrase N2: HE>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: HE>)] [(<Phrase N2: JOHN>)]
[(<Phrase N2: WHO> <Phrase N2:> <Phrase N2:>) (<Phrase N2: JOHN>) (<Phrase N2: HE>)]
```

because the non-anaphoric, non-pronominal empty category would be illegally bound by either *John* or *he*.

Finally, the *functional-determination* filter is applied for the one phrase structure to the last two possible indexings, the only possible chain, and the unambiguous empty category typology. It finds that the empty categories have the proper anaphoric and pronominal values for their usage, and filters neither structure. The parser's output is shown in figure 4.2

The only difference between these two structures is whether *John* and *he* corefer or are disjoint in reference. The first number in each noun phrase (`N2`) is a referential index, and the second number in parenthesis reflects the chain a noun phrase is part of. Figure 4.3 summarizes the work each module performs during the parse.

```
(C2 (N2 1 (0) WHO)
    (C1 (I0 DID)
        (I2 (N2 2 (1) SUBJECT NOM JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2 (N2 1 (0) -A -P)
                          (C1 (C0 THAT)
                              (I2 (N2 3 (2) SUBJECT NOM -A +P HE)
                                  (I1 (I0)
                                      (V2
                                        (V1 (V0 SAW)
                                            (N2 1
                                                (0)
                                                OBJECT
                                                ACC
                                                -A
                                                -P)))))))))))))
(C2 (N2 1 (0) WHO)
    (C1 (I0 DID)
        (I2 (N2 2 (1) SUBJECT NOM JOHN)
            (I1 (I0)
                (V2
                  (V1 (V0 SAY)
                      (C2 (N2 1 (0) -A -P)
                          (C1 (C0 THAT)
                              (I2 (N2 2 (2) SUBJECT NOM -A +P HE)
                                  (I1 (I0)
                                      (V2
                                        (V1 (V0 SAW)
                                            (N2 1
                                                (0)
                                                OBJECT
                                                ACC
                                                -A
                                                -P)))))))))))))
```

Figure 4.2: The two parse trees finally output by the parser for the sentence *Who did John say that he saw?*

Module	*# of inputs*	*# of outputs*	*max outputs/input*
`FUNCTIONAL-DETERMINATION`	2	2	1
`CONDITION-C-VAR`	5	2	1
`LICENSE-CHAINS`	2	1	1
`CONDITION-B`	10	8	1
`PRONOMINAL?`	1	2	2
`CONDITION-A`	5	5	1
`I-WITHIN-I`	5	5	1
`COINDEX-OPERATOR`	5	5	1
`CONDITION-C-REXP`	5	5	1
`FREE-INDEXING`	1	5	5
`THETA-CRITERION`	14	1	1
`SUBJACENCY`	14	14	1
`CHAIN-FORMATION`	2	14	11
`EMPTY-CASE-FILTER`	3	2	1
`ANAPHOR?`	2	4	2
`LEXICAL-CASE-FILTER`	2	2	1
`CASE-ASSIGNMENT`	2	2	1
`OPERATOR-ASSIGNMENT`	2	2	1
`THETA-ROLE-ASSIGNMENT`	4	2	1
`PHRASE-STRUCTURE`	1	4	4

Figure 4.3: For each module, the number of times it was applied to an input argument is printed in the first column; the second column holds the total number of values the module output (or in the case of filters, accepted); the third column holds the maximum number of outputs for any single input. Were all branches of a module executed concurrently with no overhead, the third column would reflect the amount of time spent in execution.

4.3 Test Results

Again, the goal of this research was not to create a particularly efficient implementation of any given linguistic theory. So the metric by which the programming language and parser implementation will be judged is neither how much time and memory was expended during the search, nor how many grammatical constructs were correctly parsed. What we *are* interested in is an abstract notion of how much less (or more) computation a linguistic theory of parsing entails if implemented using the sorts of search techniques presented in chapter 3 instead of more standard depth first search techniques, and whether the added overhead of the techniques outweighs any small improvement.

To gather empirical data on performance variations, the different parsers described in section 4.2 were run on a number of different sentences. The sentences are not in any way representative of English text and are merely designed to exercise different modules to the extent that any anomalies could be detected. The resulting data is in the form of number of input and output values to each module. For instance, over x number of sentences module `case-filter` was executed on y inputs and filtered out z of them. This sort of data was collected under a variety of different search strategies, sometimes with several different parts of the search language disabled. Thus improvements resulting directly from one technique or another can be observed.

One of the important aspects of the search language is its potential for concurrent execution. Although the implementations were tested in a simulated parallel environment, there are an enormous number of details relating to implementing such parsers in a truly concurrent environment that could not be adequately addressed here. So the only data provided on concurrent execution assumes infinite resources and zero overhead for both process creation and communication. Obviously this would not be the case in any realistic scenario.

Actual execution times and memory usages are not provided. Neither the search language nor the parsers' implementations were optimized for efficiency and such results would be misleading. We assume that the amount of computational resources expended by a module is directly proportional to the maximum of the number of inputs to it and the number of outputs from it. This is not necessarily true, and certain modules have smaller coefficients associated with them than others, but our goal is only to provide generalizations about the value of various search techniques, not to provide quantitative exactitudes.

4.3.1 Test Implementations

Three linguistic theories are tested, each under two different search strategies. The base of all three theories is described in section 4.2.1, and the variations are: `FD`, functional determination of empty categories implemented as a filter over freely determined empty categories, with a non-deterministic chain formation algorithm; `SD`, Correa's structural determination algorithm for empty category typology (the functional determination filter is also applied at the end of the search) with a non-deterministic chain formation algorithm; and `CF`, structural determination with Correa's deterministic chain formation algorithm. Each of the three linguistic theories is implemented both in a form that modularizes search as much as possible and as a standard depth-first-search implementation ordered in as efficient a manner as possible.

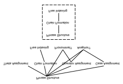


Figure 4.4: The dependencies between the generators in `SD`, the structural-determination parser.

The generator dependencies of the three linguistic theories are shown in figures 4.1, 4.4 and 4.5. In each case the dashed box diagrams the dependencies between non-deterministic generators (see section 2.3.3 for a discussion of why this is important). The exact search strategies for the modular and DFS implementations of each theory can be found in Appendix B.

The three parsers all produce slightly different output for the testbed of sentences, because structural determination produces slightly different empirical results than functional determination, and because Correa's deterministic chain formation algorithm does not work with the parasitic gap sentences in the testbed, unlike the non-deterministic algorithm found in `FD` and `SD`.

4.3.2 Result Summary

Figure 4.6 presents quantitative estimates of the work done by each of the six implementations under a variety of different search parameters. As discussed above, the numbers are the sum over each sentence of the sum of the work done by each module, where each module's work is defined to be the maximum of the number of input values and the number of values it produced (for a given sentence). To provide some sense of how much improvement could be gained by a concurrent implementation, the sum is also computed assuming that after each generator is executed, different processes apply to each result. This means that each module only contributes the maximum of the input and output over the single most complex call, not all its executions. Of course a further concurrency optimization would be to execute different modules in parallel if their dependencies permit it, such as doing all case theory work separately

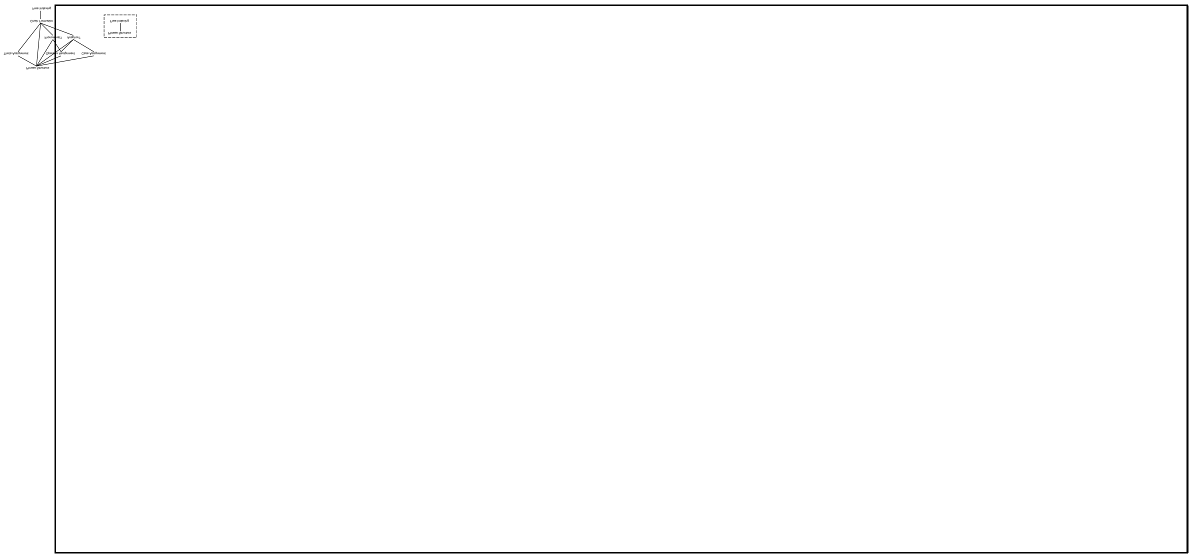


Figure 4.5: The dependencies between the generators in `CF`, the structural-determination parser with deterministic chain formation.

from theta theory work, which would result in even lower numbers.

For each of the 3 linguistic theories and search strategies, five other parameters are varied. The first (*Dep.*) is + if user-declared data-dependency failures (mstructure failures) are used; the second (*Non-Local*) is + if non-local backtracking is used; the third (*Memo.*) is + if memoized functions are used; the fourth (*All Values*) is + if all paths are explored and the search procedure is not stopped after one solution is found; and the fifth (*SD*) is + if pure structural determination is used rather than using the algorithm as a heuristic to order the empty category typology search.

4.4 Qualitative Summary

This section interprets the results from figure 4.6, discussing the efficiency gains different search techniques produce. Frequent references are made to the test numbers in figure 4.6.

4.4.1 Modularity of Search

Most of the facilities the search language provides are there to allow independent modules to be executed independently, without the combinatorial explosion that would normally result from a depth first search. How much does this facility actually improve the search efficiency of the test parsers?

Test	*Parser*	*Mod/DFS*	*Dep.*	*Non-Local*	*Memo.*	*All Values*	*SD*	*App.*	*Standard*	||
1	FD	Mod	+	+	+	+		A	3775 (1)	594 (1)
2	FD	Mod	+	+	-	+		B	3798	(1)
3	FD	Mod	+	-	+	+			(1)	(1)
4	FD	Mod	+	+	+	-			(1)	(1)
5	FD	Mod	-	+	+	+		C	3807 (1)	(1)
6	FD	DFS	+	+	+	+		D	5323 (2)	588 (2)
7	FD	DFS	+	+	+	-		E	3526 (3)	404 (3)
8	FD	DFS	+	-	+	+			(2)	(2)
9	FD	DFS	+	-	+	-			(3)	(3)
10	FD	DFS	-	+	+	+			5386	593
11	FD	DFS	-	+	+	-			3549	(3)
12	SD	Mod	+	+	+	+	+	F	2732 (4)	556 (4)
13	SD	Mod	+	+	+	+	-	G	3781	594
14	SD	Mod	+	+	+	-	+		(4)	(4)
15	SD	Mod	-	+	+	+	+		2733	(4)
16	SD	DFS	+	+	+	+	+		3095	555
17	SD	DFS	+	+	+	+	-		5367	589
18	SD	DFS	+	+	+	-	+		2379	439
19	SD	DFS	+	+	+	-	-		3491	404
20	CF	Mod	+	+	+	+	+	H	823 (5)	410 (5)
21	CF	Mod	+	+	+	+	-		1018	450
22	CF	Mod	+	+	+	-	+		(5)	(5)
23	CF	Mod	-	+	+	+	+		824	(5)
24	CF	DFS	+	+	+	+	+	I	(5)	409
25	CF	DFS	+	+	+	-	+	J	677	323
26	CF	DFS	+	+	+	-	-		819	352

Figure 4.6: Quantitative test results. Numbers in parenthesis are references to previous values. Appendix A contains a breakdown of computation by module for each of the tests in figure 4.6 that have a letter by them.

In the base case, comparing the modular and DFS versions of the `FD` parser (tests 1 and 6) the modularization results in a fairly significant reduction in work (29%), from 5323 to 3775. Examining the data in Appendix A more carefully it is clear where such improvement has come from. The `pronominal?` generator is called only 23 times (for 56 outputs) in the modularized parser, but in the DFS parser the redundant effort of calling the module after every other generator results in 216 calls with 1022 outputs. No matter how the search is rearranged, in DFS some generators will needlessly be executed after each branch of a non-deterministic generator.

Changes in the parser, however, quickly improve DFS's relative efficiency. For instance, in parsers with fewer non-deterministic generators it is possible to create a better ordering, and the difference between the modular and DFS versions of `SD` (tests 12 and 16) is only 3095 vs. 2732. Figure 4.5 indicates that because there is only one non-deterministic generator other than `phrase-structure` it can be ordered last and DFS can be just as efficient as a modular approach. Tests 20 and 24 bear this conclusion out.

Depth first search actually becomes substantially more efficient than a modular approach if only one solution is required[1] instead of a full search. The modular parsers have essentially computed every value before any can be outputed, whereas DFS can stop before many avenues of the search have been explored. For the `FD` parser, returning one value (tests 4 and 7) does not improve the performance of the modular version at all but DFS reduces to 3526, slightly more efficient than modular. Similar improvements can be found for `SD` (tests 14 and 18) and `FD` (tests 22 and 25).

4.4.2 Failure Propagation

The user-declared data-dependencies (section 3.6.2) that the search language provides (in conjunction with memoization) the opportunity to gain extra efficiency by sharing data structures, since the failure of any single component can squash several search paths at once. Tests 1 and 5; 12 and 15; 20 and 23 compare the base modular parsers with ones where this failure propagation mechanism has been disabled. The results are similar for the DFS versions in tests 6 and 10; 7 and 11.

In the `FD` parser there is a noticable, though small, improvement with the user-declared failure propagation mechanism (3775 vs. 3807). Comparing the data in Appendix A, we can see that the `lexical-case-filter` was applied 43 times in one case and 42 in the other. Apparently one search path was eliminated through the propagation of failure. The `empty-case-filter` utilized the mechanism to a greater extent, being applied 54 times instead of 85. But in the `SD` and `FD` parsers the `anaphor?` generator is deterministic and therefore the `empty-case-filter` is never applied more than 40 times, and the only savings is the single unit from the `lexical-case-filter`.

4.4.3 Concurrency

In figure 4.6 the numbers under the $||$ column reflect the number of calls made in the longest search thread, assuming that after every generator applies each of its results are processed concurrently. It does not assume that modules with no interdependencies are processed concurrently. The numbers are very

---

[1] This is perfectly reasonable in many parsing context.

low when compared with the standard sequential model, and would be much lower still were independent modules processed concurrently.

As previously discussed, these numbers are misleading, since they do not reflect the profound effects different computer architectures and consequent processing overheads would have on a concurrent implementation. For now, these numbers illustrate two important points: the first is that with unlimited parallelism the non-determinism found in the `FD` parser is little less efficient than the more deterministic `SD` and `CF` parsers; and the second is that concurrency holds great potential for improved efficiency.

4.4.4 Non-Local Exits

The non-local exiting described in section 3.6.1 is not exercised in these tests and therefore does not improve performance at all (tests 6 and 8; 7 and 9).

4.4.5 Pure vs. Ordered Structural Determination

Structural determination of empty category typology can be implemented to deterministically set the anaphoric and pronominal character of each empty category, or as an ordering on the search, so that the value for each that is chosen first is the one that structural determination would select. This sidesteps Fong's criticism that structural determination is not be descriptively adequate, so long as the functional determination filter is eventually applied.

The data in Appendix A (A, F) shows that structural determination does produce different results than functional determination. The functional determination filter is applied in the `SD` parser even though structural determination is used to initially set empty category typology. Only 23 solutions are found by this combination, though the `FD` parser finds 27. So using structural determination as an efficiency-motivated ordering heuristic is indeed necessary if descriptive equivalence is to be maintained.

Of course, as tests 1 and 13 show, there is no efficiency gain if all search paths are examined. And to efficiently extract only one solution, DFS must be used. Looking at tests 18 and 19 we can see that achieving the accuracy the ordering brings costs some (3491 vs. 2379) but is still cheaper than examining all solutions without SD (test 1, 3775).

4.4.6 Memoization

Memoization (section 3.7) can improve performance in two ways. The first is by causing a function that is applied to the same argument multiple times to do less work. The second (section 3.6.2) is that since a memoized function returns identical results when applied to the same argument, data structures returned will be shared and therefore will further improve memoization potential and allow user-declared data dependency failure to apply in more cases. Unfortunately, since memoization is inherent in the tabular CFG parser used in the `phrase-structure` generator, it is difficult to turn it off for testing

Function	*Calls with Memo.*	*Hits*	*Misses*	*Calls without Memo.*
`C-COMMANDER`	4827	4406	421	17425
`FREELY-INDEX`	58	22	36	110
`DO-CHAIN-FORMATION`	586	199	387	10212
`SET-PHRASE-PARENTS`	906	276	630	1772
`ANNOTATE-PHRASE-WITH-CASE`	501	96	405	927
`ANNOTATE-PHRASE-WITH-THETA`	1109	350	759	1604
`ANNOTATE-EMPTY-CATEGORIES-WITH-PRONOMINAL-FEATURE`	395	61	334	626
`ANNOTATE-EMPTY-CATEGORIES-WITH-PRONOMINAL-FEATURE`	488	101	387	1025

Figure 4.7: Memoization results. Presented for each memoized function are the number of calls to it with memoization, the number of times a hit occurred (the function had already been computed on the arguments), the number of misses, and the number of calls without memoization enabled.

purposes. So although the following results indicate that memoization does not generate spectacular improvements, they do not tell the whole story.

Tests 1 and 2 compare the effect memoization has on the number of times modules are called in the base `FD` parser. There is actually a difference, 3798 vs. 3775, reflecting (as described above) how the common data structures output by memoized functions allow greater use of user-declared data dependencies. So not surprisingly, the improvements come in `lexical-case-filter` and `empty-case-filter`, the same modules that user-declared data dependencies affect.

Of course the primary means by which memoization improves efficiency is in terms of function calls, which are not listed in the data from figure 4.6. Figure 4.7 compares tests 1 and 2 in terms of function calls to memoized functions.

From figure 4.7 it is clear that memoization has an enormous impact on these functions. Without memoization `c-commander` is called 17,425 times but with memoization it is only called 4827 times, and 421 of these times a memoized set of values is returned. For `do-chain-formation` only 387 new values need to be computed instead of 10,212. If these figures seem high, remember that every hit on a memoized function also eliminates all the recursive calls that would have been made during the application.

Chapter 5

Conclusions

This chapter summarizes the test results presented in chapter 4, discussing their implications for the search language and principle-based parsers, and to what extent these results can be generalized. It then examines the implications of deterministic linguistic theories and talks about future directions for this research.

5.1 An Evaluation of the Search Language

There is a lot of moderately interesting information that can be gleaned from tests like those in chapter 4: how a special search mechanism makes certain parsing computations more or less efficient; what effect a small variation in linguistic theory has on descriptive adequacy or computational cost; whether certain tools in the programming environment make it easier to implement various linguistic conditions. Rather than exploring such details, we draw two broad generalizations from the experience of coding and running the test parsers in the search language:

- The programming language makes it very easy to implement generate-and-test search problems, especially this particular brand of principle-based parsers. Easier, in fact, than other languages designed specifically for this purpose like Fong's PROLOG-based system because of the richness of the SCREAMER/COMMON LISP base and natural module connection mechanism.
- Modularity in search, error propagation in data structures, memoization and other special techniques improve the efficiency of a parser implementation to some extent or other, but the small changes they create pale in respect to the effect of a change in the search problem itself, such as the substitution of deterministic linguistic modules for modules with high branching factors.

These statements reflect the particular experiences from the tests described in chapter 4 and others not described here. It is an important question whether they generalize to other parsers and theories, such as ones that might be more psychologically plausible in their use of word-by-word information. While it's true that it is easy to formulate a problem that is greatly affected by the features of the search language (for instance, undoubtably modularization would have compared very well to a DFS implementation that used a worst-case ordering), there is no reason to believe that other linguistic theories will be any more decomposable than the simple subset of Government-Binding implemented here, or that theories will lead to so much shared structure that fancy error propagation techniques are far more efficient than the brute speed of a simple implementation. A reasonable expectation is that the statements are broadly relevant.

The one exception to the second remark may be concurrency. This research shows that there is no reason in theory that a principle-based parser can not be run in a concurrent environment with greatly improved execution time, but it has not demonstrated that the constants involved would necessarily make such an implementation worthwhile.

These results are not particularly surprising. The problem of syntactically parsing word strings is well-known to be highly combinatoric, given no determinism-inducing heuristics or semantic constraints. There are obvious limits to how much even the most sophisticated search engine can improve the efficiency of an inherently complex problem. Reducing the complexity of the problem through deterministic linguistic theories is the only reasonable approach to improving search efficiency.

5.2 Deterministic Linguistic Theories

Why didn't modularization of search help more than it did? Given the number of generators in the basic functional-determination parser one could reasonably have expected modularization to have a big impact. The answer lies in the nature of the generator dependencies. Many are deterministic, but the 3 generators with the highest branching factors (phrase-structure, chain-formation and free-indexing) are all in a dependency chain. They can not run independently. Even using a deterministic chain-formation theory (and this can only be done by moving some aspects of chain formation, such as the insertion of empty categories, into other generators) phrase-structure still feeds directly into free-indexing. This will be true in any reasonable linguistic theory, and it is impossible under the implied parsing-problem specification to make either phrase-structure or free-indexing deterministic, since a given string of words can be ambiguous with respect to both structure and anaphoric reference.

Much of the reason that the linguistic theory must be implemented non-deterministically is because of the lack of information the theory assumes. In a more psychologically plausible linguistic setting a parser would have knowledge about what tree structures were preferred, and what the likely antecedent for a given pronoun is. It is quite conceivable that with this information an adequate deterministic theory of free-indexing (see [18], [16]) or phrase-structure ([22]) exists. Therefore it is not unreasonable to expect that there is a descriptively accurate, efficiently implementable, deterministic theory of language waiting to be discovered. Correa's deterministic algorithms are not adequate descriptively, but that is no reason to abandon the effort to find better ones.

Deterministic theories eliminate the need for search, and also are an indication of how a seemingly small change in linguistic theory can have enormous computational consequence. For this reason much of this research and others in its vein are likely to be wasted effort. In such dynamic conditions, it is fruitless to think much about implementing current theories in more efficient ways when in the future the theory itself will probably have changed into a completely different character, one that may well have a transparent efficient implementation. And certainly if the goal is to produce a practical parser based on the current (flawed) theories, there is little lost by taking advantage of efficient algorithms like Correa's.

5.3 Future Work

If changes in linguistic theory are likely to affect the way we build parsers and the computational nature of these implementations far more than the availability of better tools, then it is more fruitful to explore variations in linguistic theory than to expand on this or other search languages. The one interesting and potentially great source of efficiency that is left largely unexplored in this work is concurrent execution of various search paths. The search language described here demonstrates that it is not necessary to rethink or recode a search problem to take advantage of concurrency, but the work of actually implementing the search language in a concurrent environment and testing the resulting parsers has been left for future work.

Appendix A

Test Results

This appendix presents detailed information about the tests described in figure 4.6. For each of the 10 lettered tests the number of inputs to and outputs from each module are listed, summed over the 15 test sentences.

For each test, there is a list that looks like

```
FUNCTIONAL-DETERMINATION      62   27
...
CHAIN-FORMATION               40   221
...
THETA-ROLE-ASSIGNMENT         80   43
PHRASE-STRUCTURE              15   80
```

In this case, reading from the bottom, the `phrase-structure` generator was called on 15 different values (15 sentences) and output a total of 80 different phrase structure trees. The `theta-role-assignment` generator was executed on 80 different inputs (the 80 trees) but output only 43 values, indicating that it failed completely for some inputs. In contrast, the `chain-formation` generator was called on 40 different structures but output a total of 221 different chains. Finally, the `functional-determination` filter was called 62 times and rejected all but 27 values. Since the `functional-determination` filter is dependent on all of the non-deterministic generators, for each of tests the output column of this filter is the number of different solutions found by the parser. In the above case, the 15 sentences had a total of 27 different parses (though some had none).

A. Test 1. [FD, modular.]

FUNCTIONAL-DETERMINATION	62	27
CONDITION-C-VAR	146	69
LICENSE-CHAINS	99	19
CONDITION-B	1022	727
PRONOMINAL?	23	56
CONDITION-A	234	216
I-WITHIN-I	234	234
COINDEX-OPERATOR	242	234
CONDITION-C-REXP	297	242
FREE-INDEXING	35	297
THETA-CRITERION	217	35
SUBJACENCY	221	217
CHAIN-FORMATION	40	221
EMPTY-CASE-FILTER	54	40
ANAPHOR?	40	85
LEXICAL-CASE-FILTER	42	40
CASE-ASSIGNMENT	43	43
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

B. Test 2. [FD, modular, no memoization.]

FUNCTIONAL-DETERMINATION	62	27
CONDITION-C-VAR	146	69
LICENSE-CHAINS	99	19
CONDITION-B	1022	727
PRONOMINAL?	23	56
CONDITION-A	234	216
I-WITHIN-I	234	234
COINDEX-OPERATOR	242	234
CONDITION-C-REXP	297	242
FREE-INDEXING	35	297
THETA-CRITERION	217	35
SUBJACENCY	221	217
CHAIN-FORMATION	40	221
EMPTY-CASE-FILTER	76	40
ANAPHOR?	40	85
LEXICAL-CASE-FILTER	43	40
CASE-ASSIGNMENT	43	43
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

C. Test 5. [FD, modular, no data dependency failures.]

FUNCTIONAL-DETERMINATION	62	27
CONDITION-C-VAR	146	69
LICENSE-CHAINS	99	19
CONDITION-B	1022	727
PRONOMINAL?	23	56
CONDITION-A	234	216
I-WITHIN-I	234	234
COINDEX-OPERATOR	242	234
CONDITION-C-REXP	297	242
FREE-INDEXING	35	297
THETA-CRITERION	217	35
SUBJACENCY	221	217
CHAIN-FORMATION	40	221
EMPTY-CASE-FILTER	85	40
ANAPHOR?	40	85
LEXICAL-CASE-FILTER	43	40
CASE-ASSIGNMENT	43	43
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

D. Test 6. [FD, DFS.]

FUNCTIONAL-DETERMINATION	62	27
CONDITION-C-VAR	131	62
LICENSE-CHAINS	727	131
CONDITION-B	1022	727
PRONOMINAL?	216	1022
CONDITION-A	234	216
I-WITHIN-I	234	234
COINDEX-OPERATOR	242	234
CONDITION-C-REXP	297	242
FREE-INDEXING	35	297
THETA-CRITERION	217	35
SUBJACENCY	221	217
CHAIN-FORMATION	40	221
EMPTY-CASE-FILTER	54	40
ANAPHOR?	40	54
LEXICAL-CASE-FILTER	42	40
CASE-ASSIGNMENT	43	42
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

E. Test 7. [FD, DFS, one solution.]

FUNCTIONAL-DETERMINATION	29	11
CONDITION-C-VAR	57	29
LICENSE-CHAINS	475	57

CONDITION-B	657	475
PRONOMINAL?	140	657
CONDITION-A	157	140
I-WITHIN-I	157	157
COINDEX-OPERATOR	165	157
CONDITION-C-REXP	209	165
FREE-INDEXING	32	209
THETA-CRITERION	137	32
SUBJACENCY	139	137
CHAIN-FORMATION	35	139
EMPTY-CASE-FILTER	47	35
ANAPHOR?	35	47
LEXICAL-CASE-FILTER	37	35
CASE-ASSIGNMENT	38	37
OPERATOR-ASSIGNMENT	38	38
THETA-ROLE-ASSIGNMENT	66	38
PHRASE-STRUCTURE	15	66

F. Test 12. [`SD`, modular.]

FUNCTIONAL-DETERMINATION	42	23
CONDITION-C-VAR	86	42
LICENSE-CHAINS	34	16
CONDITION-B	216	204
PRONOMINAL?	23	23
CONDITION-A	234	216
I-WITHIN-I	234	234
COINDEX-OPERATOR	242	234
CONDITION-C-REXP	297	242
FREE-INDEXING	35	297
THETA-CRITERION	217	35
SUBJACENCY	221	217
CHAIN-FORMATION	40	221
EMPTY-CASE-FILTER	40	40
ANAPHOR?	40	40
LEXICAL-CASE-FILTER	42	40
CASE-ASSIGNMENT	43	43
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

G. Test 13. [`SD`, modular, ordered SD.]

FUNCTIONAL-DETERMINATION	69	27
CONDITION-C-VAR	146	69
LICENSE-CHAINS	99	19
CONDITION-B	1022	727
PRONOMINAL?	23	56
CONDITION-A	234	216
I-WITHIN-I	234	234
COINDEX-OPERATOR	242	234
CONDITION-C-REXP	297	242
FREE-INDEXING	35	297
THETA-CRITERION	217	35
SUBJACENCY	221	217
CHAIN-FORMATION	40	221
EMPTY-CASE-FILTER	53	40
ANAPHOR?	40	85
LEXICAL-CASE-FILTER	42	40
CASE-ASSIGNMENT	43	43
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

H. Test 20. [`CF`, modular.]

FUNCTIONAL-DETERMINATION	23	15
I-WITHIN-I	27	23
COINDEX-OPERATOR	29	27
CONDITION-C-VAR	40	29
CONDITION-C-REXP	50	40
CONDITION-B	55	50
CONDITION-A	59	55
FREE-INDEXING	11	59
LICENSE-CHAINS	11	11
THETA-CRITERION	11	11
SUBJACENCY	11	11
CHAIN-FORMATION	40	11
PRONOMINAL?	40	40
EMPTY-CASE-FILTER	40	40
ANAPHOR?	40	40
LEXICAL-CASE-FILTER	42	40
CASE-ASSIGNMENT	43	42
OPERATOR-ASSIGNMENT	43	43
THETA-ROLE-ASSIGNMENT	80	43
PHRASE-STRUCTURE	15	80

I. Test 24. [`CF`, DFS.]

FUNCTIONAL-DETERMINATION	23	15
I-WITHIN-I	27	23
COINDEX-OPERATOR	29	27
CONDITION-C-VAR	40	29
CONDITION-C-REXP	50	40
CONDITION-B	55	50
CONDITION-A	59	55
FREE-INDEXING	11	59
LICENSE-CHAINS	11	11

```
THETA-CRITERION              11   11
SUBJACENCY                   11   11
CHAIN-FORMATION              40   11
PRONOMINAL?                  40   40
EMPTY-CASE-FILTER            40   40
ANAPHOR?                     40   40
LEXICAL-CASE-FILTER          42   40
CASE-ASSIGNMENT              43   42
OPERATOR-ASSIGNMENT          43   43
THETA-ROLE-ASSIGNMENT        80   43
PHRASE-STRUCTURE             15   80
```

J. Test 25. [`CF`, DFS, one solution.]

```
FUNCTIONAL-DETERMINATION     17   9
I-WITHIN-I                   21   17
COINDEX-OPERATOR             23   21
CONDITION-C-VAR              29   23
CONDITION-C-REXP             32   29
CONDITION-B                  32   32
CONDITION-A                  34   32
FREE-INDEXING                11   34
LICENSE-CHAINS               11   11
THETA-CRITERION              11   11
SUBJACENCY                   11   11
CHAIN-FORMATION              38   11
PRONOMINAL?                  38   38
EMPTY-CASE-FILTER            38   38
ANAPHOR?                     38   38
LEXICAL-CASE-FILTER          40   38
CASE-ASSIGNMENT              41   40
OPERATOR-ASSIGNMENT          41   41
THETA-ROLE-ASSIGNMENT        74   41
PHRASE-STRUCTURE             15   74
```

Appendix B

Parser Code

This appendix presents the actual code needed to implement the test parsers described in chapter 4.

B.1 `FD` Parser

```
(defmstructure (phrase :allow-dependencies t)
    ((category :initarg :category :accessor phrase-category :type symbol)
     (daughters :initarg :daughters :accessor phrase-daughters :type list)
     (parent :initform nil :accessor phrase-parent :nondestructive (phrase-structure))
     (inherent-features :initform nil :initarg :inherent-features
                        :accessor phrase-inherent-features :type list)
     (theta-role :initform nil :accessor phrase-theta-role :type symbol
                 :nondestructive (theta-role-assignment))
     (anaphor? :initform nil :accessor phrase-anaphor? :type symbol
               :nondestructive (anaphor? operator-assignment))
     (pronominal? :initform nil :accessor phrase-pronominal? :type symbol
                  :nondestructive (pronominal? operator-assignment))
     (assigned-case :initform nil :accessor phrase-assigned-case :type symbol
                    :nondestructive (case-assignment)))
  (set-phrase-parents annotate-phrase-with-case
                      do-chain-formation
                      annotate-phrase-with-theta
                      annotate-empty-categories-with-anaphor-feature
                      annotate-empty-categories-with-pronominal-feature)
  )

(define-tree-positions '((top PHRASE-STRUCTURE)
                         (generate phrase-structure THETA-ROLE-ASSIGNMENT)
                         (generate phrase-structure CHAIN-FORMATION)
                         (generate phrase-structure OPERATOR-ASSIGNMENT)
                         (generate phrase-structure CASE-ASSIGNMENT)
```

```
                          (generate chain-formation FREE-INDEXING)
                          (generate operator-assignment ANAPHOR?)
                          (generate operator-assignment PRONOMINAL?)
                          (cross free-indexing anaphor? ANAPHOR-FI)
                          (cross free-indexing pronominal? PRON-FI)
                          (cross chain-formation theta-role-assignment CF-TR)
                          (cross case-assignment anaphor? CA-A)
                          (cross anaphor? pronominal? ANAPRO)
                          (cross anapro chain-formation VARIABLES)
                          (cross pron-fi anaphor-fi A1)
                          (cross a1 ca-a A2)
                          (cross a2 variables a3)
                          (cross a3 cf-tr ALL)
                          ))

(defsearcher MOD-PARSER ((generate PHRASE-STRUCTURE)
                       (generate THETA-ROLE-ASSIGNMENT)
                       (generate OPERATOR-ASSIGNMENT)
                       (generate CASE-ASSIGNMENT)
                       (filter LEXICAL-CASE-FILTER case-assignment)
                       (generate ANAPHOR?)
                       (generate CA-A)
                       (filter EMPTY-CASE-FILTER ca-a)
                       (generate CHAIN-FORMATION)
                       (filter SUBJACENCY chain-formation)
                       (generate CF-TR)
                       (filter THETA-CRITERION cf-tr)
                       (generate FREE-INDEXING)
                       (filter CONDITION-C-REXP free-indexing)
                       (filter COINDEX-OPERATOR free-indexing)
                       (filter I-WITHIN-I free-indexing)
                       (generate ANAPHOR-FI)
                       (filter CONDITION-A anaphor-fi)
                       (generate PRONOMINAL?)
                       (generate PRON-FI)
                       (filter CONDITION-B pron-fi)
                       (generate ANAPRO)
                       (generate VARIABLES)
                       (filter LICENSE-CHAINS variables)
                       (generate A1)
                       (filter CONDITION-C-VAR a1)
                       (generate A2)
                       (filter FUNCTIONAL-DETERMINATION a2)
                       (generate A3)
                       (generate ALL)
                       (filter PRINTER all)
                       ))

(defsearcher DFS-PARSER ((generate PHRASE-STRUCTURE
                       (generate THETA-ROLE-ASSIGNMENT
                       (generate OPERATOR-ASSIGNMENT
                       (generate CASE-ASSIGNMENT
                       (filter LEXICAL-CASE-FILTER case-assignment)
```

```
                      (generate ANAPHOR?
                      (generate CA-A
                      (filter EMPTY-CASE-FILTER ca-a)
                      (generate CHAIN-FORMATION
                      (filter SUBJACENCY chain-formation)
                      (generate CF-TR
                      (filter THETA-CRITERION cf-tr)
                      (generate FREE-INDEXING
                      (filter CONDITION-C-REXP free-indexing)
                      (filter COINDEX-OPERATOR free-indexing)
                      (filter I-WITHIN-I free-indexing)
                      (generate ANAPHOR-FI
                      (filter CONDITION-A anaphor-fi)
                      (generate PRONOMINAL?
                      (generate PRON-FI
                      (filter CONDITION-B pron-fi)
                      (generate ANAPRO
                      (generate VARIABLES
                      (filter LICENSE-CHAINS variables)
                      (generate A1
                      (filter CONDITION-C-VAR a1)
                      (generate A2
                      (filter FUNCTIONAL-DETERMINATION a2)
                      (generate A3
                      (generate ALL
                      (filter PRINTER all)
                      ))))))))))))))))))))
```

B.2 SD Parser

```
(defmstructure (phrase :allow-dependencies t)
    ((category :initarg :category :accessor phrase-category :type symbol)
     (daughters :initarg :daughters :accessor phrase-daughters :type list)
     (parent :initform nil :accessor phrase-parent :nondestructive (phrase-structure))
     (inherent-features :initform nil :initarg :inherent-features
                        :accessor phrase-inherent-features :type list)
     (theta-role :initform nil :accessor phrase-theta-role :type symbol
                 :nondestructive (theta-role-assignment))
     (anaphor? :initform nil :accessor phrase-anaphor? :type symbol
               :nondestructive (anaphor? operator-assignment))
     (pronominal? :initform nil :accessor phrase-pronominal? :type symbol
                  :nondestructive (pronominal? operator-assignment))
     (assigned-case :initform nil :accessor phrase-assigned-case :type symbol
                    :nondestructive (case-assignment)))
  (set-phrase-parents annotate-phrase-with-case do-chain-formation
                      annotate-phrase-with-theta)
  )

(define-tree-positions '((top PHRASE-STRUCTURE)
```

```
                          (generate phrase-structure THETA-ROLE-ASSIGNMENT)
                          (generate phrase-structure CHAIN-FORMATION)
                          (generate phrase-structure OPERATOR-ASSIGNMENT)
                          (generate phrase-structure CASE-ASSIGNMENT)
                          (generate chain-formation FREE-INDEXING)
                          (cross operator-assignment case-assignment OP-CASE)
                          (generate op-case ANAPHOR?)
                          (generate operator-assignment PRONOMINAL?)
                          (cross free-indexing anaphor? ANAPHOR-FI)
                          (cross free-indexing pronominal? PRON-FI)
                          (cross chain-formation theta-role-assignment CF-TR)
                          (cross case-assignment anaphor? CA-A)
                          (cross anaphor? pronominal? ANAPRO)
                          (cross anapro chain-formation VARIABLES)
                          (cross pron-fi anaphor-fi A1)
                          (cross a1 variables a2)
                          (cross a2 cf-tr ALL)
                          ))

(defsearcher MOD-PARSER ((generate PHRASE-STRUCTURE)
                       (generate THETA-ROLE-ASSIGNMENT)
                       (generate OPERATOR-ASSIGNMENT)
                       (generate CASE-ASSIGNMENT)
                       (filter LEXICAL-CASE-FILTER case-assignment)
                       (generate OP-CASE)
                       (generate ANAPHOR?)
                       (filter EMPTY-CASE-FILTER anaphor?)
                       (generate CHAIN-FORMATION)
                       (filter SUBJACENCY chain-formation)
                       (generate CF-TR)
                       (filter THETA-CRITERION cf-tr)
                       (generate FREE-INDEXING)
                       (filter CONDITION-C-REXP free-indexing)
                       (filter COINDEX-OPERATOR free-indexing)
                       (filter I-WITHIN-I free-indexing)
                       (generate ANAPHOR-FI)
                       (filter CONDITION-A anaphor-fi)
                       (generate PRONOMINAL?)
                       (generate PRON-FI)
                       (filter CONDITION-B pron-fi)
                       (generate ANAPRO)
                       (generate VARIABLES)
                       (filter LICENSE-CHAINS variables)
                       (generate A1)
                       (filter CONDITION-C-VAR a1)
                       (filter FUNCTIONAL-DETERMINATION a1)
                       (generate A2)
                       (generate ALL)
                       (filter PRINTER all)
                       ))

(defsearcher DFS-PARSER ((generate PHRASE-STRUCTURE
                       (generate THETA-ROLE-ASSIGNMENT
```

```
                      (generate OPERATOR-ASSIGNMENT
                      (generate CASE-ASSIGNMENT
                      (filter LEXICAL-CASE-FILTER case-assignment)
                      (generate OP-CASE
                      (generate ANAPHOR?
                      (filter EMPTY-CASE-FILTER anaphor?)
                      (generate CHAIN-FORMATION
                      (filter SUBJACENCY chain-formation)
                      (generate CF-TR
                      (filter THETA-CRITERION cf-tr)
                      (generate FREE-INDEXING
                      (filter CONDITION-C-REXP free-indexing)
                      (filter COINDEX-OPERATOR free-indexing)
                      (filter I-WITHIN-I free-indexing)
                      (generate ANAPHOR-FI
                      (filter CONDITION-A anaphor-fi)
                      (generate PRONOMINAL?
                      (generate PRON-FI
                      (filter CONDITION-B pron-fi)
                      (generate ANAPRO
                      (generate VARIABLES
                      (filter LICENSE-CHAINS variables)
                      (generate A1
                      (filter CONDITION-C-VAR a1)
                      (filter FUNCTIONAL-DETERMINATION a1)
                      (generate A2
                      (generate ALL
                      (filter PRINTER all)
                      )))))))))))))))))))
```

B.3 `CF` Parser

```
(defmstructure (phrase :allow-dependencies t)
    ((category :initarg :category :accessor phrase-category :type symbol)
     (daughters :initarg :daughters :accessor phrase-daughters :type list)
     (parent :initform nil :accessor phrase-parent :nondestructive (phrase-structure))
     (inherent-features :initform nil :initarg :inherent-features
                        :accessor phrase-inherent-features :type list)
     (theta-role :initform nil :accessor phrase-theta-role :type symbol
                 :nondestructive (theta-role-assignment))
     (anaphor? :initform nil :accessor phrase-anaphor? :type symbol
               :nondestructive (anaphor? operator-assignment))
     (pronominal? :initform nil :accessor phrase-pronominal? :type symbol
                  :nondestructive (pronominal? operator-assignment))
     (assigned-case :initform nil :accessor phrase-assigned-case :type symbol
                    :nondestructive (case-assignment)))
  (set-phrase-parents annotate-phrase-with-case get-chains
                      annotate-phrase-with-theta)
  )
```

```
(define-tree-positions '((top PHRASE-STRUCTURE)
                         (generate phrase-structure THETA-ROLE-ASSIGNMENT)
                         (generate phrase-structure OPERATOR-ASSIGNMENT)
                         (generate phrase-structure CASE-ASSIGNMENT)
                         (cross case-assignment operator-assignment OP-CASE)
                         (generate op-case ANAPHOR?)
                         (generate operator-assignment PRONOMINAL?)
                         (cross anaphor? pronominal? ANAPRO)
                         (cross anapro theta-role-assignment anaprotheta)
                         (generate anaprotheta CHAIN-FORMATION)
                         (generate chain-formation FREE-INDEXING)
                         ))

(defsearcher MOD-PARSER ((generate PHRASE-STRUCTURE)
                      (generate THETA-ROLE-ASSIGNMENT)
                      (generate OPERATOR-ASSIGNMENT)
                      (generate CASE-ASSIGNMENT)
                      (filter LEXICAL-CASE-FILTER case-assignment)
                      (generate OP-CASE)
                      (generate ANAPHOR?)
                      (filter EMPTY-CASE-FILTER anaphor?)
                      (generate PRONOMINAL?)
                      (generate ANAPRO)
                      (generate ANAPROTHETA)
                      (generate CHAIN-FORMATION)
                      (filter SUBJACENCY chain-formation)
                      (filter THETA-CRITERION chain-formation)
                      (filter LICENSE-CHAINS chain-formation)
                      (generate FREE-INDEXING)
                      (filter CONDITION-A free-indexing)
                      (filter CONDITION-B free-indexing)
                      (filter CONDITION-C-REXP free-indexing)
                      (filter CONDITION-C-VAR free-indexing)
                      (filter COINDEX-OPERATOR free-indexing)
                      (filter I-WITHIN-I free-indexing)
                      (filter FUNCTIONAL-DETERMINATION free-indexing)
                      (filter PRINTER free-indexing)
                      ))

(defsearcher DFS-PARSER ((generate PHRASE-STRUCTURE
                      (generate THETA-ROLE-ASSIGNMENT
                      (generate OPERATOR-ASSIGNMENT
                      (generate CASE-ASSIGNMENT
                      (filter LEXICAL-CASE-FILTER case-assignment)
                      (generate OP-CASE
                      (generate ANAPHOR?
                      (filter EMPTY-CASE-FILTER anaphor?)
                      (generate PRONOMINAL?
                      (generate ANAPRO
                      (generate ANAPROTHETA
                      (generate CHAIN-FORMATION
                      (filter SUBJACENCY chain-formation)
                      (filter THETA-CRITERION chain-formation)
```

```
                      (filter LICENSE-CHAINS chain-formation)
                      (generate FREE-INDEXING
                      (filter CONDITION-A free-indexing)
                      (filter CONDITION-B free-indexing)
                      (filter CONDITION-C-REXP free-indexing)
                      (filter CONDITION-C-VAR free-indexing)
                      (filter COINDEX-OPERATOR free-indexing)
                      (filter I-WITHIN-I free-indexing)
                      (filter FUNCTIONAL-DETERMINATION free-indexing)
                      (filter PRINTER free-indexing)
                      )))))))))))))
```

B.4 Inputs

```
;;;
;;; The Test Sentences
;;;

(defparameter *sentences*
  '((John was killed)
    (who did John say that he saw)
    (the man that John saw)
    (that John saw Mary kill herself)
    (John likes to bicycle)
    (John saw Mary)
    (who did John give a picture of to)
    (what did you file without reading)
    (Mary saw herself)
    (Bill said that he saw Mary shoot him)
    (who did Mary say was killed)
    (John to see Mary) ;; *
    (what did you reading) ;; *
    (John promised kill himself) ;; *
    (who did John kill Mary) ;; *
    ))

;;;
;;; The Grammar
;;;

(defparameter *grammar*
  '((I2 => (N2) I1)
    ;; Sentential Subject.
    ;; (I2 => (C2) I1)
    (I1 => (I0) V2)
    ;; Topicalization.
    ;; (I2 => N2 I2)
    (I0 =>)
    (N2 => D2 N1)
    ;; "Who did John kill [np [t] [that he liked]]?"
```

```
    ;;(N2 => (N2) C2)
    (N2 ==> N2 C2)
    (N2 => N2 P2)
    (N2 =>)
    (N1 => N0)
    (D2 => (D0))
    (D0 =>)
    (V2 => V1)
    (V1 => V1 P2)
    (V1 => V0)
    (V1 => V0 (N2))
    (V1 => V0 C2)
    (C2 => C1)
    (C2 => (N2) C1)
    (C1 => (C0) I2)
    (C1 => I0 I2)
    (C0 =>)
    (P2 => P1)
    (P1 => P0 (N2))
    (P1 => P0 C2)
    ))

;;;
;;; The Lexicon
;;;

(defparameter *lexicon*
  '((john (n2 (r-expression . +)))
    (bill (n2 (r-expression . +)))
    (mary (n2 (r-expression . +)))
    (i (n2 (pronoun . +) (anaphor . -) (required-case . nom)))
    (you (n2 (pronoun . +) (anaphor . -) (required-case . nom)))
    (he (n2 (pronoun . +) (anaphor . -) (required-case . nom)))
    (she (n2 (pronoun . +) (anaphor . -) (required-case . nom)))
    (him (n2 (pronoun . +) (anaphor . -) (required-case . acc)))
    (her (n2 (pronoun . +) (anaphor . -) (required-case . acc)))
    (it (n2 (pronoun . +) (anaphor . -)))
    (myself (n2 (pronoun . -) (anaphor . +)))
    (himself (n2 (pronoun . -) (anaphor . +)))
    (herself (n2 (pronoun . -) (anaphor . +)))
    (likes (v0 (theta . np-cp)))
    (liked (v0 (theta . np-cp)))
    (like (v0 (theta . np-cp)))
    (promise (v0 (theta . cp)) (n0))
    (promised (v0 (theta . cp)))
    (promising (v0 (theta . cp) (tense . -)))
    (think (v0 (theta . cp)))
    (thinks (v0 (theta . cp)))
    (seems (v0 (theta . subjectless-cp)))
    (seem (v0 (theta . subjectless-cp)))
    (know (v0 (theta . np-cp)))
    (knows (v0 (theta . np-cp)))
    (bicycle (v0 (theta . none)) (n0))
```

```
    (saw (v0 (theta . np-cp)) (n0))
    (said (v0 (theta . cp)))
    (say (v0 (theta . cp)))
    (see (v0 (theta . np-cp)))
    (walk (v0 (theta . np-none)) (n0))
    (walks (v0 (theta . np-none)) (n0))
    (walked (v0 (theta . np-none)))
    (walking (v0 (theta . np-none) (tense . -)))
    (won (v0 (theta . np-none)))
    (win (v0 (theta . np-none)) (n0))
    (bothered (v0 (theta . np)))
    (to (i0 (tense . -)) (p0 (theta . np)))
    (of (p0 (theta . np)))
    (without (p0 (theta . np-cp)))
    (with (p0 (theta . np)))
    (killed (v0 (theta . np)))
    (kill (v0 (theta . np)))
    (shoot (v0 (theta . np)))
    (give (v0 (theta . np)))
    (read (v0 (theta . np)))
    (giving (v0 (theta . np) (tense . -)))
    (reading (v0 (theta . np) (tense . -)))
    (file (v0 (theta . np)))
    (the (d0))
    (a (d0))
    (ball (n0))
    (man (n0))
    (dog (n0))
    (picture (n0))
    (book (n0))
    (men (n0))
    (did (i0))
    (was (i0 (passive . +)))
    (is (i0))
    (were (i0))
    (what (n2 (wh . +)))
    (who (n2 (wh . +)))
    (whom (n2 (wh . +)))
    (why (adv (wh . +)))
    (when (adv (wh . +)))
    (where (adv (wh . +)))
    (how (adv (wh . +)))
    (that (c0))))
```

B.5 Phrase Structure Generator

```
(defgenerator PHRASE-STRUCTURE ()
  ;; Build up phrase-structure representations for the words in the variable
  ;; *SENTENCE*.
  (parse (mapcar #'(lambda (word)
```

```
                     (mapcar #'(lambda (e) (list* (car e) `(word . ,word) (cdr e)))
                             (cdr (assoc word *lexicon*))))
                 *sentence*)
         *grammar*))

(defoption '*print-phrase-structures* nil "Print Phrase Structures" :parser)

(defun parse (input grammar)
  ;; Use a chart parser to generate phrase-structure trees.
  (let ((rules (order-rules grammar))
        (categories (all-categories grammar))
        (n (length input)))
    (declare (fixnum n))
    (let ((l (length categories))
          (chart (make-array (list (1+ n) (1+ n) (length categories))
                             :initial-element nil)))
      (declare (fixnum l))
      (dotimes (i n)
        (declare (fixnum i))
        (dolist (word (nth i input))
          (add-to-chart chart i (+ i 1) (position (car word) categories) (car word) nil
                        (cdr word))))
      (dotimes (k (1+ n))
        (declare (fixnum k))
        (dotimes (i (1+ (- n k)))
          (declare (fixnum i))
          (dolist (r rules)
            (case (- (length (the list r)) 2)
              (0 (when (zerop k)
                   (add-to-chart chart i i (position (car r) categories) (car r) nil :empty)))
              (1 (unless (and (symbolp (third r)) (= k 0))
                   (for-effects
                    (let ((d (transform-empty
                              (a-member-of
                               (aref chart i (+ i k) (position (core-cat (third r))
                                                               categories))))))
                      (unless (comp-checker-special-1 d (car r)) (fail))
                      (add-to-chart chart i (+ i k) (position (car r) categories) (car r)
                                    (list d))))))
              (2 (dotimes (j (1+ k))
                   (declare (fixnum j))
                   (unless (or (and (= j 0) (symbolp (third r)))
                               (and (= j k) (symbolp (fourth r))))
                     (for-effects
                      (let ((d1 (transform-empty
                                 (a-member-of
                                  (aref chart i (+ i j) (position (core-cat (third r))
                                                                  categories)))))
                            (d2 (transform-empty
                                 (a-member-of
                                  (aref chart (+ i j) (+ i k) (position (core-cat (fourth r))
                                                                        categories))))))
                        (unless (comp-checker-special-2 d1 d2 (car r)) (fail))
```

```
                        (add-to-chart chart i (+ i k) (position (car r) categories) (car r)
                                      (list d1 d2)))))))
              (otherwise (error "Rule is illegal ~S!" r))))))
      (let ((phrase (a-member-of (aref chart 0 n (an-integer-between 0 (1- l))))))
        ;; Only output CPs and NPs.
        (when (or (cp? phrase) (np? phrase))
          (set-phrase-parents phrase)
          (when *print-phrase-structures* (print (phrase-tree phrase)))
          (return-result phrase))))))

(defun core-cat (cat) (if (symbolp cat) cat (first cat)))

(defm! set-phrase-parents ((phrase) phrase)
  ;; Phrases are shared between different parses, but the PHRASE-PARENT slot
  ;; of a phrase must vary between each parse.  This function sets it using
  ;; set!-local so that as each parse is enumerated the PHRASE-PARENT slots
  ;; are properly updated.
  (mapc #'(lambda (daughter)
            (set-phrase-parents daughter)
            (set!-local (phrase-parent daughter) phrase))
        (phrase-daughters phrase)))

(defun add-to-chart (chart i j k category daughters &optional features)
  (let ((phrase (if (eq features :empty)
                    category
                    (make-instance 'phrase
                                   :category category
                                   :inherent-features features
                                   :daughters daughters))))
    ;; Declare PHRASE to be dependent its daughters.
    (dolist (daughter daughters)
      (is-part-of daughter phrase))
    (push phrase (aref chart i j k))))

(defun transform-empty (phrase)
  (if (typep phrase 'phrase)
      phrase
      (make-instance 'phrase
                     :category phrase
                     :daughters nil)))

(defun all-categories (rules)
  (let ((categories nil))
    (dolist (r rules)
      (pushnew (first r) categories)
      (setf categories (union categories (mapcar #'core-cat (cddr r)))))
    categories))

(defun order-rules (rules)
  (let ((empty-categories (empty-categories rules))
        (rules (copy-list rules)))
    (do ((change t change))
        ((null change))
```

```
      (setq change nil)
      (mapl #'(lambda (l)
                (let ((r2 (first l)))
                  (mapl #'(lambda (ll)
                            (let ((r1 (car ll)))
                              (if (or (and (eq (first r1) (core-cat (third r2)))
                                           (or (null (core-cat (fourth r2)))
                                               (find (core-cat (fourth r2)) empty-categories)))
                                      (and (eq (first r1) (core-cat (fourth r2)))
                                           (or (null (core-cat (third r2)))
                                               (find (core-cat (third r2)) empty-categories))))
                                  (setf change t
                                        (first l) r1
                                        (first ll) r2
                                        r2 r1))))
                        (rest l))))
            rules))
    rules))

(defun empty-categories (rules)
  (let ((empty-categories (mapcar #'car (remove-if #'cddr rules))))
    (do ((old-ecs nil empty-categories))
        ((eq old-ecs empty-categories))
      (dolist (r rules)
        (if (every #'(lambda (c) (find c empty-categories)) (cddr r))
            (pushnew (car r) empty-categories))))
    empty-categories))

(defmethod print-object ((p phrase) stream)
  (format stream "<Phrase ~S:~{ ~S~}>" (phrase-category p)
          (if (null (phrase-daughters p))
              (let ((w (feature-value p 'word)))
                (and w (list w)))
              (mapcar #'phrase-category (phrase-daughters p)))))

(defun phrase-tree (phrase)
  (append (list (phrase-category phrase))
          (if (null (phrase-daughters phrase))
              (let ((word (feature-value phrase 'word)))
                (and word (list word))))
          (mapcar #'phrase-tree (phrase-daughters phrase))))
```

B.6 Binding Theory

```
(deffilter CONDITION-A ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING) ANAPHOR?)
  (map-up-phrase-structure tree (phrase)
    (when (and (np? phrase) (is-anaphor? phrase))
      (let ((gc (governing-category phrase)))
        (unless (or (null gc) (find-binders phrase gc indices))
```

```
          (reject))))))

(deffilter CONDITION-B ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING) PRONOMINAL?)
  (map-up-phrase-structure tree (phrase)
    (when (and (np? phrase) (is-pronominal? phrase))
      (let ((gc (governing-category phrase)))
        (unless (or (null gc) (not (find-binders phrase gc indices)))
          (reject))))))

(deffilter CONDITION-C-REXP ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING))
  (map-up-phrase-structure tree (phrase)
    (when (r-expression? phrase)
      (if (find-if #'a-position? (find-binders phrase tree indices))
          (reject)))))

(deffilter CONDITION-C-VAR ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING)
                            ANAPHOR? PRONOMINAL?)
  (map-up-phrase-structure tree (phrase)
    (when (variable? phrase)
      (let ((binders (find-binders phrase tree indices)))
        (if (and (find-if #'a-position? binders)
                 ;; Addition to theory- condition C does not apply to wh-t's
                 ;; with anaphor antecedents.  Allows Topicalization of form
                 ;; "Himself, John likes"
                 (not (find-if #'is-anaphor? binders)))
            (reject))))))

(defun find-binders (phrase top-phrase bindings)
  (let ((phrases-with-index (get-phrases-with-index (get-index phrase bindings) bindings)))
    (remove-if-not #'(lambda (p) (member p phrases-with-index))
                   (all-values (c-commander phrase top-phrase)))))

(defun local-binder (phrase bindings inside-of)
  (let ((phrases-with-index (get-phrases-with-index (get-index phrase bindings) bindings)))
    (find-if #'(lambda (p) (member p phrases-with-index))
             (all-values (c-commander phrase inside-of)))))

(defun governing-category (phrase)
  (first
   (all-values
    (let ((governor (governor phrase)))
      (unless governor (fail))
      (minimal-phrase-containing (list phrase governor)
                                 :restricted-to '(N2 I2))))))

(defndmemo c-commander ((phrase inside-of &key (restricted-to t))
                        (phrase inside-of restricted-to))
  (if (eq phrase inside-of) (fail))
  (let ((parent (phrase-parent phrase)))
    (unless parent (fail))
    (if (= 1 (length (phrase-daughters parent)))
        (c-commander parent inside-of :restricted-to restricted-to)
        (let ((sister (a-member-of (phrase-daughters parent))))
```

```
          (if (eq sister phrase) (fail))
          (if (and (listp restricted-to)
                   (not (member (phrase-category sister) restricted-to)))
              (fail))
          (either sister
                  (c-commander parent inside-of
                               :restricted-to restricted-to))))))

(defun governor (phrase)
  (let ((potential-governor
         (c-commander phrase (smallest-maximal-projection-containing phrase)
                      :restricted-to '(v0 p0 i1))))
    (if (i1? potential-governor)
        (let ((i0 (head potential-governor))
              (v0 (head (bar-node (complement potential-governor)))))
          (if (or (eq '- (feature-value i0 'tense))
                  (eq '- (feature-value v0 'tense)))
              (fail))))
    potential-governor))

(defun smallest-maximal-projection-containing (phrase)
  (let ((parent (phrase-parent phrase)))
    (unless parent (fail))
    (if (level-2? parent) parent (smallest-maximal-projection-containing parent))))

(defun is-pronominal? (phrase)
  (eq '+ (phrase-pronominal? phrase)))

(defun is-anaphor? (phrase)
  (eq '+ (phrase-anaphor? phrase)))
```

B.7 Free Determination of Empty Categories

```
(defgenerator ANAPHOR? ((tree OPERATOR-ASSIGNMENT))
  (return-result (annotate-empty-categories-with-anaphor-feature tree)))

(defndm! annotate-empty-categories-with-anaphor-feature ((phrase) phrase)
  ;; Do not assign binding category features to ecs in Comp.
  (virtual-map-up-phrase-structure
   (phrase annotate-empty-categories-with-anaphor-feature)
   (when (np? phrase)
     (if (empty? phrase)
         (unless (phrase-anaphor? phrase)
           (set!-local (phrase-anaphor? phrase) (either '+ '-)))
         (if (word? phrase)
             (set!-local (phrase-anaphor? phrase) (feature-value phrase 'anaphor)))))))

(defgenerator PRONOMINAL? ((tree OPERATOR-ASSIGNMENT))
  (return-result (annotate-empty-categories-with-pronominal-feature tree)))
```

```
(defndm! annotate-empty-categories-with-pronominal-feature ((phrase) phrase)
  ;; Do not assign binding category features to ecs in Comp.
  (virtual-map-up-phrase-structure
   (phrase annotate-empty-categories-with-pronominal-feature)
   (when (np? phrase)
     (if (empty? phrase)
         (unless (phrase-pronominal? phrase)
           (set!-local (phrase-pronominal? phrase) (either '+ '-)))
         (if (word? phrase)
             (set!-local (phrase-pronominal? phrase) (feature-value phrase 'pronoun)))))))
```

B.8 Structural Determination of Empty Categories

```
(defgenerator ANAPHOR? ((tree OPERATOR-ASSIGNMENT) CASE-ASSIGNMENT)
  ;; Although a strictly deterministic theory like Structural
  ;; Determination should not need to use virtual trees, they are used
  ;; because when *pure-sd* is false structural determination is used
  ;; only to order the non-deterministic choices.
  (return-result (annotate-empty-categories-with-anaphor-feature tree)))

(defun annotate-empty-categories-with-anaphor-feature (phrase)
  ;; Do not assign binding category features to ecs in Comp.  Cannot memoize
  ;; here because properties depend on phrase's parents.
  (virtual-map-up-phrase-structure
   (phrase annotate-empty-categories-with-anaphor-feature)
   (when (np? phrase)
     (if (empty? phrase)
         (unless (c-spec? phrase)
           (let ((plus? (and (a-position? phrase)
                             (or (not (is-lexically-governed? phrase))
                                 (not (phrase-assigned-case phrase))))))
             (set!-local (phrase-anaphor? phrase)
                         (if *pure-sd*
                             (if plus? '+ '-)
                             (if plus?
                                 (either '+ '-)
                                 (either '- '+))))))
         (if (word? phrase)
             (set!-local (phrase-anaphor? phrase) (feature-value phrase 'anaphor)))))))

(defgenerator PRONOMINAL? ((tree OPERATOR-ASSIGNMENT))
  ;; Although a strictly deterministic theory like Structural
  ;; Determination should not need to use virtual trees, they are used
  ;; because when *pure-sd* is false structural determination is used
  ;; only to order the non-deterministic choices.
  (return-result (annotate-empty-categories-with-pronominal-feature tree)))
```

```
(defun annotate-empty-categories-with-pronominal-feature (phrase)
  ;; Do not assign binding category features to ecs in Comp.
  (virtual-map-up-phrase-structure
   (phrase annotate-empty-categories-with-pronominal-feature)
   (when (np? phrase)
     (if (empty? phrase)
         (unless (c-spec? phrase)
           (let ((plus? (not (is-lexically-governed? phrase))))
             (set!-local (phrase-pronominal? phrase)
                         (if *pure-sd*
                             (if plus? '+ '-)
                             (if plus?
                                 (either '+ '-)
                                 (either '- '+))))))
         (if (word? phrase)
             (set!-local (phrase-pronominal? phrase) (feature-value phrase 'pronoun)))))))
```

B.9 Case Theory

```
(defgenerator CASE-ASSIGNMENT ((tree PHRASE-STRUCTURE))
  (if (eq (phrase-category tree) 'n2) ;; Top-level NP
      (assign-case tree 'NOM))
  (return-result (annotate-phrase-with-case tree)))

(defm! annotate-phrase-with-case ((phrase) phrase)
  ;; Since case assignment is deterministic, we do not really need to use virtual
  ;; trees, but this is done for compatibility with a non-deterministic theory of
  ;; case assignment.
  (virtual-map-up-phrase-structure
   (phrase annotate-phrase-with-case)
   (when (eq phrase (core-phrase phrase))
     (cond ((p1? phrase)
            (when (and (complement phrase) (np? (complement phrase)))
              (assign-case (complement phrase) 'acc)))
           ((i1? phrase)
            (unless (eq '+ (feature-value (head phrase) 'passive))
              (let ((v1 (bar-node (complement phrase))))
                (when (and (v1? v1) (np? (complement v1)))
                  (assign-case (complement v1) 'acc)))))
           ((np? phrase)
            (let ((inherent-case (feature-value phrase 'inherent-case)))
              (when inherent-case (assign-case phrase inherent-case))))
           ((ip? phrase)
            (when (np? (spec-of phrase))
              (let ((i0 (head (bar-node phrase)))
                    (v0 (head (bar-node (complement (bar-node phrase))))))
                (unless (or (eq '- (feature-value i0 'tense))
                            (eq '- (feature-value v0 'tense)))
```

```
                 (assign-case (spec-of phrase) 'nom))))))))

(deffilter EMPTY-CASE-FILTER ((virtual-tree ANAPHOR?) CASE-ASSIGNMENT)
  ;; Filter that says +A ecs must not get case, -As must (if in A positions).
  (virtual-phrase-map
   virtual-tree (phrase :category category)
   (when (and (eq category 'n2) (empty? phrase) (a-position? phrase))
     (let ((ana (phrase-anaphor? phrase))
           (case (phrase-assigned-case phrase)))
       (when (or (and (eq '+ ana) case)
                 (and (eq '- ana) (not case)))
         ;; It is not the empty category itself that is responsible
         ;; for the failure, but whatever domain that should contain
         ;; a case assigner.  This is usually the parent phrase of
         ;; the empty category, though if ECM were handled we would
         ;; have to be more careful.  Reject up one level of the tree
         ;; from PHRASE.
         (virtual-reject 1))))))

(deffilter LEXICAL-CASE-FILTER ((virtual-tree CASE-ASSIGNMENT))
  ;; Filter that Lexical NPs must get case if in a-position.
  (virtual-phrase-map
   virtual-tree (phrase :category category)
   (when (and (eq category 'n2) (not (empty? phrase)) (a-position? phrase))
     (unless (phrase-assigned-case (core-phrase phrase))
       (unless (and (eq '+ (feature-value phrase 'wh))
                    (cp? (phrase-parent phrase)))
         ;; It is not the noun phrase itself that is responsible
         ;; for the failure, but whatever domain that should contain
         ;; a case assigner.  This is usually the parent phrase of
         ;; the noun phrase, though if ECM were handled we would
         ;; have to be more careful.  Reject up one level of the tree
         ;; from PHRASE.
         (virtual-reject 1))))))

(defparameter *case-compatibilities*
  '((nom nom-acc)
    (acc nom-acc)))

(defun assign-case (phrase case)
  (let ((core-phrase (core-phrase phrase)))
    (unless (check-case-compatibility (phrase-assigned-case core-phrase)
                                      (feature-value core-phrase 'required-case)
                                      case)
      (fail))
    (set!-local (phrase-assigned-case core-phrase) case)))

(defun check-case-compatibility (previously-assigned-case
                                 required-case
                                 assigned-case)
  (labels ((compatible? (c1 c2)
             (find-if #'(lambda (case-list) (and (member c1 case-list) (member c2 case-list)))
                      *case-compatibilities*)))
```

```
    (and (or (not previously-assigned-case)
             (compatible? previously-assigned-case assigned-case))
         (or (not required-case)
             (compatible? required-case assigned-case)))))
```

B.10 Theta Theory

```
(defgenerator THETA-ROLE-ASSIGNMENT ((tree PHRASE-STRUCTURE))
  ;; Assign thematic roles from tensed verbs to subject, verbs
  ;; to objects, and prepositions to their objects.  Rather than
  ;; storing exact theta roles in the lexicon, the shorthand
  ;; REFERENCE, SUBJECT and OBJECT symbols are used.
  (when (np? tree) ;; Top-level NP
    (set!-local (phrase-theta-role (core-phrase tree)) 'REFERENCE))
  (return-result (annotate-phrase-with-theta tree)))

(defm! annotate-phrase-with-theta ((phrase) phrase)
  (let ((daughters (phrase-daughters phrase))
        (category (phrase-category phrase)))
    (mapc-nd #'annotate-phrase-with-theta daughters)
    (when (eq (core-phrase phrase) phrase)
      (case category
        ((v1 p1)
         ;; Assign roles from verbs and prepositions to their complements.
         (let ((x (complement phrase)))
           (cond ((null x)
                  (if (or (not (feature-value (head phrase) 'theta))
                          (member (feature-value (head phrase) 'theta) '(none)))
                      t
                      (fail)))
                 ((np? x)
                  (if (or (not (feature-value (head phrase) 'theta))
                          (member (feature-value (head phrase) 'theta)
                                  '(np np-cp np-none)))
                      (set!-local (phrase-theta-role (core-phrase x)) 'OBJECT)
                      (fail)))
                 ((cp? x)
                  (if (or (not (feature-value (head phrase) 'theta))
                          (member (feature-value (head phrase) 'theta)
                                  '(cp np-cp subjectless-cp)))
                      t
                      (fail))))))
        (i2
         ;; Assign roles from tensed verbs to their subjects.
         (let ((i0 (head (bar-node phrase)))
               (v0 (head (bar-node (complement (bar-node phrase))))))
           (unless (or (and (i0? i0) (eq '+ (feature-value i0 'passive)))
                       (eq 'subjectless-cp (feature-value v0 'theta)))
             (set!-local (phrase-theta-role (core-phrase (spec-of phrase))) 'SUBJECT))))))
```

```
    phrase))

(deffilter THETA-CRITERION ((chains CHAIN-FORMATION) THETA-ROLE-ASSIGNMENT)
  ;; Enforce criterion that every chain receive exactly one thematic role.
  (unless (every #'(lambda (chain)
                     (= 1 (count-if #'(lambda (phrase)
                                        (phrase-theta-role (core-phrase phrase)))
                                    chain)))
                 chains)
    (reject)))
```

B.11 Basic Chain Formation

```
;;; Chain Formation.  The process of chain formation automatically handles
;;; subjacency, though this could be removed.  The particular algorithm borrows
;;; loosely from that found in Sandiway Fong's PhD thesis.

(defstruct chain-state
  (partial-chains nil :type list)
  (free-phrases nil :type list)
  (completed-chains nil :type list)
  (marked-things nil :type list))

(defgenerator CHAIN-FORMATION ((tree PHRASE-STRUCTURE))
  ;; The result of CHAIN-FORMATION is a set (list) of chains, each a list
  ;; of noun phrases linked by movement.
  (let ((result (do-chain-formation tree)))
    (unless (or (chain-state-partial-chains result)
                (chain-state-free-phrases result))
      (return-result (chain-state-completed-chains result)))))

(defun get-chain-index (phrase chains)
  (position (core-phrase phrase) chains :test #'member))

(defndm do-chain-formation ((phrase) phrase)
  (case (length (phrase-daughters phrase))
    (0 (if (np? phrase)
           (if (empty? phrase)
               (either (make-chain-state :partial-chains (list (list phrase)))
                       (make-chain-state :free-phrases (list phrase))
                       (make-chain-state :completed-chains (list (list phrase))))
               (either (make-chain-state :free-phrases (list phrase))
                       (make-chain-state :completed-chains (list (list phrase)))))
           (make-chain-state)))
    (1 (subjacency-mark
        (if (and (np? phrase)
                 (eq (core-phrase phrase) phrase))
            (merge-higher-np phrase (do-chain-formation (first (phrase-daughters phrase))))
            (do-chain-formation (first (phrase-daughters phrase))))
```

```
        phrase))
    (otherwise
     (subjacency-mark
      (if (and (np? phrase)
               (eq phrase (core-phrase phrase)))
          (merge-higher-np phrase (do-multiple-chain-formation (phrase-daughters phrase)))
          (do-multiple-chain-formation (phrase-daughters phrase)))
      phrase))))

(defun subjacency-mark (result potential-bounding-phrase)
  (if (member (phrase-category potential-bounding-phrase) *bounding-nodes*)
      (if (or (intersection (mapcar #'car (chain-state-partial-chains result))
                            (chain-state-marked-things result))
              (intersection (chain-state-free-phrases result)
                            (chain-state-marked-things result)))
          (fail);; Already marked, now passing through second bounding node.
          (make-chain-state :partial-chains (chain-state-partial-chains result)
                            :free-phrases (chain-state-free-phrases result)
                            :completed-chains (chain-state-completed-chains result)
                            :marked-things
                            (set-difference
                             (append (chain-state-free-phrases result)
                                     (mapcar #'car (chain-state-partial-chains result)))
                             (all-parts potential-bounding-phrase))))
      result))

(defun do-multiple-chain-formation (list-of-daughters)
  (if (null (cdr list-of-daughters))
      (do-chain-formation (first list-of-daughters))
      (let ((result1 (do-chain-formation (first list-of-daughters)))
            (result2 (do-multiple-chain-formation (rest list-of-daughters))))
        ;; Do we want to rule out cases with Free Phrases (because of
        ;; the c-command requirement)?  Let's go for it!
        (let ((chain (merge-chain-formations result1 result2)))
          (if (chain-state-free-phrases chain)
              (fail)
              chain)))))

(defun merge-chain-formations (result1 result2)
  (let ((completed-chains (append (chain-state-completed-chains result1)
                                  (chain-state-completed-chains result2))))
    (let ((c1 (submerge (chain-state-partial-chains result1)
                        (chain-state-free-phrases result2)
                        completed-chains
                        (chain-state-marked-things result2)
                        (chain-state-marked-things result1))))
      (let ((c2 (submerge (chain-state-partial-chains result2)
                          (chain-state-free-phrases result1)
                          (third c1)
                          (chain-state-marked-things result1)
                          (chain-state-marked-things result2))))
        (make-chain-state :partial-chains (append (first c1) (first c2))
                          :free-phrases (append (second c1) (second c2))
```

```
                          :completed-chains (third c2)
                          :marked-things (append (chain-state-marked-things result1)
                                                 (chain-state-marked-things result2)))))))

(defun merge-higher-np (phrase result)
  (either (make-chain-state :partial-chains (chain-state-partial-chains result)
                            :free-phrases (cons phrase (chain-state-free-phrases result))
                            :completed-chains (chain-state-completed-chains result)
                            :marked-things (chain-state-marked-things result))
          (make-chain-state :partial-chains (chain-state-partial-chains result)
                            :free-phrases (chain-state-free-phrases result)
                            :completed-chains (cons (list phrase)
                                                    (chain-state-completed-chains result))
                            :marked-things (chain-state-marked-things result))))

(defun submerge (partial-chains free-phrases completed-chains marked-free-phrases
                                marked-partial-chains)
  (either (list partial-chains free-phrases completed-chains)
          (let ((free-phrase (a-member-of free-phrases))
                (partial-chain (a-member-of partial-chains)))
            (if (and (member free-phrase marked-free-phrases)
                     (member (car partial-chain) marked-partial-chains))
                (fail) ;; Subjacency
                (let ((new-chain (cons free-phrase partial-chain))
                      (old-free-phrases (remove free-phrase free-phrases))
                      (old-partial-chains (remove partial-chain partial-chains)))
                  (either (submerge old-partial-chains
                                    old-free-phrases
                                    (cons new-chain completed-chains)
                                    marked-free-phrases
                                    marked-partial-chains)
                          (if (empty? free-phrase)
                              (submerge (cons new-chain old-partial-chains)
                                        old-free-phrases
                                        completed-chains
                                        marked-free-phrases
                                        marked-partial-chains)
                              (fail))))))))
```

B.12 Correa Chain Formation

```
;;; Correa's deterministic Chain Formation.  See "Empty Categories, Chains, and Parsing"
;;; in R.C.Berwick et al, Principle Based Parsing (1991).

(defgenerator CHAIN-FORMATION ((tree PHRASE-STRUCTURE) THETA-ROLE-ASSIGNMENT ANAPHOR? PRONOMINAL?)
  (let ((result (get-chains tree)))
    (unless (or (first result) (second result))
      (return-result (third result)))))
```

```lisp
(defun get-chain-index (phrase chains)
  (position (core-phrase phrase) chains :test #'member))

(defun get-chains (phrase)
  ;; Return: (PARTIAL-A-CHAIN PARTIAL-ABAR-CHAIN COMPLETED-CHAINS)
  (let ((result
         (cond ((and (ip? phrase) (core-phrase? phrase) (find-if #'np? (phrase-daughters phrase)))
                (let* ((i1 (bar-node phrase))
                       (i1-chains (get-chains i1))
                       (np (spec i1))
                       (np-chains (get-chains np)))
                  (if (and (not (and (variable? np) (second i1-chains)))
                           (not (and (first i1-chains) (phrase-theta-role (core-phrase np))))
                           (or (first i1-chains) (phrase-theta-role (core-phrase np))))
                      (list (if (np-trace? np) (cons (core-phrase np) (first i1-chains)) nil)
                            (if (variable? np)
                                (cons (core-phrase np) (first i1-chains))
                                (second i1-chains))
                            (if (or (np-trace? np) (variable? np))
                                (third i1-chains)
                                (cons (cons (core-phrase np) (first i1-chains))
                                      (append (third i1-chains) (third np-chains)))))
                      (fail))))
               ((cp? phrase)
                (if (np? (spec (bar-node phrase)))
                    (let* ((np (spec (bar-node phrase)))
                           (np-chains (get-chains np))
                           (c1 (bar-node phrase))
                           (c1-chains (get-chains c1)))
                      (if (and (second c1-chains)
                               (or (variable? np) (operator? np)))
                          (list (first c1-chains)
                                (if (variable? np) (cons (core-phrase np) (second c1-chains)) nil)
                                (if (variable? np)
                                    (third c1-chains)
                                    (cons (cons (core-phrase np) (second c1-chains))
                                          (append (third np-chains) (third c1-chains)))))
                          (fail)))
                    (let* ((c1 (bar-node phrase))
                           (c1-chains (get-chains c1)))
                      (if (not (second c1-chains))
                          c1-chains
                          (fail)))))
               ((find-if #'np? (phrase-daughters phrase))
                (let* ((o (find-if-not #'np? (phrase-daughters phrase)))
                       (o-chains (get-chains o))
                       (np (find o (phrase-daughters phrase) :test-not #'eq))
                       (np-chains (get-chains np)))
                  (if (and (not (and (variable? np) (second o-chains)))
                           (not (and (np-trace? np) (first o-chains))))
                      (list (if (np-trace? np) (list (core-phrase np)) (first o-chains))
                            (if (variable? np) (list (core-phrase np)) (second o-chains))
                            (if (or (np-trace? np) (variable? np))
```

```
                                (third o-chains)
                                (cons (list (core-phrase np))
                                      (append (third o-chains) (third np-chains)))))
                      (fail))))
               ((= 2 (length (phrase-daughters phrase)))
                (let ((o1 (get-chains (first (phrase-daughters phrase))))
                      (o2 (get-chains (second (phrase-daughters phrase)))))
                  (if (and (not (and (first o1) (first o2)))
                           (not (and (second o1) (second o2))))
                      (list (or (first o1) (first o2))
                            (or (second o1) (second o2))
                            (append (third o1) (third o2)))
                      (fail))))
               ((= 1 (length (phrase-daughters phrase)))
                (get-chains (first (phrase-daughters phrase))))
               (t (list nil nil nil)))))
    result))
```

B.13 Free Indexing

```
(defgenerator FREE-INDEXING ((list-of-chains CHAIN-FORMATION))
  ;; Return a list of chain sets, that looks like
  ;;
  ;; ((REFERENTIAL-INDEX-1 CHAIN-1-1 CHAIN-1-2 ...)
  ;;  (REFERENTIAL-INDEX-2 CHAIN-2-1 ...)
  ;;  )
  ;;
  ;; where each referential index is an integer and each chain is a list
  ;; of phrases.  The chains paired with a referential index all co-refer.
  ;;
  (when list-of-chains
    (return-result
     (loop for integer-set in (freely-index (length list-of-chains))
           for referential-index from 1
           collect (cons referential-index
                         (mapcar #'(lambda (i) (nth i list-of-chains))
                                 integer-set))))))

(defndmemo freely-index ((n) () eql)
  ;; Nondeterministically return a partition on the integers 0 through
  ;; N-1.  Each partition is a list of integers.  For example, the
  ;; five values returned by (freely-index 3) are
  ;;
  ;;    ((2) (1) (0))
  ;;    ((2 1) (0))
  ;;    ((2 0) (1))
  ;;    ((2) (1 0))
  ;;    ((2 1 0))
  ;;
```

```
  (unless (zerop n)
    (let ((indexing (freely-index (1- n))))
      (either
       ;; Put integer into its own partition.
       `((,(1- n)) ,@indexing)
       ;; Put integer into some existing partition.
       (let ((index-set-to-merge-with (a-member-of indexing)))
         `((,(1- n) ,@index-set-to-merge-with)
           ,@(remove index-set-to-merge-with indexing)))))))

(defun get-index (phrase index-list)
  (car (find (core-phrase phrase) index-list
             :key #'cdr
             :test #'(lambda (p chain-list)
                       (some #'(lambda (chain) (member p chain)) chain-list)))))

(defun get-phrases-with-index (index index-list)
  (apply #'append (mapcar #'all-parts (apply #'append (cdr (assoc index index-list))))))
```

B.14 Functional Determination Filter

```
(deffilter FUNCTIONAL-DETERMINATION ((chains CHAIN-FORMATION)
                                     (tree PHRASE-STRUCTURE)
                                     (indices FREE-INDEXING)
                                     ANAPHOR?
                                     PRONOMINAL?)
  ;; Enforce the effects of Functional Determination.
  ;; For a description of the theory of functional determination,
  ;; see Chomsky "Some Concepts and Consequences of the Theory
  ;; of Govenment and Binding", pg. 34.
  (map-up-phrase-structure
   tree (phrase)
   (when (and (np? phrase) (empty? phrase) (not (operator? phrase)))
     (when (a-position? phrase)
       (let ((lb (local-binder phrase indices tree)))
         (if (and lb (a-bar-position? lb))
             (unless (variable? phrase)
               (reject))
             (when (variable? phrase)
               (reject)))
         (unless (variable? phrase)
           (unless (is-anaphor? phrase)
             (reject))
           (when (or (null lb)
                     (and (a-position? lb)
                          (not (eql (get-chain-index lb chains)
                                    (get-chain-index phrase chains)))))
             (unless (is-pronominal? phrase)
               (reject))))))))))
```

B.15 Subjacency, I-within-I

```
(deffilter I-WITHIN-I ((indices FREE-INDEXING))
  (dolist (index-set indices)
    (let ((all-daughter-phrases nil)
          (all-indexed-phrases nil))
      (dolist (chain (cdr index-set))
        (dolist (phrase chain)
          (if (member phrase all-daughter-phrases)
              (reject)
              (labels ((add-phrase (p)
                         (when (member p all-indexed-phrases)
                           (reject))
                         (unless (member p all-daughter-phrases)
                           (push p all-daughter-phrases)
                           (mapc #'add-phrase (phrase-daughters p)))))
                (add-phrase phrase)
                (push phrase all-indexed-phrases))))))))

(defparameter *bounding-nodes* '(i2 n2))

(deffilter subjacency ((chains CHAIN-FORMATION))
  ;; Enforce the subjacency requirement on chains.
  (dolist (chain chains)
    (mapl #'(lambda (chain-part)
              (let ((phrase1 (first chain-part))
                    (phrase2 (second chain-part)))
                (when (and phrase1 phrase2)
                  (unless (subjacent? phrase1 phrase2)
                    (reject)))))
          chain)))

(defun subjacent? (phrase1 phrase2)
  (let ((pp1 (all-phrase-parents phrase1))
        (pp2 (all-phrase-parents phrase2)))
    (let ((s1 (set-difference pp1 pp2))
          (s2 (set-difference pp2 pp1)))
      (> 2 (max (count-if #'(lambda (phrase)
                              (member (phrase-category phrase) *bounding-nodes*)) s1)
                (count-if #'(lambda (phrase)
                              (member (phrase-category phrase) *bounding-nodes*)) s2)
                )))))
```

B.16 License Chains

```
(defoption '*little-pro* nil "Little pro" :generator)

(deffilter LICENSE-CHAINS ((chains CHAIN-FORMATION) ANAPHOR? PRONOMINAL?)
```

```
  ;; Enforce various conditions on chains and empty categories.
  (dolist (chain chains)
    ;; At least one trace per chain.
    (when (> (length chain) 1)
      (unless (some #'(lambda (v) (and (empty? v) (eq '- (phrase-pronominal? v))))
                    chain)
        (reject)))
    ;; Operators must bind variables.
    (if (and (some #'operator? chain)
             (not (some #'variable? chain)))
        (reject))
    ;; If there is an operator, it must be at head of chain.
    (if (or (> (count-if #'operator? chain) 1)
            (some #'operator? (rest chain)))
        (reject))
    ;; Wh-trace in Comp must be licensed by an operator.
    (if (some #'(lambda (c) (and (variable? c) (c-spec? c))) chain)
        (unless (some #'operator? chain)
          (reject)))
    ;; English does not have -anaphor, +pronominal empty categories.
    (unless *little-pro*
      (if (some #'(lambda (v)
                    (and (empty? v)
                         (eq '- (phrase-anaphor? v))
                         (eq '+ (phrase-pronominal? v))))
                chain)
          (reject)))
    (let ((head (first chain)))
      ;; Only lexical items, operators, variables, PRO and pro may head a chain.
      (unless (or (not (empty? head)) (eq (phrase-pronominal? head) '+) (variable? head)
                  (operator? head))
        (reject)))))
```

B.17 Operator Assignment

```
(defgenerator OPERATOR-ASSIGNMENT ((phrase PHRASE-STRUCTURE))
  ;; Make empty categories in operator position be operators by setting their
  ;; anaphoric and pronominal values to be "o" instead of either + or -.
  (return-result (operator-assigner phrase)))

(defun operator-assigner (phrase)
  (dolist (daughter (phrase-daughters phrase))
    (operator-assigner daughter))
  (when (and (c-spec? phrase) (empty? phrase))
    (let ((value (if (np? (spec (phrase-parent phrase))) 'o '-)))
      (set!-local (phrase-anaphor? phrase) value)
      (set!-local (phrase-pronominal? phrase) value)))
  phrase)
```

B.18 Other Modules

```
(deffilter coindex-operator ((tree PHRASE-STRUCTURE) (indices FREE-INDEXING))
  (map-up-phrase-structure
   tree (phrase :daughters daughters)
   (when (and (np? phrase) (= 2 (length daughters))
              (find-if #'np? daughters) (find-if #'cp? daughters))
     ;; PHRASE is a NP with an CP daughter that contains an operator.
     (let ((operator (spec-of (find-if #'cp? daughters))))
       (unless (and (np? operator)
                    (= (get-index (find-if #'np? daughters) indices)
                       (get-index operator indices)))
         (reject))))))

(defoption '*lexical-np-in-comp* nil "Lexical NP in Comp")

(defun comp-checker-special (phrase1 phrase2 category)
  ;; Return T if comp is OK.
  (let ((phrases (list phrase1 phrase2)))
    (not (cond ((eq category 'c2)
                (let ((np (find-if #'np? phrases))
                      (c1 (find-if #'c1? phrases)))
                  (and np c1
                       (or
                        ;; Prevent lexical NP and lexical C in comp.
                        (and (not (empty? np)) (c0? (head c1)))
                        ;; Lexical NP must be +wh.
                        (not (or (empty? np) *lexical-np-in-comp* (eq '+ (feature-value np 'wh))))
                        ;; Prevent I unless accompanied by lexical NP in comp.
                        (and (i0? (head c1)) (empty? np))))))
               ((eq category 'n2)
                ;; Prevent name or pronoun in relative clause.
                (let ((np (core-phrase (find-if #'np? phrases)))
                      (c2 (find-if #'cp? phrases)))
                  (and np c2
                       (or (feature-value np 'r-expression)
                           (feature-value np 'pronoun)
                           (feature-value np 'anaphor)))))))))

(defun comp-checker-special-2 (phrase1 phrase2 category)
  ;; Return T if OK.
  (let ((phrases (list phrase1 phrase2)))
    (not (cond ((eq category 'c2)
                (let ((np (find-if #'np? phrases))
                      (c1 (find-if #'c1? phrases)))
                  (and np c1
                       (or
                        ;; Prevent lexical NP and lexical C in comp.
                        (and (not (empty? np)) (c0? (head c1)))
                        ;; Lexical NP must be +wh.
                        (not (or (empty? np) *lexical-np-in-comp* (eq '+ (feature-value np 'wh))))
                        ;; Prevent I unless accompanied by lexical NP in comp.
```

```
                         (and (i0? (head c1)) (empty? np))))))
               ((eq category 'n2)
                ;; Prevent name or pronoun in relative clause.
                (let ((np (core-phrase (find-if #'np? phrases)))
                      (c2 (find-if #'cp? phrases)))
                  (and np c2
                       (or (feature-value np 'r-expression)
                           (feature-value np 'pronoun)
                           (feature-value np 'anaphor)))))))))

(defun comp-checker-special-1 (phrase category)
  ;; Return T if OK.
  (not (cond ((eq category 'c2)
              (let ((c1 phrase))
                (i0? (head c1))))
             (t nil))))
```

B.19 General Utility Functions

```
(defparameter *heads* '(a0 n0 v0 d0 c0 p0 i0 p0))
(defparameter *1cats* '(a1 n1 v1 d1 c1 p1 i1))
(defparameter *2cats* '(a2 n2 v2 d2 c2 p2 i2))

(defun np? (phrase) (and phrase (eq (phrase-category phrase) 'n2)))
(defun cp? (phrase) (and phrase (eq (phrase-category phrase) 'c2)))
(defun c1? (phrase) (and phrase (eq (phrase-category phrase) 'c1)))
(defun c0? (phrase) (and phrase (eq (phrase-category phrase) 'c0)))
(defun ip? (phrase) (and phrase (eq (phrase-category phrase) 'i2)))
(defun i1? (phrase) (and phrase (eq (phrase-category phrase) 'i1)))
(defun i0? (phrase) (and phrase (eq (phrase-category phrase) 'i0)))
(defun vp? (phrase) (and phrase (eq (phrase-category phrase) 'v2)))
(defun v1? (phrase) (and phrase (eq (phrase-category phrase) 'v1)))
(defun p1? (phrase) (and phrase (eq (phrase-category phrase) 'p1)))

(defun level-0? (phrase) (member (phrase-category phrase) *heads*))
(defun level-1? (phrase) (member (phrase-category phrase) *1cats*))
(defun level-2? (phrase) (member (phrase-category phrase) *2cats*))
(defun complement? (phrase) (eq phrase (complement phrase)))
(defun head? (phrase) (eq phrase (head phrase)))
(defun spec? (phrase) (eq phrase (spec phrase)))

(defun empty? (phrase)
  ;; Is PHRASE lexically realized?
  (and (null (phrase-daughters phrase))
       (not (word? phrase))))

(defun word? (phrase)
  ;; Is PHRASE a lexically realized word?
  (feature-value phrase 'word))
```

```
(defun c-spec? (phrase)
  ;; Is PHRASE in Spec of C?
  (and (cp? (phrase-parent phrase)) (spec? phrase)))

(defun operator? (phrase)
  ;; Is PHRASE an operator?
  (and (c-spec? phrase)
       (or (not (empty? phrase))
           (and (eq 'o (phrase-anaphor? phrase))
                (eq 'o (phrase-pronominal? phrase))))))

(defun variable? (phrase)
  ;; Is PHRASE a variable (-pro, -ana empty category)?
  (and (np? phrase)
       (empty? phrase)
       (eq '- (phrase-anaphor? phrase))
       (eq '- (phrase-pronominal? phrase))))

(defun np-trace? (phrase)
  ;; Is PHRASE an NP-trace (-pro, +ana empty category)
  (and (np? phrase)
       (empty? phrase)
       (eq '+ (phrase-anaphor? phrase))
       (eq '- (phrase-pronominal? phrase))))

(defun r-expression? (phrase)
  ;; Is PHRASE an R-expression?
  (and (not (empty? phrase))
       (eq '+ (feature-value phrase 'r-expression))))

(defun is-lexically-governed? (phrase)
  ;; Nothing fancy- no barriers.
  (let ((pp (phrase-parent phrase)))
    (or (and (v1? pp) (complement? phrase))
        (and (p1? pp) (complement? phrase))
        (and (ip? pp) (spec? phrase)
             (not (eq '- (feature-value (head (bar-node phrase)) 'tense)))))))

(defun core-phrase (phrase)
  ;; Gets at core phrase for adjunction phenomena.  For instance, when applied
  ;; to [NP [NP] [PP]] returns the inner NP.
  (when phrase
    (let ((potential-core (find (phrase-category phrase) (phrase-daughters phrase)
                                :key #'phrase-category)))
      (if potential-core
          (core-phrase potential-core)
          phrase))))

(defun core-phrase? (phrase)
  ;; Is PHRASE a core, or is it the result of some adjunction?
  (not (adjunction? phrase)))

(defun adjunction? (phrase)
```

```
  ;; Is PHRASE the result of adjunction?
  (when phrase
    (find (phrase-category phrase) (phrase-daughters phrase) :key #'phrase-category)))

(defun complement (phrase)
  ;; Given the standard x-bar structure
  ;;   [Y ... [X2 SPEC [X1 HEAD COMP]]], for input
  ;;
  ;;     X1 return COMP
  ;;     X2 return (complement Y)
  ;;     HEAD return COMP
  ;;     COMP return COMP
  ;;     SPEC return (complement Y)
  ;;
  (when phrase
    (cond ((level-1? phrase)
           ;; Check that it's not adjunction
           (if (find-if #'level-0? (phrase-daughters (core-phrase phrase)))
               (find-if #'level-2? (phrase-daughters (core-phrase phrase)))))
          ((level-0? phrase)
           (complement (phrase-parent phrase)))
          ((level-2? phrase) (complement (phrase-parent phrase))))))

(defun spec (phrase)
  ;; Given the standard x-bar structure
  ;;   [X2 SPEC [X1 HEAD COMP]], for input
  ;;
  ;;     X1 return SPEC
  ;;     SPEC return SPEC
  ;;     otherwise return nil.
  ;;
  (when phrase
    (let ((pp (phrase-parent phrase)))
      (and pp (level-2? pp) (find-if-not #'level-1? (phrase-daughters pp))))))

(defun spec-of (phrase)
  ;; Given the standard x-bar structure
  ;;   [X2 SPEC [X1 HEAD COMP]], for input
  ;;
  ;;     X2 return SPEC
  ;;     otherwise return nil.
  ;;
  (when phrase
    (and (level-2? phrase) (find-if-not #'level-1? (phrase-daughters phrase)))))

(defun head (phrase)
  ;; Given the standard x-bar structure
  ;;   [Y [X2 SPEC [X1 HEAD COMP]]], for input
  ;;
  ;;     X1 return HEAD
  ;;     HEAD return HEAD
  ;;     COMP return HEAD
  ;;     otherwise return nil.
```

```
  ;;
  (when phrase
    (cond ((level-1? phrase)
           (find-if #'level-0? (phrase-daughters (core-phrase phrase))))
          ((phrase-parent phrase)
           (let ((pp (phrase-parent phrase)))
             (when (level-1? pp)
               (head pp)))))))

(defun bar-node (phrase)
  ;; Given the standard x-bar structure
  ;;   [Y [X2 SPEC [X1 HEAD COMP]]], for input
  ;;
  ;;     X2 return X1
  ;;     COMP return (bar-node COMP)
  ;;     SPEC return X1
  ;;     otherwise return nil.
  ;;
  (when phrase
    (or (and (level-2? phrase)
             (find-if #'level-1? (phrase-daughters phrase)))
        (let ((pp (phrase-parent phrase)))
          (and pp (level-2? pp) (core-phrase (find-if #'level-1? (phrase-daughters pp))))))))

(defun all-parts (phrase)
  ;; Return a list of all phrases that have resulted from adjunction to
  ;; the core phrase of PHRASE.
  (labels ((all-parts-1 (phrase)
             (cons phrase
                   (if (and (phrase-parent phrase)
                            (eq (phrase-category (phrase-parent phrase))
                                (phrase-category phrase)))
                       (all-parts-1 (phrase-parent phrase))))))
    (all-parts-1 (core-phrase phrase))))

(defun top-part (phrase)
  ;; Return the largest phrase that has resulted from adjunction to PHRASE.
  (if (and (phrase-parent phrase)
           (eq (phrase-category (phrase-parent phrase))
               (phrase-category phrase)))
      (top-part (phrase-parent phrase))
      phrase))

(defun a-position? (pphrase)
  ;; Is PPHRASE in an argument position?
  ;; This is defined as: Subject, Object, or Object of Preposition (for now).
  (let ((phrase (top-part pphrase)))
    (let ((pp (phrase-parent phrase)))
      (and pp (eq pp (core-phrase pp))
           (or (and (ip? pp) (spec? phrase))
               (and (v1? pp) (complement? phrase))
               (and (p1? pp) (complement? phrase)))))))
```

```
(defun a-bar-position? (phrase)
  ;; Is PHRASE in a non-argument position?
  (not (a-position? phrase)))

(defun all-phrase-parents (phrase)
  (labels ((all-phrase-parents-1 (phrase)
             (unless (null (phrase-parent phrase))
               (cons (phrase-parent phrase) (all-phrase-parents-1 (phrase-parent phrase))))))
    (all-phrase-parents-1 (top-part phrase))))

(defun feature-value (phrase feature)
  (cdr (assoc feature (phrase-inherent-features phrase))))
```

B.20 Tree Structure Walkers

```
;;;
;;; Failures can not always be attributed directly to nodes in a
;;; phrase structure tree, but only to nodes in combination with
;;; features assigned by generators.  For instance, in the structure
;;;
;;;   Who did you [VP see [NP +anaphor]]
;;;
;;; the empty category [NP +anaphor] illegally receives case.  It
;;; would be nice to use mstructures to fail all trees containing the
;;; VP, but the blame really lies with the combination of the case
;;; assigning verb AND the assignment of the +anaphor feature to the
;;; empty NP.
;;;
;;; The following facilities provide functions that can be used when
;;; performing destructive operations like assignment of anaphoric
;;; properties to build tree-like mstructures, so that when such a
;;; failure occurs there is a specific node that can be rejected,
;;; potentially saving a great deal of effort if other paths also
;;; contain the same node.

(defmstructure (virtual-phrase :allow-dependencies t)
    ((real-phrase :initarg :real-phrase :accessor virt-phr-real-phrase :type phrase)
     (virtual-phrase-daughters :initarg :daughters :accessor virt-phr-daughters
                               :type (list virtual-phrase))
     (value :initarg :value :accessor virt-phr-value))
  ()
  )

(defmacro virtual-map-up-phrase-structure ((phrase function) &rest body)
  ;; Create "virtual" phrase structure, as described above, so that
  ;; "reject" can kill many threads at once.  The code BODY is
  ;; executed to non-deterministically produce values, potentially
  ;; executing side-effects.  The code is presumed to apply to the node PHRASE
  ;; and the function FUNCTION is then executed on the daughters of PHRASE to
  ;; produce more values.  For example,
```

```
  ;;
  ;; (defndm! assign-case ((phrase) phrase)
  ;;   (virtual-map-up-phrase-structures
  ;;     (phrase assign-case)
  ;;       (when ...
  ;;         (set!-local (phrase-case phrase) (either :accusative :oblique)))))
  ;;
  (let ((phrase-symbol (gensym "phrase-"))
        (daughters-symbol (gensym "daughters-")))
    `(let ((,phrase-symbol ,phrase))
       (let ((,daughters-symbol
              (case (length (phrase-daughters ,phrase-symbol))
                (0 nil)
                (1 (list (,function (first (phrase-daughters ,phrase-symbol)))))
                (2 (list (,function (first (phrase-daughters ,phrase-symbol)))
                         (,function (second (phrase-daughters ,phrase-symbol))))))))
         (let ((value (progn ,@body)))
           (let ((virtual-phrase
                  (make-instance 'virtual-phrase
                                 :daughters ,daughters-symbol
                                 :real-phrase ,phrase-symbol :value value)))
             (dolist (daughter ,daughters-symbol)
               (is-part-of daughter virtual-phrase))
             virtual-phrase))))))

(defmacro virtual-phrase-map
    (virtual-phrase (phrase &key category daughters virtual) &rest body)
  ;; This macro uses the same syntax as MAP-UP-PHRASE-STRUCTURE (see
  ;; below).  It applies to virtual trees produced with
  ;; VIRTUAL-MAP-UP-PHRASE-STRUCTURE, rooted with the virtual phrase
  ;; VIRTUAL-PHRASE.  If VIRTUAL is provided, BODY is executed with
  ;; the symbol VIRTUAL bound to the current virtual phrase, which can
  ;; be rejected.  Alternatively, VIRTUAL-REJECT can be used (see
  ;; below).  For example,
  ;;
  ;; (deffilter case-filter ((virtual-tree CASE-ASSIGNMENT))
  ;;   (virtual-phrase-map virtual-tree (phrase)
  ;;     (when (case-not-assigned? phrase)
  ;;       ;; Reject parent of PHRASE.
  ;;       (virtual-reject 1))))
  ;;
  (let ((virt-phr-symbol (gensym "virt-phr-")))
    `(virtual-map-over-phrase-structure-fn
      #'(lambda (,virt-phr-symbol list-of-previous-virtual-phrases)
          list-of-previous-virtual-phrases
          (let ((,phrase (virt-phr-real-phrase ,virt-phr-symbol))
                ,@(if category
                      `((,category (phrase-category
                                    (virt-phr-real-phrase ,virt-phr-symbol)))))
                ,@(if daughters
                      `((,daughters (phrase-daughters
                                     (virt-phr-real-phrase ,virt-phr-symbol)))))
                ,@(if virtual `((,virtual ,virt-phr-symbol))))
```

```
          ,@body))
      ,virtual-phrase nil)))

(defmacro virtual-reject (&optional (number 0))
  ;; Reject current "virtual" phrase.  See above.  If NUMBER is provided then
  ;; the nth parent is rejected.
  `(reject (or (nth ,number list-of-previous-virtual-phrases)
            (first list-of-previous-virtual-phrases))))

(defun virtual-map-over-phrase-structure-fn (function virtual-phrase previous-list)
  (let ((new-list (cons virtual-phrase previous-list)))
    (dolist (daughter (virt-phr-daughters virtual-phrase))
      (virtual-map-over-phrase-structure-fn function daughter new-list))
    (funcall function virtual-phrase new-list)))

;;;
;;; Macro Facilities for mapping over phrase structure trees.
;;;

(defmacro map-up-phrase-structure (top-phrase (phrase &key category daughters)
                                               &rest body)
  ;; Map over every node in the phrase structure tree with root
  ;; TOP-PHRASE, from the bottom of the tree up, executing for each
  ;; node the deterministic code BODY in an environment with the
  ;; symbol PHRASE bound to the current NODE, and the symbol CATEGORY
  ;; bound to that phrase's category, and the symbol DAUGHTERS bound
  ;; to the list of that phrase's daughter nodes.
  `(map-up-phrase-structure-fn
    #'(lambda (,phrase)
        (let (,@(if category `((,category (phrase-category ,phrase))))
                ,@(if daughters `((,daughters (phrase-daughters ,phrase)))))
          ,@body))
    ,top-phrase))

(defmacro map-up-phrase-structure-nd (top-phrase (phrase &key category daughters)
                                               &rest body)
  ;; Map over every node in the phrase structure tree with root
  ;; TOP-PHRASE, from the bottom of the tree up, executing for each
  ;; node the non-deterministic code BODY in an environment with the
  ;; symbol PHRASE bound to the current NODE, and the symbol CATEGORY
  ;; bound to that phrase's category, and the symbol DAUGHTERS bound
  ;; to the list of that phrase's daughter nodes.
  `(map-up-phrase-structure-fn-nd
    #'(lambda (,phrase)
        (let (,@(if category `((,category (phrase-category ,phrase))))
                ,@(if daughters `((,daughters (phrase-daughters ,phrase)))))
          ,@body))
    ,top-phrase))

(defmacro map-down-phrase-structure (top-phrase (phrase &key category daughters)
                                               &rest body)
  ;; Map over every node in the phrase structure tree with root
  ;; TOP-PHRASE, from the top of the tree down, executing for each
```

```
  ;; node the deterministic code BODY in an environment with the
  ;; symbol PHRASE bound to the current NODE, and the symbol CATEGORY
  ;; bound to that phrase's category, and the symbol DAUGHTERS bound
  ;; to the list of that phrase's daughter nodes.
  `(map-down-phrase-structure-fn
    #'(lambda (,phrase)
        (let (,@(if category `((,category (phrase-category ,phrase))))
                ,@(if daughters `((,daughters (phrase-daughters ,phrase)))))
          ,@body))
    ,top-phrase))

(defmacro map-down-phrase-structure-nd (top-phrase (phrase &key category daughters)
                                                   &rest body)
  ;; Map over every node in the phrase structure tree with root
  ;; TOP-PHRASE, from the top of the tree down, executing for each
  ;; node the non-deterministic code BODY in an environment with the
  ;; symbol PHRASE bound to the current NODE, and the symbol CATEGORY
  ;; bound to that phrase's category, and the symbol DAUGHTERS bound
  ;; to the list of that phrase's daughter nodes.
  `(map-down-phrase-structure-fn-nd
    #'(lambda (,phrase)
        (let (,@(if category `((,category (phrase-category ,phrase))))
                ,@(if daughters `((,daughters (phrase-daughters ,phrase)))))
          ,@body))
    ,top-phrase))

(defun map-up-phrase-structure-fn (function phrase)
  (dolist (daughter (phrase-daughters phrase))
    (map-up-phrase-structure-fn function daughter))
  (funcall function phrase))

(defun map-down-phrase-structure-fn (function phrase)
  (funcall function phrase)
  (dolist (daughter (phrase-daughters phrase))
    (map-down-phrase-structure-fn function daughter)))

(defun map-up-phrase-structure-fn-nd (function phrase)
  (dolist (daughter (phrase-daughters phrase))
    (map-up-phrase-structure-fn-nd function daughter))
  (funcall-nondeterministic function phrase))

(defun map-down-phrase-structure-fn-nd (function phrase)
  (funcall-nondeterministic function phrase)
  (dolist (daughter (phrase-daughters phrase))
    (map-down-phrase-structure-fn-nd function daughter)))

;;;
;;; Provide a function for finding minimal phrases meeting certain conditions.
;;; This is useful for binding theory.
;;;

(defun minimal-phrase-containing (phrase-list &key (restricted-to t))
  ;; Return the small phrase strictly containing all the phrases in
```

```
  ;; PHRASE-LIST, such that the phrase also is of type RESTRICTED-TO,
  ;; should that argument be provided.
  (let ((smallest-containers
         (mapcar #'(lambda (phrase)
                     (first-parent-restricted-to phrase restricted-to))
                 phrase-list)))
    (let ((minimal-phrase (first smallest-containers)))
      (dolist (starting-phrase (rest smallest-containers))
        (do ((phrase starting-phrase (phrase-parent phrase)))
            ((or (null phrase) (eq phrase minimal-phrase))
             (if (null phrase) (setq minimal-phrase phrase)))))
      minimal-phrase)))

(defun first-parent-restricted-to (phrase category-restrictions)
  ;; Find the smallest phrase strictly containing PHRASE and of a
  ;; category found in the list CATEGORY-RESTRICTIONS.
  (let ((parent (phrase-parent phrase)))
    (unless parent (fail))
    (if (or (eq category-restrictions t)
            (member (phrase-category parent) category-restrictions))
        parent
        (first-parent-restricted-to parent category-restrictions))))
```

Bibliography

[1] G. Edward Barton. Towards a principle-based parser. Memo A.I. Memo No. 788, MIT Artificial Intelligence Lab., Cambridge, Massachusetts, July 1984.

[2] G. Edward Barton. The computational structure of natural language. Phd thesis, MIT Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 1987.

[3] G. Edward Barton, Robert C. Berwick, and Eric S. Ristad. *Computational Complexity and Natural Language.* MIT Press, Cambridge, MA, 1987.

[4] Robert C. Berwick. Principle-based parsing. Memo 972, MIT Artificial Intelligence Lab., Cambridge, Massachusetts, June 1987.

[5] Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors. *Principle-Based Parsing: Computation and Psycholinguistics.* Kluwer Academic Publishers, Norwell, Massachusetts, 1991.

[6] Noam A. Chomsky. *Aspects of The Theory of Syntax.* MIT Press, Cambridge, MA, 1965.

[7] Noam A. Chomsky. *Lectures on Government and Binding.* Foris, Dordrecht, Holland, 1981.

[8] Noam A. Chomsky. *Some Concepts and Consequences of the Theory of Government and Binding.* MIT Press, Cambridge, MA, 1982.

[9] Noam A. Chomsky. A minimalist program for linguistic theory. Occasional Paper in Linguistics 1, MIT Department of Linguistics, 1992.

[10] Nelson Correa. Empty categories, chain binding, and parsing. In Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics.* Kluwer Academic Publishers, 1991.

[11] Matthew W. Crocker. *A Logical Model of Competence and Performance in the Human Sentence Processor.* PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1992.

[12] Bonnie Jean Dorr. Unitran: A principle-based approach to machine translation. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[13] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, Reading, MA, 1988.

[14] Sandiway Fong. *Computational Properties of Principle-Based Grammatical Theories.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1991.

[15] Gerald Gazdar, Ewan Klein, Geoffrey Pullam, and Ivan Sag. *Generalized Phrase Structure Grammar.* Harvard University Press, Cambridge, Massachusetts, 1985.

[16] Jerry R. Hobbs. Resolving pronoun references. *Lingua*, pages 311–338, 1978.

[17] Mark Johnson. Deductive parsing. In Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics.* Kluwer Academic Publishers, 1991.

[18] Shalom Lappin and Herbert Leass. A syntactically based algorithm for pronominal anaphora resolution. In *Proc. of the Univ. of Pennsylvania Cognitive Science Colloquium*, Philadelphia, Pennsylvania, 1992.

[19] Howard Lasnik and Juan Uriagereka. *A Course in GB Syntax: Lectures on Binding and Empty Categories.* MIT Press, Cambridge, MA, 1988.

[20] John M. Lucassen. Types and effects: Towards the integration of functional and imperitive programming. Memo MIT/LCS/TR-408, MIT Laboratory for Computer Science, Cambridge, Massachusetts, August 1987.

[21] Maria Rita Manzini. *Locality.* MIT Press, Cambridge, MA, 1992.

[22] Mitchell Marcus. *A Theory of Syntactic Recognition for Natural Language.* MIT Press, Cambridge, MA, 1980.

[23] Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, April 1968.

[24] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, March 1991.

[25] Eric Sven Ristad. The anaphora problem. *Information and Computation*, in press.

[26] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp.

[27] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type region and effect inference. Memo EMP-CRI E/150, Ecole des Mines de Paris, Paris, France, December 1991.

[28] Henk van Riemsjijk and Edwin Williams. *Introduction to the Theory of Grammar.* MIT Press, Cambridge, MA, 1986.