Computational Properties of Principle-Based Grammatical Theories

by

Sandiway Fong

B.Sc.(Eng), Computing Science, Imperial College of Science and Technology
University of London
(1984)

S.M. Electrical Engineering and Computer Science, Massachusetts Institute of Technology
(1986)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the
Massachusetts Institute Of Technology
June 1991

Signature of Author ______________________________

Department of Electrical Engineering and Computer Science
May 20th 1991

Certified by ______________________________

Robert C. Berwick
Associate Professor of Computer Science and Engineering,
Artificial Intelligence Laboratory,
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith
Chairman, Departmental Graduate Committee

Computational Properties of Principle-Based Grammatical Theories

by

Sandiway Fong

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science.

Abstract

Recently there has been a conceptual shift to modular theories of syntax in linguistics. Instead of being based on complex and large rule systems, these theories rely on the interaction of a small set of universal principles that are parameterized across languages. These theories pose many new and difficult problems for the design of parsing systems. The major issue tackled in this thesis is how to achieve substantial linguistic coverage and efficient parsing whilst maintaining a level of representation of principles close to that used in the linguistics literature.

This thesis describes a principle-based parsing system that achieves substantial coverage. Because of the free and complex interaction between principles, one major barrier to the use of these theories has been the problem of finding a comprehensive, yet consistent set of principles. Using a small set of twenty-five principles, the system has been demonstrated to correctly account for hundreds of different constructions from an introductory linguistics textbook. Also, illustrating the universal nature of principle-based theories, the same set of principles has been demonstrated to cover examples of Japanese as well as English data.

This thesis also investigates the problem of configuring principles for efficient parsing. The parsing system incorporates flexible parameters of control independent from principle definitions. These control parameters effectively define a family of parsers that incorporate the same linguistic knowledge, but with different performance characteristics. By investigating the effect of variations in control settings, we obtain a characterization of the relevant computational properties of principles that determine the most appropriate control configurations for efficient parsing.

Lastly, this thesis investigates the problem of representing principles at a level that is readily accessible to the linguist. Given the continual revision of linguistic theories, parsing systems must be updated frequently to track the revisions and advances in linguistic theory. By adopting a high-level, logic-based representation, together with the use of automatic compilation techniques, the system tackles the problem of bridging the gap between an abstract and an efficiently executable representation. This approach simplifies the necessary task of formalization and provides an answer to the parser obsolescence problem.

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

To See Seow Chu

Acknowledgments

This thesis is the product of countless, but occasionally productive hours spent staring at a computer console, fortunately relieved by invaluable and stimulating discussions with many people at MIT. In particular, I would like to extend my gratitude to the following individuals:

> My advisor, Bob Berwick, who provided constant encouragement and prodded for answers to many hard questions.
>
> The linguists with whom I had the opportunity to discuss various aspects of this project and those who patiently rectified many of my misconceptions and inadequate knowledge of syntactic theory: Howard Lasnik, Alec Marantz, David Pesetsky, Mamoru Saito, Noam Chomsky, Carol Tenny, Miori Kubo and Doug Jones.
>
> Certain members of the AI Lab. for many hours of both intellectual and non-intellectual conversations at odd hours of the morning, chiefly: Eric Ristad and Peng Wu.

Some sections from various chapters have appeared separately as papers. Section 3.4.2 is taken from “Free Indexation: Combinatorial Analysis and a Compositional Algorithm” in *ACL-90.* Chapter 5 is a much expanded version of “The Computational Implementation of Principle-Based Parsers” in *1st International Workshop on Parsing Technologies* 1989, in the book *Current Issues in Parsing Technologies*, ed. M. Tomita, Kluwer 1990, and partly in chapter 12 of *Artificial Intelligence at MIT: Expanding Frontiers. Volume 1.* ed. P.H. Winston. MIT Press. 1990. Sections 6.1 and 6.2 are taken from “Principle-Based Parsing and Type Inference.” in *3rd International Workshop on Natural Language Understanding and Logic Programming.* 1991, and will also appear as a chapter in the book *Natural Language and Logic Programming III*, eds. Brown & Koch. North Holland. 1991.

Contents

CHAPTER 1

Introduction

This thesis describes a natural language parser that:

- Is based on modern principle-based grammatical theories rather than traditional rule-based systems. Principle-based theories can replace thousands of traditional rules using a small number of fundamental linguistic principles. The implemented system employs only twenty-five principles plus a small number of (about thirty) phrase structure rules.

- Implements a more comprehensive linguistic theory than existing principle-based systems. The system achieves its goal of systematically covering hundreds of different examples from an introductory linguistics textbook, *A Course in GB Syntax* by Lasnik & Uriagereka [35].

- Has the ability to handle different languages using the same core set of universal principles despite seemingly large typological differences. For instance, the parser can be parameterized to handle either English or Japanese by specifying an appropriate parameter vector and lexicon.

- Permits the construction of a family of parsers which make use of the same set of principles (and therefore make the same grammaticality judgements), but with varying parsing times for different sentences. The system allows the user to alter simple parameters of control, independent from principle definitions, that can greatly affect the time spent parsing. For instance, the order in which principles are applied is subject to user control. Another option, called *principle interleaving*, allows a principle to be co-routined so as to apply to partial phrase structures as they are built, or alternatively, to wait until structure for a complete sentence has been recovered.

- Uses a high-level, logic-based representation allowing principle definitions that closely resemble the natural language definitions found in the linguistics literature to be constructed.

- Employs a variety of compilation techniques including canonical LR(1)-based parser-generation, type inference, simple partial evaluation, and program transformation to tackle the problem of automatically and transparently bridging the gap between an abstract and an efficiently executable representation. Of the above techniques, we will briefly expand on the two most interesting:

  1. The use of a canonical LR(1)-based algorithm for recovering phrase structures is particularly noteworthy. Although the algorithm is generally superior to comparable methods; in particular, with respect to the use of lookahead, very large tables have to be constructed for non-trivial grammars. This has precluded its use in natural language parsing as well as for programming languages. However, the relatively small size of the phrase structure grammar used here makes canonical LR(1)-based parsing eminently practical, as will be discussed in chapter 4.
  2. Another interesting technique is the use of type inference methods, based on a novel definition of the "type" of a linguistic principle, to safely reduce the amount of work required to apply principles as early as possible in the principle interleaving framework mentioned earlier.

- Provides a possible answer to the *parser obsolescence problem*; that is, the problem of updating parsers to keep track of frequent revisions and advances in linguistic theory. Because of the small grammar size, the high-level nature of the representation, the separation between principle definitions and control options, and the presence of automatic compilation procedures, the necessary tasks of formalizing definitions, ensuring correctness, and generating efficient parsers can be made much less tedious.

1.1 Outline of the Introduction

The following sections of this chapter is organized around the following topics:

- *Linguistic principles.*

  To introduce the notion of a linguistic principle and other concepts employed by principle-based syntactic theories, we will begin by examining what we can expect such theories to tell us about the interpretation of sentences.

- *Problems for principle-based systems.*

  We will then briefly list some of the major obstacles that must be overcome if principle-based parsers are to become widely adopted.

The major question addressed in this thesis is one of general parser design; that is, how to achieve substantial linguistic coverage and maintain efficient parsing, whilst retaining a perspicuous representation of grammar that can be readily maintained or updated? The remaining sections in this chapter will summarize how the implemented system tackles this question.

- *Linguistic coverage.*

  First, we will review the linguistic coverage of the implemented system. As noted earlier, the system has been extensively tested on a wide range of linguistic data found in Lasnik & Uriagereka. Using snapshots of the system in action, we will cover examples of well-formed input, both ambiguous and non-ambiguous. We will also illustrate how the parser explains ill-formed input in terms of the underlying principles. Furthermore, to provide an idea of the flexibility of the implemented theory, we will describe how the same system can also handle Japanese sentences. We will introduce the notion of language parameters by describing some of the important typological distinctions between English and Japanese. These distinctions will be highlighted using actual parses produced by the system on Japanese examples taken from the linguistics paper *On the Nature of Proper Government* by Lasnik & Saito [34].

- *Control options and parsing efficiency.*

  Secondly, we will describe the control options that the current system makes available to the user; namely, the freedom to shuffle the order in which principles are processed and the option of incrementally applying selected principles to partial structures as phrase structure construction proceeds. All possible control choices have the important property that they cannot affect the linguistic judgements rendered by the parser — only the amount of work that has to done to reach a particular judgement. By running the same sentence on two particular settings, we will demonstrate the large effect that these control parameters can have on parsing time. From this one example, we will expand the discussion to address the general problem of configuring the parser for efficient processing. Here, we will summarize the relevant results of chapters 5 and 6 by describing the computationally relevant features of individual principles that help to determine the appropriate control settings. Because control choices are transparently and automatically reflected in the parser (with respect to principle definitions), parser control can also be easily updated (as necessary) as principle definitions are revised. We will also take the opportunity to list some practical implementation issues that will turn out to be important factors in determining the effectiveness of the various options.

- *Implementation.*

> Next, we will provide simple examples of how principles may be perspicuously represented in a notation that closely resembles the definitions found in the linguistics literature. An important feature of the representation is that principle definitions can be transparently targeted for different parser configurations, i.e. the grammar writer is free to specify principles that are readily comprehensible without needing to know exactly how principles are realized or combined. A large portion of the system is devoted to maintaining this separation. Type inference and program transformation are among the methods employed to automatically compile principles into the underlying forms actually used to parse. We will briefly review the major components and general organization of the grammar compilation system.

Finally, we will end the introduction with an outline of the main chapters of the thesis.

1.2 Linguistic Principles

What do we mean when we say that a computer program can "parse" sentences? For example, what can a syntax tree tell us about a particular sentence? To answer these questions, let us consider what is involved when we say we understand a sentence such as (1):

(1) Which report did you file without reading?

To interpret this, we must know some simple facts about the words in the sentence: for example, that *file* is a predicate that takes two arguments, an 'agent' that does the filing and an element to be filed; similarly for *reading*, that there is an agent to do the reading and an element that is being read. Apart from the import of individual words, we must also understand that the element to be filed and read (or not read, for that matter) are one and the same object. Furthermore, this object must be construed, or be *coreferent*, with the 'report' that was mentioned at the beginning of the sentence. In a parallel fashion, it is also understood that the agent doing the filing, the (potential) reader, and the person denoted by the pronoun *you* must be one and the same person. Finally, we understand that no other interpretation is possible; that is, the sentence is unambiguous.

Modern syntactic theories can identify such coreference relations and the semantic, or *thematic*, roles that syntactic elements such as noun phrases and clauses play in a sentence. In addition to identifying the necessary elements for interpretation, such theories are capable of not only distinguishing between similar-looking sentences with radically different grammatical status, but also to provide a principled explanation for that difference. For example, such theories can also account for the following three counterparts to (1):

(2) a. *The report was filed without reading?[1]
b. *Who filed which report without reading?
c. Which report did you file without reading it?

A surprising property of some of these theories is that they can account for a wide variety of facts using a relatively small set of fundamental assumptions, or *principles.* What do these principles look like? And what are the kinds of objects that such principles operate on? Although it is beyond the scope of this introduction to fully explain how the theory described in Lasnik & Uriagereka accounts for the above data (see chapter 3 in Lasnik & Uriagereka for the details), we will briefly introduce some of the relevant principles that are crucial to the analysis. (The following description will also serve as an introduction to linguistic concepts that will be relevant to the parsing examples presented later in the chapter.)

- *Empty categories.*

  In general, thematic arguments such as the ones we have mentioned above must be represented in syntactic structure, either by overt or non-overt elements. For example, the position occupied by the argument *it* in (2c) is also assumed to exist in the parallel example (1). This so-called "gap" will be occupied by an *empty category* (EC). An important point to note is that, in many theories, these ECs will be treated as "first-class syntactic elements" in a sense to be made clear immediately below.

- *Binding conditions.*

  Binding conditions are principles that determine the coreference possibilities for elements such as anaphors, e.g. *himself*, or pronouns, e.g. *him* or *it* in (2c):

  (A) *An anaphor must be coreferent with an element in its clause.*
  (B) *A pronoun must **not** be coreferent with any element in its clause.*
  Even in this simplest form, such conditions are powerful enough to explain a variety of facts:

  (3) a. *[John_1 likes him_1] — *cf.* [John_1 likes him_2]
  b. John_1 believes that [Mary likes him_1]
  c. John_1 believes that [Mary likes him_2]
  d. [John_1 likes himself_1] — *cf.**[John_1 likes himself_2]
  e. John believes that [Mary_1 likes herself_1]
  f. *John_1 believes that [Mary likes himself_1]

  (Here we have used brackets to delimit the extent of the relevant clause and simple indices to distinguish the various interpretations.)

[1] Note: it is standard notation is to use an asterisk '*' to mark ungrammatical sentences.

Generally, noun phrases will be classified in terms of the independent binary features [±A] (anaphoric) and [±P] (pronominal). For example, *himself* and *him* are assumed to have the features [+A,−P] and [−*A*,+P], respectively. Referential expressions such as names, e.g. *John*, will be classified as [−A,−P]. These features are extended in a uniform manner to ECs that occupy the same subject and object positions as the overt pronouns, anaphors and names shown in (3). Binding conditions (A) and (B) will apply to overt and non-overt elements bearing the feature values [+A] and [+P], respectively. However, one question immediately arises: what are the appropriate values for ECs? The general rule that applies in such cases will be the null hypothesis; that is, all options should be freely available. The essential idea is that all but the appropriate option will be ruled out by the interaction of principles. For example, the EC that represents the object of *reading* in (1) cannot have value [+A] because it would violate Binding condition (A). The concept of free assignment operating under interacting constraints is a general and pervasive characteristic of many theories. Note that the interaction of constraints need not result in unique assignments, e.g. as in the case of coreference relations. For example, a pronoun can be ambiguous in the sense that it may have more than one possible referent, as illustrated by (3b) and (3c). The relevant principle in this case, condition (B), admits both possibilities.[2]

- *Functional determination.*

  Functional determination is another important constraint on the free assignment of [±A,±P] features to ECs. For the purpose of this discussion, we will be concerned with ECs that have feature values [−A,−P]. Such ECs are termed *variables* because of certain similarities to variables in standard logic that will be made apparent below. If we assume that the result of syntactic analysis will be some kind of *Logical Form* (LF) that will serve as the input to interpretation, then variables are elements that are bound by operators, e.g. a *wh*-word like *what* or *why*, or quantifiers, e.g. *someone*. For example, a LF representation for (1) will be of the form:

  (4) for which x, x a report, you file x without reading x

  In general, an EC that is most closely associated with an operator will be *functionally determined* as a variable, i.e. have feature values [−A,−P]. In particular, to be interpreted as a variable, an EC must occur within the scope

[2] Note that condition (B) also predicts that *it* in (2c) is ambiguous. In this case, the more obvious interpretation is for the pronoun to refer to the 'report'. However, a second interpretation is also available. Consider the following scenario: *"Did you get my 'Ten Rules of Filing'? Which report did you file without reading it?"*

> of an operator, in much the same way that a logical variable x must occur within the scope of a $\forall x$ or $\exists x$ in standard logic. In the case of (1), the objects of *file* and *reading* will be uniquely determined as variables so that (4) will be the only possible LF. This licensing restriction also explains why the empty object of *reading* in the similar example (2a) cannot be interpreted as a variable. Assuming that: (1) all other assignments of [$\pm$A,$\pm$P] are ruled out in both cases — we will not go into the details here, and (2) such an EC *must* bear some assignment of the [$\pm$A,$\pm$P] features; then functional determination has accounted for the contrast in grammaticality between the two examples.

To summarize, we have briefly seen how a modern syntactic theory can explain a variety of phenomena using just a few assumptions motivated by semantic interpretation concerns. Consider for a moment how one might account for the same facts in traditional rule-based systems such as context-free grammars (CFGs) or early transformational grammars (TGs), or computational formalisms such as the augmented transition networks (ATNs)? Barton [4] has described in detail the many scientific and engineering problems of such frameworks. We will not replicate those arguments here, save to mention that most formalisms seemed to suffer from a combination of the following problems: (1) it seemed descriptively necessary to have a large number of rules; (2) individual rules seemed to be overly detailed and complex, in some cases the unconstrained framework makes system extension difficult; and (3) because of the "closeness" to surface forms, rules were highly language-specific. Given these disadvantages, a descriptively adequate formulation in such frameworks (assuming it would be possible to do so) would certainly not be as succinct, nor have the explanatory force of the principle-based approach. Note also that the very general nature of the constraints we have described suggests that such theories are not restricted to any one particular language. Note also that there is no mention of word order, for instance, in any of them. In fact, linguists have shown that such principles, albeit with minor parametric variations, do indeed apply to a variety of different languages (as we will illustrate in section 1.4). Given all these advantages, we might naturally ask the question:

> Why, then, have there been so few natural language systems built using principle-based theories of grammar?

In the next section, we will review possible reasons why this might be the case. Then, from these and other considerations, we will provide an overview of how specific problems posed by the linguistics framework are tackled in a parser that uses principles to parse sentences such as the ones we have considered above.

1.3 Problems with Principles

Let us now turn to list some possible objections to principle-based system in general:

- *Principles are too vague, or underspecified. They cannot be formalized.*

  Most principles are stated incompletely, and sometimes ambiguously, using natural language (as opposed to some rigid mathematical formalism). Since a precise formulation of knowledge is normally considered to be a necessary first step towards use of that knowledge; it is not surprising that proper formalization is considered to be a difficult problem for potential grammar writers. However, Stabler [52] has painstakingly described how the *Barriers* framework of Chomsky [11] (by no means a complete theory, but nonetheless a good indication of the kind of theory that would be required for a "full" parser), may be axiomatized in first order logic. Of course, other issues such as that of correctness and completeness naturally arise from any formalization. We will return to consider these questions in chapter 2.

- *Principle-based theories are not logically consistent. Hence, even if they can be formalized, they cannot be used to parse.*

  Principles are naturally organized into *modules*, each dealing with a different aspect of syntax. For example, the *Binding conditions* mentioned earlier are part of *Binding theory* which deals with general conference conditions. The identification of thematic roles, e.g. 'agent', with positions in syntactic structure is part of *Theta* (θ) *-theory* which governs the proper distribution of thematic roles. Many theories actually focus on only a few modules; hence leaving open the possibility for conflicts. Moreover, linguists often propose several different accounts, sometimes within the same paper, without settling on any one particular version. Of course, to avoid an overly under-constrained system, the grammar writer will be forced to assemble a comprehensive set of modules.

- *There are too many gaps in coverage for current theories to be practical.*

  It is a fact that all existing linguistic frameworks are descriptively incomplete. However, coverage in current principle-based theories is somewhat uneven. Although such theories are capable of explaining linguistically sophisticated examples that are relatively uncommon in everyday usage, e.g. the sentences shown in (1) and (2), they cannot handle unrestricted text.[3]

[3] Note that systems with goals to "parse" arbitrary text do exist (see Fujisaki [25] and de Marcken [19] for examples of recent work). Such systems are typically based on traditional rules, sometimes augmented with statistical methods. However, it is not clear that such systems will be able to handle sentences such as the ones that we have been discussing, nor recover as much in the way of linguistic information.

- *Current implementations are "toy" systems.*

  Although nothing near what could be called a complete set of principles is currently available, most initial efforts can be characterized as making use only of a very restricted subset of the available principles. The lack of a readily-available formal specification of a fairly large and consistent set of principles is probably the major contributing factor here.

- *Principle-based systems are too slow.*[4]

  Conventional systems based on, say, context-free (CF) rules have been able to draw upon the many efficient methods that were initially developed for fast parsing of computer programs. By comparison, the principle-based systems that have been built are certainly much slower. Barton [4] notes that Fodor has argued that modular systems are at a disadvantage because they must access and integrate information from more than one source.[5] We note here that there has been relatively little in the way of direct comparisons between different ways of organizing the same principles for efficient processing. Moreover, current systems have not always exploited specific computational properties of individual principles to obtain efficient interpretation or compilation methods.

The above list is by no means an exhaustive catalogue of the potential problems, but it should give the reader some indication of the (very real) difficulties that challenge parser designers. We will return to expand on these (and many other relevant issues for parser design) in chapter 2, where we will take the opportunity to contrast the approach taken in this thesis with other existing systems. However, in the next section, we will focus on the current implementation. We believe that the system goes a long way towards demonstrating that (relatively) large scaled and sophisticated principle-based parsing systems can be built. As we shall see, principles from many modules can be formalized and shown to be consistent insofar as many hundreds of examples taken from Lasnik & Uriagereka have been "correctly", or faithfully, accounted for. The operational notion of *correctness* that we will adopt in this thesis is that the parser must recover the same structure (or structures in the case of ambiguity) given in Lasnik & Uriagereka. In the case of ill-formed sentences or structures, the parser must report that the input violated the same principles as predicted in Lasnik & Uriagereka.

---

[4] On a tangential note, Chomsky [13] (also in [9] and earlier work) has argued because of the divergence between a (human) parser and grammar (with respect to performance issues), that there's no reason to expect grammars to be (readily) parsable. He also argues that an analogous observation holds with respect to learnability.

[5] Incidentally, by Fodor's definition, systems that maintain a strict separation of principles in visible definitions, but merge principles during a (hidden) compilation step would be non-modular. The system described in this thesis is modular provided no principles are interleaved.

1.4 Linguistic Coverage

As far as we are aware, the system implements a substantially larger (and more sophisticated) fragment of a principle-and-parameters theory than any extant parser.[6] To provide some idea of the capability of the implemented system, we will take the opportunity here to present examples of linguistic phenomena that the system can handle. To test the feasibility of implementing a substantial grammar, an introductory linguistics textbook, namely *A Course in GB Syntax* by Lasnik & Uriagereka [35], was chosen as a representative example of a current principle-based theory. The basic idea was to simply try to "implement" the book; that is, to implement the principles defined there, and to have the system correctly handle all the accompanying linguistic data. Of course, since the text covers several different (mutually incompatible) theories, it was not possible to achieve that goal. Moreover, the theory was found to be substantially incomplete. Consequently, additional principles were imported from *Knowledge of Language*, Chomsky [12], and other sources. Nevertheless, the subset of the principles implemented achieves substantial coverage. In fact, the system successfully accounts for almost all of the examples to be found in the first four chapters of the reference text.[7] For example, figure 1.1 contains a partial listing of representative sentences that the current system has been tested on. The surprising fact is that such coverage can be achieved using a core set of only twenty-five principles.[8]

As mentioned earlier, the parser produces correct parses in the sense that it makes the same judgements as described in the textbook for each example. That is, in the case when a sentence is "well-formed", the parser will return the structure, or structures, predicted by the theory. Operationally, a sentence will be considered to be *well-formed* only if there is at least one structural description that satisfies all the principles. On the other hand, if a sentence is *ill-formed*, there will be at least

[6] For example, consider Correa's attribute grammar formulation described in his thesis ([15]). The major differences in coverage can be attributed to the fact that his system is based on a much earlier instantiation of standard theory. Also, his parser lacks the Empty Category Principle (ECP), Control, LF-movement and (and the concomitant LF licensing constraints), head movement, adjunction resulting from movement, multiple-*wh*-extractions, genitive Case markers, and various other minor operations. Correa's system is essentially unparameterized and can only handle English. However, the system contains many interesting features which we return to in chapter 2.

[7] *A Course in GB Syntax* contains six chapters. Chapter five which covers extensions and alternatives to the Binding theory, Aoun's Generalized Binding and Higginbotham's Linking theory, has not been implemented. Chapter six discusses several open questions and unresolved problems. Of the various proposals discussed there, the current implementation has adopted the prohibition against Case-marking of NP-traces, and genitive Case realization (see Chomsky [12]) as a partial solution for *illicit NP-movement.*

[8] This figure does not include the grammar rules that define basic phrase structure syntax. There are ten basic $\overline{\text{X}}$-rules plus another twenty covering adjunction and the insertion of empty categories. A detailed description of the grammar rule system can be found in chapter 4.

(1:12a)	Someone likes everyone	*Quantifier raising*
(1:15a)	Who that John knows does he like	*Movement & Condition C*
(1:15b)	He likes everyone that John knows	
(1:18)	It is likely that John is here	*Simple Case theory*
(1:19)	*It is likely John to be here	
(1:21)	I am eager for John to be here	*Exceptional Case Marking*
(1:22)	*I am eager John to be here	*(ECM)*
(1:24a)	I believe John to be here	
(1:24b)	I believe John is here	
(1:25)	*I believe sincerely John to be here	*Case Adjacency*
(1:35a)	I want to be clever	*Optional vs. obligatory ECM*
(1:35b)	*I believe to be clever	
(1:36a)	John was persuaded to leave	*Ordinary passivization*
(1:48)	John was arrested by the police	
(1:36c)	*John was wanted to leave	*Exceptional passivization*
(1:49a)	I believe John to be intelligent	
(1:49b)	*It was believed John to be intelligent	
(1:49c)	John was believed to be intelligent	
(1:52a)	*I am proud John	*Genitive Case realization*
(1:52b)	I am proud of John	
(1:53a)	I wonder who you will see	Wh-*movement and Subjacency*
(1:59c)	*What does Bill wonder who saw	
(2:19b)	Their pictures of each other are nice	*Simple Binding theory*
(2:26a)	John likes Mary's pictures of him	
(2:26b)	*John likes Mary's pictures of himself	
(2:45a)	Who does he think Mary likes	*Strong crossover*
(2:88)	*(I am proud of) my belief to be intelligent	*Government of PRO*
(2:91)	The men think that pictures of each other will be on sale	
(2:103)	*The men think that Mary's pictures of each other will be on sale	
(3:17)	Which report did you file without reading	*Parasitic gaps*
(3:18)	*Which book did you file the report without reading	
(3:19)	*Who filed which report without reading	
(3:20)	*The report was filed without reading	✓*Resumptive pronoun*
(3:46)	*The article which I filed it yesterday without reading is (over) here	
(4:3)	*John is crucial to see this	*ECP*
(4:4)	John is certain to see this	
(4:7a)	Who do you think that John saw	*That-trace effect*
(4:7b)	Who do you think John saw	
(4:8a)	*Who do you think that saw Bill	
(4:8b)	Who do you think saw Bill	
(4:18)	*John is believed is intelligent	*Raising and Super-raising*
(4:20)	*John seems that it is likely to leave	
(4:21b)	Who will read what	*Superiority*
(4:21c)	*What will who read	
(4:35a)	Why did you read what	*Complement-noncomplement*
(4:35b)	*What did you read why	*asymmetries*
(4:45a)	Who believes the claim that Mary read what	*Subjacency and LF*
(4:45b)	*What do you believe the claim that Mary read	
(4:57)	*Who does Mary wonder why John hit	*ECP and syntactic movement*
(4:58)	*Why does Mary wonder who John hit	

Figure 1.1 Examples from Lasnik & Uriagereka

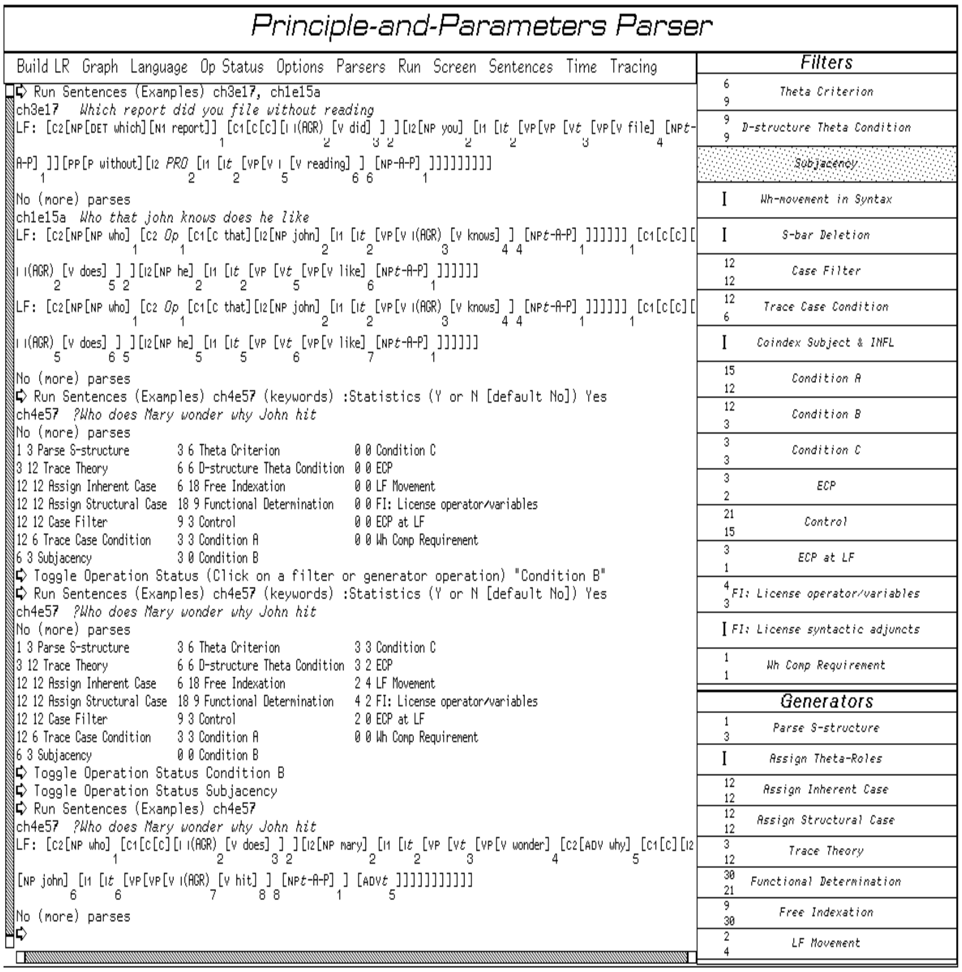


Figure 1.2 System snapshot.

one principle that no structural description will satisfy.

Consider the system snapshot shown in figure 1.2. The display is divided into two parts. Parses are displayed on the left. The right side contains a panel listing the parser operations. In most cases, each parser operation will map onto a single principle. This panel is used to provide an animated display that provides much useful information on how parsing is proceeding. Basically, there are two counters associated with each operation. (Some operations may have an 'I' instead of two counters. The relevance of this mark will be explained when we return to discuss parser control.) The upper counter records how many times the operation was called, and the lower counter records how many times the operation succeeded. The relative number of calls and successes will indicate if a principle is behaving as a *filter*, i.e. ruling out structures, or as a *generator*, i.e. instantiates more structures than it eliminates. For example, the operation *Parse S-structure* has been called once, but has succeeded three times, which means that there are three possible phrase structures that can be assigned to the input sentence at S-structure. Similarly, *Condition B* has ruled out nine of the twelve structures passed to it.[9] We can also identify the most and least active operations in terms of eliminating ill-formed structures. For example, unlike *Condition B*, the *Case Filter* has had no effect since all of the structures it has received have been allowed to pass. Generally, if a sentence is well-formed, all counters will be non-zero. When a sentence is ill-formed, there will be a single operation F (the current system executes operations serially) for which the number of calls will be non-zero but there will be no successes registered. Then, let us say for now that the sentence is an example of a F filter violation. (We will revise this definition shortly.)

The display on the left contains the result of parsing three sentences. Let us consider each of them in turn:

- *Identifying coreference relations.*

  The first parse at the top of the figure shows the LF structure returned for the *parasitic gap* sentence *which report did you file without reading?* (1) (= (3:18) in Lasnik & Uriagereka) that was discussed at the beginning of the chapter.

  The LF shown here has considerably more detail than necessary to identify the coreference relations. We will just pick out the relevant information. The object positions immediately to the right of both *file* and *reading* have been occupied by two empty noun phrases, shown as [NP*t*-A-P] and [NP-A-P], respectively. In either case, the features -A (non-anaphoric) and -P (non-pronominal) indicate that they are not subject to either Binding condition introduced earlier in the chapter. The fact that they both have index 1, as indicated by the subscripts shown, will mean that they will be both interpreted as variables bound by the NP *which report* that shares the same index.[10] Using the simple interpretation rule:

[9] Actually, most of the structures passed around are merely further instantiations of the same basic structures. For example, all twelve of the structures produced by *Trace Theory* actually are variants of the three (underspecified) phrase structure generated by *Parse S-structure*. The twelve structures are simply distinguished by different assignments of movement chain features. In particular, no constituents have been discarded or built by *Trace Theory*.

[10] Note that there is a syntax-internal difference (irrelevant to interpretation) in the derivation of the two variables. [NP*t*-A-P], as indicated by the '*t*' is the trace of *which report*, i.e. it is formed as a result of fronting *which report*. On the other hand, [NP-A-P] is not formed by movement, but "base-generated".

(5) Two elements are coreferential if and only if they share the same index

and ignoring all indices not attached to NPs, we can see that the parse contains the correct coreference option. In a similar fashion, the embedded subject position of the prepositional phrase *without reading* has been occupied by an empty NP, *PRO*. As expected, *you*, the subject of *file*, gets the same index as *PRO*, namely 2 and, crucially, not 1.

- *Reporting ambiguity.*

  In the case of ambiguity, the system will return as many structures as the degree of ambiguity demands. For example, consider the sentence:

  (6) who that John knows does he like?

  This is the next sentence shown in the snapshot. Here, the two possible LFs returned for are identical save for the index on *he*. The first LF corresponds to the case where *he* should be interpreted as being coreferential with *John*, the second LF corresponds to the case where *he* refers to someone not named in the sentence (ignoring non-NPs again, index 5 does not occur anywhere else in the structure).

- *Explaining ill-formed input.*

  Consider the following ill-formed sentence:

  (7) *who does Mary wonder why John hit? *= (4:57) in [35]*

  According to Lasnik & Uriagereka (section 4.4.2), this is a straightforward example of a *Subjacency* violation. The basic assumption here is that *who* has been extracted from its original position as the object of *hit*. The standard explanation why the sentence is ill-formed is that *who* has moved "too far" from its base position. Now consider the output shown in the snapshot. No LF has been returned since the sentence is ill-formed. Instead, the parser has reported the counter values for each operation.[11] Surprisingly, the "blocking" filter reported in this case is *Condition B* from the Binding theory, not *Subjacency* as predicted by Lasnik & Uriagereka.

  A useful feature of the system is that it can be directed to turn off any parser operation. Thus, ill-formed parses can sometimes be "saved" by simply turning off the relevant blocking filters. For some cases, this will result in a still

[11] The order in which the operations have been executed can be determined by reading the list top-down, starting from the leftmost column.

interpretable sentence.[12] So, let us explore what happens when *Condition B* is turned off. (This is accomplished by the 'Toggle Operation Status' command.) Running the sentence again, we see that it is still blocked, but this time at the *Empty Category Principle (ECP)* that operates at LF. Therefore the sentence cannot be an example of a *Condition B* violation.

Let us now backtrack and see if Lasnik & Uriagereka were correct. First, we need to re-enable *Condition B* and then turn off *Subjacency*. (This is indicated in the latter case by the "blanked out" display on the right panel.) Running the sentence one more time, we can see that a LF is returned. Again, ignoring irrelevant details in the structure, both *who* and the EC [NP*t*-A-P] immediately following the verb *hit* have the same index. Moreover, the '*t*' indicates that it is indeed the trace of movement of *who*, as Lasnik & Uriagereka predicted. Therefore, the sentence is not a *Condition B* but a *Subjacency* violation. Hence, we should revise the tentative definition of a F filter violation to include the condition that the sentence is well-formed if F does not apply.[13]

In general, the search for all possible parses can produce somewhat unexpected results. In the next section, we will illustrate other cases when the parser actually produces unanticipated parses. We note that, in such cases, the parser is not "wrong" in the sense that it is producing parses that are inconsistent with linguistic theory, but rather that the theory as implemented is under-constrained.

1.4.1 Language Parameters

Using basically the same set of principles, but with a different language parameter vector and lexicon, the system can be automatically re-configured to parse Japanese examples instead. We would like to emphasize, however, that the system has not been rigorously tested on Japanese sentences. So far, only a very limited set of test cases have been tried, namely the Japanese sentences found in the linguistics paper *On the Nature of Proper Government* by Lasnik & Saito [34]. These sentences are listed in figure 1.3. (Note that they are all *wh*-questions.) Nonetheless, these sentences display many of the typological Japanese–English differences. Let us now consider some of these:

[12] This roughly corresponds to the situation in which a human parser detects that a sentence is "syntactically" anomalous, but nevertheless is still able to assign it an interpretation.

[13] However, one mystery remains: What were the structures that got past *Subjacency* but *Condition B* ruled out? All three were of the form: *who does Mary wonder why* $John_i$ *hit* PRO_i. To satisfy the θ-criterion, both *who* and *PRO* are assumed to have been base-generated in Comp and object position of *hit*, respectively. *PRO* had to be bound by *John* in order to satisfy *Condition A*. But, with *PRO* bound in its governing category, all three are subsequently ruled out by *Condition B*.

(2)	Watashi-wa Taro-ga nani-o katta ka shitte iru	*Basic* wh-*questions*
	(I know what John bought)	
(6)	Kimi-wa dare-ni Taro-ga naze kubi-ni natta tte itta no	*Good in Japanese*
	(To whom did you say that John was fired why)	*but not in English*
(32)	*Meari-wa Taro-ga nani-o katta ka do ka shiranai	*Semantic parallelism:*
	(Mary does not know whether or not John bought what)	*non-absorption of* ka do ka
(37a)	Taro-wa naze kubi-ni natta no	*Comp-to-Comp*
	(Why was John fired)	*movement at LF*
(37b)	Biru-wa Taro-ga naze kubi-ni natta tte itta no	*Long distance*
	(Why did Bill say that John was fired)	*movement of* naze
(39a)	Taro-ga nani-o te-ni ireta koto-o sonnani okotteru no	*Complement-noncomplement*
	(What are you so angry about the fact that Taro obtained)	*asymmetries*
(39b)	*Taro-ga naze sore-o te-ni ireta koto-o sonnani okotteru no	
	(Why are you so angry about the fact that Taro obtained it)	
(41a)	Hanoko-ga Taro-ga nani-o te-ni ireta tte itta koto-o sonnani okotteru no	
	(What are you so angry about the fact that Hanoko said that Taro obtained)	
(41b)	*Hanoko-ga Taro-ga naze sore-o te-ni ireta tte itta koto-o sonnani okotteru no	
	(Why are you so angry about the fact that Hanoko said that Taro obtained it)	
(60)	Kimi-wa nani-o doko-de katta no	*Multiple*-wh*s in Comp*
	(Where did you buy what)	
(63)	Kimi-wa nani-o sagashiteru no	
	(Why are you looking for what)	

Figure 1.3 *Wh*-movement examples in Japanese from Lasnik & Saito.

- *SOV Language.*

  Japanese is classified as a SOV (Subject-Object-Verb) language. In general, heads such as verbs and adjectives are preceded by their objects and modifiers. However, subjects do normally appear before verbs and objects, as in English. This distinction can be encoded by two binary parameters that specify head/complement and specifier/head order: [14]

Language	Head/ Complement	Specifier/ Head
English	head first	specifier first
Japanese	head final	specifier first

- *Scrambling.*

  Apart from the fact that sentences normally begin with a topic and end with a verb, the order of other elements in the sentence is relatively free. In partic-

[14] We have oversimplified things here somewhat. The parser can use a slightly more complex system. In general, the actual order can vary depending on the syntactic category. See chapter 3 of Webelhuth [57] for a summary of the differences between the seven modern Germanic languages.

ular, direct and indirect objects can be switched, direct (and indirect) objects can appear in front of the subject in sentence-initial position. For example, consider the following four sentences corresponding to *John gave Mary a book* (taken from Hoji [29]):

(8) a. [*S* John-ga [*VP* Mary-ni hon-o ageta]]
b. hon-o John-ga Mary-ni ageta
c. Mary-ni John-ga hon-o ageta
d. John-ga hon-o Mary-ni ageta

This can be accomodated in the present theory using movement. The canonical order is assumed to be 'subject' followed by 'indirect object' followed by 'direct object', as in (8a). Then, the direct object, *hon-o* in this case, is free to move (by *VP-adjunction*) to a position in front of the indirect object *Mary-ni*, as in (8d), or to a sentence-inital position (by *S-adjunction*) as in (8b). Similarly, the indirect object may move to a sentence-initial position as in (8c).

- *Empty subjects.*

  Subjects can be omitted in Japanese. (In general the conditions that determine which elements can or can not be omitted are largely dependent on discourse considerations. We ignore such concerns here.) For example, consider the second sentence in (9) (taken from Makino & Tsutsui [38]):

  (9) Taro-wa sono mise-de nani-o kaimashita ka.
  (What did Taro buy at the store?)
  Pen-o kaimashita
  (He bought a pen)

  Other languages with this feature include Chinese, Italian and Spanish. In standard theory, the omitted subject is actually represented in syntax by an empty pronoun *pro* (which is not available in English). The binary option that determines whether *pro* is available or not is known as the *pro*-drop parameter.

- *No visible Wh-movement.*

  In English, wh-words such as *what* and *why* must appear in clause-initial position, as shown in (10) (taken from Lasnik & Saito [34]):[15]

  (10) a. I know [what John bought]
  b. *I know John bought what

[15] Excluding cases of echo-questions such as *you saw who*, and sentences containing multiple-*wh*-words — for example, see sentences (4:35a) and (4:45a) in figure 1.1.

It is usually assumed that (10b) is the "underlying", or *D-structure*, form for (10a). The *wh*-noun phrase *what* then moves to a position at the front of the embedded clause. By contrast, in Japanese the correspond *wh*-word, *nani*, remains in object position, as example (11) illustrates (= (2) in figure 1.3):

(11) Watashi-wa Taro-ga nani-o katta ka shitte iru

Although, Japanese sentences do not seem to exhibit *Wh*-movement as in English; it is generally assumed that similar sentences will have basically the same representation at LF. As with *which report* in the parasitic gap sentence discussed at the beginning of the chapter, it is assumed that *what* behaves somewhat like a quantifier in logic. In particular, it must move to a position that has clausal scope. To summarize, the basic difference here is that *wh*-elements move at different levels of representation in English and Japanese. That is, the following generative model of phrase structure is assumed:

$$\begin{tikzcd}[column sep=small] & \text{D-structure} \arrow[d] & \\ & \text{S-structure} \arrow[dl] \arrow[dr] & \\ \text{Phonetic Form} & & \text{LF} \end{tikzcd}$$

Here, roughly speaking, *Phonetic Form* serves as the level of input for parsing. Hence in Japanese, there wil be no *wh*-movement between D- and S-structure; all cases of *wh*-movement taking place "invisibly" between S-structure and LF. Thus, the option of whether to allow *wh*-movement between D- and S-structure is a language parameter. We will not comment further except to note that this distinction has enabled Lasnik & Saito to explain a variety of facts including why the counterpart of a sentence such as (6) in figure 1.3 (which is well-formed in Japanese) is ill-formed in English.

While obviously this is very far from being a complete characterization of the differences between Japanese and English, it is sufficient to cover the examples shown in figure 1.3. Figure 1.4 shows the system in operation for the first five sentences from figure 1.3. We will just point out a few interesting features of some of these parses:

- Wh-*movement at LF.*

  Consider example (11) that was discussed above. As shown at the top of the snapshot, *nani* has moved at LF to a position that has scope over the embedded sentence (as indicated by the bracketing) leaving behind a trace $\mathsf{LF}t$, to be interpreted as a variable, in its original position.

- *Multiple wh-elements at LF.*

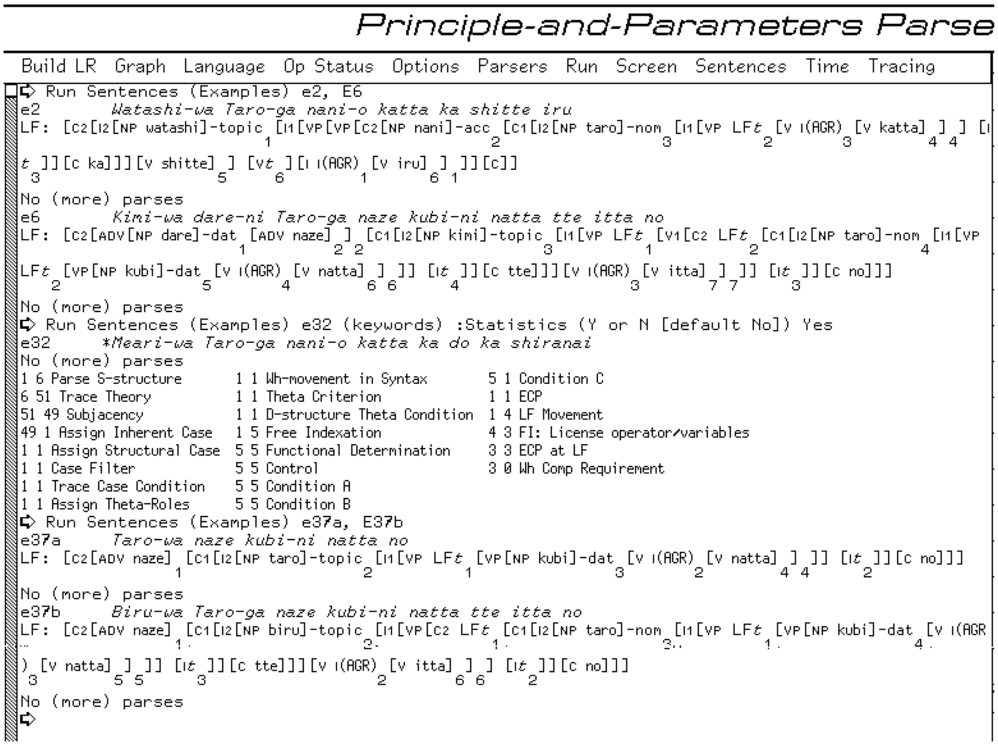


Figure 1.4 Examples from Lasnik & Saito: Part 1.

Consider example (6) in figure 1.3. There are two wh-elements: (1) *dare* ('who') the indirect object of *itta* ('said' as in 'said something to somebody'), and (2) *naze* ('why'). As the second parse shows, both elements will move at LF to a sentence-initial position. Hence, roughly speaking, the sentence will be interpreted as a double question: 'for which x, x a person, *and* for what reason y, you said y (a reason such that [taro was fired]) to x.'

We will conclude this section with a demonstration of how the parser can produce unexpected parses. Consider (39a) from figure 1.3, repeated here as (12):

(12) Taro-ga nani-o te-ni ireta koto-o sonnani okotteru no
(What are you so angry about the fact that Taro obtained)

(Here, note that the subject of the matrix clause (= *you*) has been omitted. Also, *nani* and *te* ('hand') have been permuted from the canonical order described above — a simple case of scrambling.) The LF for this sentence should be something along the lines of:

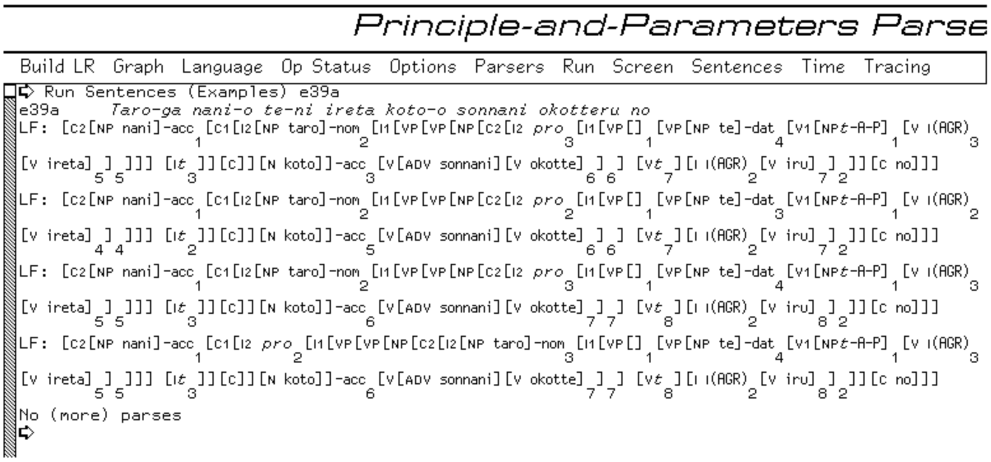


Figure 1.5 Examples from Lasnik & Saito: Part 2.

(13) for what x, *pro* is so angry about [the fact that Taro obtained x]

Here, *pro* represents the understood subject of *okotteru* ('be angry'). Now consider the LFs returned by the system in figure 1.5. As we can see, the system does correctly recover this form, as the last LF in the snapshot. However, it also recovers three additional LFs which (ignoring indices for the time being) all have the same form:

(14) for what x, Taro is so angry about [the fact that *pro* obtained x]

The essential difference is that the parser has predicted that it is possible to "exchange" the embedded subject *Taro* for the matrix subject *pro*.[16] As it turns out, the basic prediction is correct as (12) happens to be ambiguous with respect to the two basic interpretations.[17] (Actually, the original example as stated in [34] is unambiguous because of the use of bracketing.[18]) This concludes the overview of the linguistic capabilities of the system. In the next section, we will briefly describe the available control options.

[16] For simplicity, we have omitted discussion of many complex but interesting details about the parses returned by the system. The actual derivation sequence in figure 1.5 is quite complicated. For example, the observant reader may notice that *nani* ('what') undergoes both S-structure and LF movement. At S-structure, the direct object "scrambles" by VP-adjunction leaving the trace $[_{NP}t\text{-A-P}]_1$. At LF, *nani* undergoes *wh*-movement into COMP (actually, the specifier position of COMP) with the LF trace being subsequently deleted, as indicated by $[\,]_1$.

[17] This was pointed out to the author by D. Pesetsky, and confirmed by M. Saito.

[18] For completeness, here are the three variants of (14) that correspond to the first three LFs reported by the system:

1. *pro* is coreferent with *koto* ('fact'):

1.5 Parser Control

In the previous section, we have seen one simple example where the system allowed the user to control how principles are applied. There we described how filter violations could be verified by turning off certain parser operations. In this section, we will turn our attention to control options of a different nature. These control options have the common property that all linguistic judgements are preserved. Therefore, the only difference that the user will see is in the time it takes for the system to return parses.

Most existing principle-based parsers have their program control "hardwired" into the system. For example, the order in which principles are processed is normally fixed. However, a natural question arises: What is the justification for preferring any one ordering over another? Or more generally we can ask: What is the most efficient way to apply principles? For some systems such as Abney's [2], the goal has not been necessarily to find the most effective organization, but instead to find one that is psycholinguistically plausible. Since the linguistic theory given in Lasnik & Uriagereka is one that deals with aspects of language competence rather than performance, the system described here has no processing constraints of that sort; the task being simply to recover all possible parses in no particular order.

In general, it may not be possible to predict an appropriate control strategy ahead of time, i.e. without doing some experimentation. Furthermore, an appropriate strategy may not continue to be one if principles are altered. To conclude, there seems to be little reason in the current system to unnecessarily restrict the range of control options.[19] Although it would be quite infeasible to cover all possible parsing strategies, the approach taken here is to allow simple variations on the basic processing framework, so long as the system is able to re-configure the parser control *automatically*, i.e. without assistance or special programming on the part

---

for what x, Taro is so angry about [the fact that the fact obtained x]

This interpretation can probably be eliminated by imposing selectional restrictions on the possible 'agents' of *okotteru*.

2. *pro* is coreferent with *taro*:

   for what x, Taro is so angry about [the fact that Taro obtained x]

   (We have not consulted a native Japanese speaker on the grammaticality of this interpretation, but currently the system does not rule it out.)

3. *pro* is free in the sentence:

   for what x, Taro is so angry about [the fact that (someone else) obtained x]

[19] Perhaps being able to vary control parameters (even if those parameters may not quite be the appropriate ones) would be a useful option for parsers that attempt to model psycholinguistic predictions.

of the grammar writer, and *transparently*, i.e. without missing well-formed parses, accepting ill-formed sentences, or altering the structures that the user sees in any way. (We will return to consider possible strategies in more detail in chapter 2.)

1.5.1 An Outline of the Section

This section consists of three parts:

1. In the first part, we will introduce the two models of parsing that the present system accommodates. We will discuss the *principle ordering* model of parsing that relies on the fact that principles are generally unordered. Next, we describe an alternative *principle interleaving* model in which principles are applied to partially-built structures as phrase structure construction proceeds.

2. We will illustrate the differences in parsing times that can occur as a result of different control choices with a simple comparison of two parsers, one based on the principle ordering model and the other one a hybrid model that combines both approaches.

3. We will end the section with a general overview of the control decisions and tradeoffs for configuring efficient parsers in the present system that will be discussed in depth later in chapters 5 and 6.

1.5.2 Two Models of Parsing

- *Principle ordering.*

  Consider again the panel of parser operations shown in the system snapshot of figure 1.2. The basic model of parsing allowed by the system is to connect, or "link", up these operations in some prescribed order, and then to pass hypothesized structures from one operation to the next in an assembly line fashion. Each operation may: (1) reject or pass on without comment the structure it is given, thus behaving as a filter, or (2) augment the structure in some way, e.g. the operation *Assign Theta-Roles* instantiates the θ-slots of the arguments of heads such as verbs, nouns and adjectives, or (3) create another level of representation by rearranging the elements of the existing structure, e.g. the *LF movement* operation may move *wh*-noun phrases and quantifiers to clause-initial positions. In general, the user is free to rearrange the "assembly line" order, say, to obtain faster processing (as we shall see below). Hence, a *parser specification* in this system is simply a description of how the operations are to be connected.

  However, there are certain restrictions on what permutations are allowed. (See the dependency graph in figure 1.6.) For example, the system does not

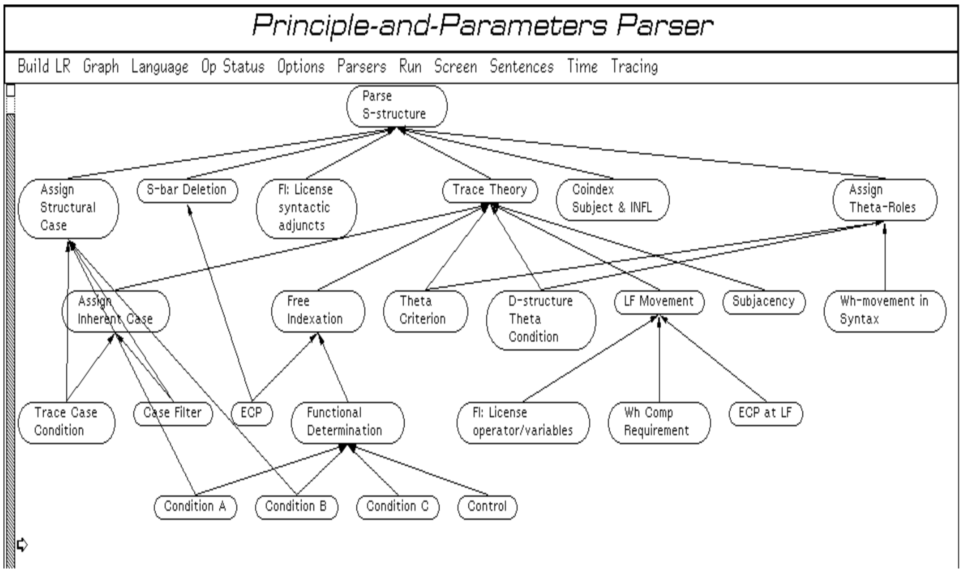


Figure 1.6 Operational dependency graph.

allow the construction of a parser that puts the *Case Filter* (a requirement that all non-empty noun phrases be assigned Case) ahead of either *Inherent Case Assignment* or *Structural Case Assignment* (the two operations that can assign Case to noun phrases in certain configurations). In turn, neither Case assignment operations can be put ahead of *Parse S-structure* (the operation that builds syntax trees) because both operations need to know the positions that noun phrases occupy in syntactic structure in order to know which noun phrases are permitted to receive Case. (Note that this is an overly simplistic model; that is, the restrictions described here are not fundamental to the linguistic theory, but only to this particular implementation.[20])

- *Principle interleaving.*

  The "unit of currency" adopted in the assembly line model is a "complete" phrase structure tree. (Here, by "complete" we mean only that all constituents

[20] For example, we can remove such restrictions by viewing the Case assignment process in a slightly different way. We can re-express the *Case Filter* as simply marking each overt NP with a feature indicating that the NP must get Case. Then we can proceed with both Case assignment operations in the normal fashion. Finally, we can check that each *Case Filter* tag was properly satisfied. Another option is to eliminate this extra last step by combining the three Case theory operations into one that visits each noun phrase in turn. However, for simplicity and other design considerations that we will discuss later, we do not consider these alternate formulations.

have been constructed. In particular, syntactic features of individual constituents, e.g. Binding features [±A,±P] or indices, may remain incompletely instantiated.) One limitation of this model is that principles cannot be applied to smaller fragments of phrase structure. One reason why one might want such a model would be to avoid the unnecessary work of constructing a complete but ill-formed phrase structure if some fragment of that structure could be shown to be ill-formed independently from the rest of the structure. For example, consider the sentence:

(15) *[the man I saw him] wondered why Mary bought the tree

The matrix subject here is the (ill-formed) relative clause construction *the man I saw him*. If the parser can apply the principle that rules out this matrix subject without building the structure for the whole sentence, then we can save a considerable amount of work. The present system accommodates such strategies by allowing the user to specify that principles should be *interleaved*, i.e. co-routined, with phrase structure construction in *Parse S-structure*. Then, *Parse S-structure* will only output complete structures that satisfy the interleaved principles. (We will return to discuss the connection between relative clauses and principle interleaving in the context of *licensing parsers* in chapter 2.)

Note that the two models we have described are not mutually exclusive options. In the current system, a parser specification consists of two parts: (1) a designated subset of principles to be interleaved (possibly none), and (2) a particular ordering of the remaining principles. (We emphasize that such control options are specified independently from the representation of the principles used by the system. It is perhaps also worth pointing out that the grammar writer need not be concerned about how principles are to be interleaved (or re-ordered). Both procedures are carried out in a completely automatic fashion by the system.)

1.5.3 Two Parsers, One Theory

To illustrate the difference in parsing time that can result from different parser specifications using exactly the same set of principles, consider the system snapshot shown in figure 1.7.

Here, the results of parsing *who does Mary wonder why John hit* (= (7), the subjacency violation discussed in the previous section) are shown for two different parsers. The default parser used by the system is one in which no principles are interleaved. Instead, the principles are applied in the order given by the list of operations at the top left side of the snapshot. Here, the order of application can be read top-down starting at the left-most column. So *Parse S-structure* is applied

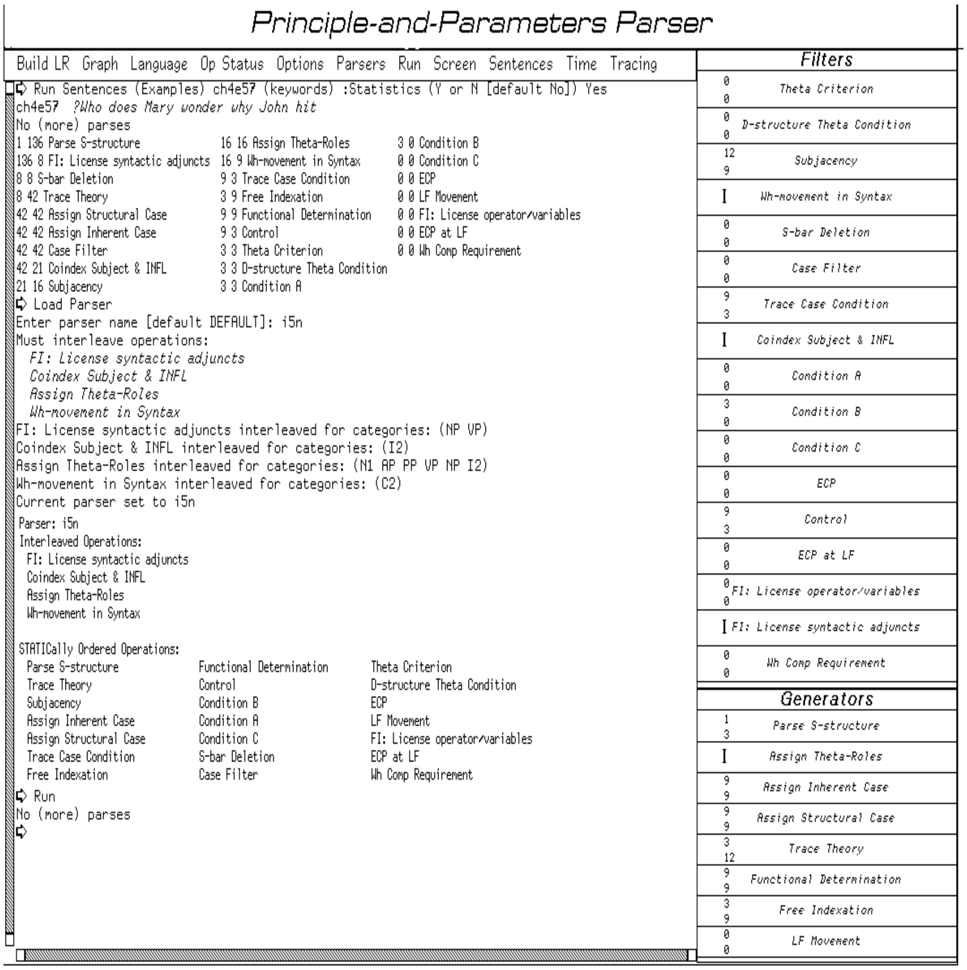


Figure 1.7 Two parsers with different control strategies.

first, followed by *License Syntactic Adjuncts*, then *Trace Theory*, and so on. (Note that the number of calls recorded for each operation is the same as the number of successes recorded for the immediately preceding operation in the chain.) As the record of calls and successes indicates, nineteen out of the twenty-five operations were invoked in the course of rejecting the input sentence.

Next, the snapshot shows the loading of a different parser, i5n. (We will not discuss the significance of the lists of categories shown for each interleaved operation here. A complete explanation can be found in chapter 6.) Here, we only note that the system has been re-configured with four operations interleaved, and the remaining twenty ordered quite differently from the previous configuration. For example, that *Condition B* now appears in front of *Condition A*, *Subjacency* has been moved up four places ahead, and the *Case Filter* has been relegated to the second column. Running the same sentence again, we can see (from the panel on the right side) that only nine operations (compared with nineteen) were consulted this time. (No calls or successes have been recorded for the four interleaved operations (each marked with an 'I' in the right panel) because they have been completely integrated into the *Parse S-structure* operation.) In fact, parser i5n caused parser operations to be invoked a total of 57 times — a considerably smaller number than for the previous case (a total of 415 calls).[21] Using total time taken as a measure of relative efficiency, i5n ran almost nine times faster than the default parser.

This is not a surprising result. Of the nineteen operations invoked by the first parser, only five were useful in the sense that they helped to eliminate ill-formed structures. Most of the remaining fourteen could be eliminated without affecting the overall linguistic judgement since they did no useful work. For example, the *Case Filter* operation eliminated none of the forty-two structures it was passed. In i5n, the *Case Filter* appeared in a position later than the blocking filter, so it was never called. Of the five effective filters, note that three were interleaved in i5n so that they could be applied as early as possible. Moreover, parser i5n also included no operation in a position ahead of the blocking filter that was not either a useful filter, or an operation that was required as a precondition by the dependency restrictions in figure 1.6.

1.5.4 Parser Control for Efficiency

In the following paragraphs, we will review the issues involved in choosing an efficient control configuration. In particular, we will discuss the differences and trade-offs between *dynamic* and *static* ordering, factors that affect the utility of inter-

[21] One caveat: it would be dangerous in general to extrapolate directly from the relative number of calls to the relative time actually spent parsing. In particular, the time spent per call on *Parse S-structure* cannot be directly compared. The fact that four operations have been merged into *Parse S-structure* will affect the unit time per call.

leaving, and the effect of *well-formed* and *ill-formed* input. We will also take the opportunity to summarize results and observations (to be covered in depth in chapters 5 and 6) derived from tests on a large number of examples from Lasnik & Uriagereka.

Principle ordering

- *The filter and generator model.*

  In general, it will prove more efficient to apply the simple strategy of ordering principles so that "filters" and "generators" (in the operational sense defined above) are applied as early and as late as possible (provided data dependencies between principles are respected), respectively. The essential idea being to eliminate hypothesized structures as early as possible. By ordering filters in terms of their relative *filtering power*, i.e. the proportion of structures eliminated, it is often possible to improve upon the default configuration by an order of magnitude or more.

- *Hardwiring a specific ordering.*

  Unfortunately, as will be discussed in chapter 5, factor-of-ten improvements over the default configuration are difficult to maintain. A particular configuration such as i5n may be much more effective for certain examples such as (7), but may be only slightly faster or even slower for others. The basic reason is that different inputs will "exercise" different parser operations. For example, the Case Filter which did no work in the example in the previous section figures prominently as the dominant filter for examples such as **I am eager John to win — cf. I am eager for John to win.*

- *Dynamic ordering.*

  Although, it does not seem to pay to hardwire any one parser configuration; on the other hand, the possibility of a factor-of-ten improvement does suggest it may be worthwhile to consider "on-the-fly" reconfiguration. This leads us to briefly mention one other control option. As chapter 5 will describe, another possibility is to allow the parser to *dynamically* pick its own ordering for each structure generated on the basis of the presence of certain "cues" in a structure instead of imposing a *static* ordering as we have seen here. The key observation here being that most structures generated by the parser will be ill-formed. By examining structural cues, the dynamic ordering mechanism tries to "approximate" the filters by guessing which filter each structure will violate *without* incurring the expense of actually applying the filters.

  The cost-benefit tradeoff that determines the success of dynamic ordering depends on a combination of how reliable structural cues are, and the amount of overhead, both in terms of the cost of making those filter predictions and the extra machinery required for re-organizing the assembly line order. In the current implementation, the goal has been to try to minimize this overhead as much as possible by only making use of cues that require relatively little computation to determine, e.g. testing for the presence of certain syntactic features. However, in general there will be a tradeoff in terms of the simplicity of the approximation, i.e. the cost of computing the approximation, and the reliability of the resulting predictions. Finally, we should also point out that eliminating ill-formed structures is almost as important in parsing well-formed as well as ill-formed sentences. The reason being that, even for well-formed sentences, almost all of the structures hypothesized by the parser must be ill-formed, unless, of course, the input sentence is massively ambiguous.

Principle interleaving

- *On-line interleaving is impractical.*

  As chapter 6 will explain, it will not prove effective to interleave parser operations at run-time because of the overhead involved. More precisely, a typical parser operation has a very limited "range" of application in the sense that many pre-conditions may need to be satisfied before, say, some feature is assigned or some condition is applied. For example, before Case can be assigned: (1) a Case assigner and a noun phrase to receive Case must be found, (2) the assigner must "govern" the receiver which holds only in a certain number of configurations, and (3) the assigner must be adjacent to the receiver (at least in English). In fact, most of the work in applying Case assignment comes from verifying that the above pre-conditions are met. A parser that tries to apply Case assignment to partial phrase structures as soon as possible will spend a considerable amount of effort failing to assign Case.

- *Eliminating the overheads of interleaving.*

  However, if the principle to be interleaved is known beforehand, then a considerable amount of unnecessary testing can be eliminated by using an off-line type inference procedure (described in chapter 6) to identify syntactic configurations that can never be configurations of Case assignment.

  The utility of the interleaving mechanism also depends to a large extent on seemingly unrelated factors such as the relative "efficiency" of the phrase structure recovery mechanism. The pertinent notion of efficiency in this case will be the ability of the parser to detect "dead-ends" in structure recovery (i.e. the fact that the current sentential form cannot be reduced to a single sentential symbol), as early as possible. The current implementation uses a canonical LR(1)-based algorithm (described in chapter 4) to generate phrase structures. It is well-known that canonical LR(k) parsers are guaranteed to detect erroneous sentential forms as early as possible given k symbols of terminal lookahead. However, one surprising result of empirical testing is that, even with a low intrinsic overhead, it is more efficient *not* to interleave for many parser operations.

This concludes our discussion of the control options of the system. In the next section, we will turn to discuss implementation issues.

1.6 The Implementation: Grammar Representation and Compilation

Ihis section will briefly review how the grammar, i.e. the phrase structure rules and principles, is encoded and subsequently turned into parser operations. The section is divided in three parts:

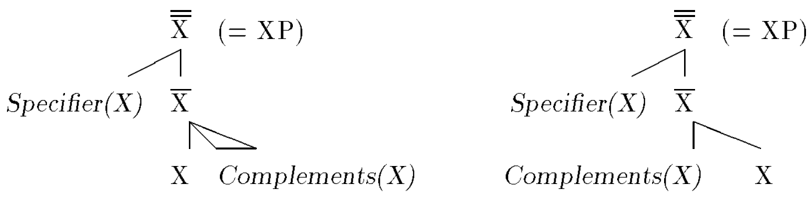
1. The representation of phrase structure rules.
2. The representation of principles using various primitives defined on constituent structure.
3. The organization of the various sub-systems that turn grammar and control definitions into parsers.

1.6.1 Phrase Structure Rules

This section provides a brief overview of the representation used for the phrase structure component of grammar. Chapter 4 contains a complete description of the representation used and its compilation into canonical LR(1)-based tables.

$\overline{\text{X}}$-theory

In $\overline{\text{X}}$-theory, all phrases have the same basic internal structure. Two forms of a typical $\overline{\text{X}}$-prototype for a generic phrase are shown below:



(English) (Japanese)

As discussed earlier in section 1.4.1, Japanese is classified as a SOV language and thus requires a different order of phrasal components from English. In either case, X ranges over the usual categories N (noun), V (verb), A (adjective) and P (preposition), with extensions to include sentences (S) and full clauses ($\overline{\mathrm{S}}$). The lowest and highest levels corresponds to the traditional levels of individual lexical items and complete phrases, respectively.

In the current system, a small number of non-ground rules headed by the following two rules is used to represent the $\overline{\mathrm{X}}$-prototype:

```
rule XP -> [X1|specifiers(X1)] ordered specFinal
                                st max(XP), proj(X1,XP).

rule X1 -> [X|complements(X)] ordered headInitial
                               st bar(X1), proj(X,X1).
```

> Some PROLOG notation: logical variables are distinguished from constants by the use of an uppercase letter as the initial character. The scope of a variable is a clause.
>
> Elements delimited by square brackets denote lists. In particular, `[]` denotes the empty list. `[Y]` denotes a list with one element `Y`, `[Y1,Y2]` a list of two elements, and so on. `[X|L]` denotes a list with head `X` and tail `L`.

Here, `specFinal` and `headInitial` are binary language parameters that determine the relative order of the right-hand-side elements. The possible values for `XP`, `X1` and `X` are defined by the predicates `max/1`, `bar/1` and `head/1` as follows:

```
% Noun   Verb     Adj.     Prep.    S=i2     S̄=c2     Level
head(n). head(v). head(a). head(p). head(i). head(c). % X
bar(n1). bar(v1). bar(a1). bar(p1). bar(i1). bar(c1). % X̄
max(np). max(vp). max(ap). max(pp). max(i2). max(c2). % X̿

proj(X,Y) :- head(X), bar(Y).
proj(X,Y) :- bar(X), max(Y).
```

> More PROLOG notation: commas (`,`) outside lists denote conjunction. `:-` (to be read as 'if') denotes implication.

Another eight rules (not shown here) suffice to describe the appropriate choices of specifiers (`specifiers(X1)`) and complements (`complements(X)`) for each category. For completeness, we should mention that an additional twelve to twenty rules (also not shown here) will be required to cover empty categories and adjunction structures.

1.6.2 Principles

A variety of formalisms have been used to encode linguistic principles, including Attribute Grammars (Correa [16]), PASCAL (Wehrli [58]), LISP (Kashket [32] and Dorr [20]), PROLOG (Crocker [17], Johnson [31] and Sharp [48]). As mentioned earlier, Stabler has described a theorem-proving system based on a first order logic formulation, but that system is not a parser. However, if theory maintenance is to become practical, we believe that it will be necessary to choose a representation that is as close as possible to the descriptions found in the linguistics literature. This means that the representation should provide the essentially the same "vocabulary", e.g. primitives and combinatory elements, used by linguists so that the formalization process can become as close to a simple one-to-one mapping as possible. In the current representation, there are several compromises due to implementation considerations (to be described in chapter 3). However, we believe that many of the definitions used in the current implementation are straightforward translations of the original definitions. We will illustrate this with a few examples. Many other examples can be found elsewhere in the main chapters of the thesis. In particular, chapter 3 discusses the representation in considerable detail. Chapter 6 also covers others in the context of principle interleaving. All definitions including the linguistically-motivated primitives have been written in PROLOG, a programming language based on a subset of 1st order logic. Whilst PROLOG has its disadvantages as a grammar representation (which will be discussed in chapter 2), we believe that perspicuous definitions that are readily understandable by linguists can be written.

The Case Filter

We will begin with perhaps the simplest principle implemented in the current system. Here is the statement of the filter given in Lasnik & Uriagereka:

(16) At S-structure, every lexical NP needs Case. *(from ch.1, no.20)*

Case, e.g. nominative, accusative, genitive and oblique, is normally assigned to all *lexical*, i.e. non-empty, NPs. For example, in English objects of verbs seem to take accusative Case, *John saw him/*he*. We will not go into the details here (see chapter 3), but various facts such as the distinction between *it is likely that John is here* and **it is likely John to be here* can be explained using the Case Filter in conjunction with the conditions of Case assignment which permit lexical NPs to receive Case only in certain configurations. Compare (16) with the following formulation:

```
:- caseFilter in_all_configurations CF where
       lexicalNP(CF) then assignedCase(CF).
```

```
lexicalNP(NP) :- cat(NP,np), \+ ec(NP).
assignedCase(X) :- X has_feature case(Case), assigned(Case).
```

Some PROLOG notation: (`\+`) denotes negation as failure (to prove).

The first statement represents a universally quantified formula over constituent structure. It is meant to be read as follows: "in every configuration `CF` where `lexicalNP(CF)` holds, then `assignedCase(CF)` must also hold." The next two clauses define the meaning of the terms *lexical NP* and (to be) *assigned Case*:

1. The first clause states that:

   > "the category `NP` is a lexical NP if it has the category label `np`, and it is not an empty category (`ec/1`)."

   Both `cat/2` and `ec/1` are primitives in the system.

2. The next clause also makes use of another two primitives `has_feature/2` (which tests whether a constituent has the feature in question) and `assigned/1` (which tests if some feature slot has been bound). Hence, `assignCase/1` can be read as:

   > "`X` is assigned Case if `X` has a Case feature with slot `Case` and that slot has been filled."

Note that in standard PROLOG, conjunctive goals are executed from left to right. This is unimportant for `lexicalNP/1`. However, `assignedCase/1` relies on the Case slot being identified before it is tested. Finally, we have assumed here that the structures passed to `caseFilter/1` will be from the level of S-structure.

Binding Condition A

We will now sketch how Binding condition A, which constrains the possible coreference options for anaphors, may be perspicuously encoded. Only the top-level portions of the relevant definitions will be supplied here, a more complete definition will be given later in section 3.5.

The following statement of the condition given in Lasnik & Uriagereka is a more sophisticated version of the one introduced at the beginning of the chapter:

(17) An anaphor must be (A-) bound in its governing category (GC).
(from ch.2, no.27a)

The notion of a *governing category* is defined as follows:

(18) The GC of α is the minimal NP or S containing α, a governor of α, and a SUBJECT accessible to α. *(from ch.2, no.105)*

Compare (17) and (18) with the following formulation:

```
:- conditionA in_all_configurations CF
        where anaphor(CF) then gc(CF,GC), aBound(CF,GC).

anaphor(NP) :- cat(NP,np), NP has_feature a(+).

aBound(B,S) :- binds(A,B,S), A has_feature apos.

gcDomain(i2). % i2=S
gcDomain(np).

:- gc(X) smallest_configuration CF st cat(CF,C), gcDomain(C)
        with_components
                X,
                G given_by governs(G,X,CF),
                S given_by accSubj(S,X,CF).
```

The definitions given above should be fairly self-explanatory. For brevity, we have omitted the definitions of the linguistic relations `binds/3`, `accSubj/3` and `governs/3`. The only significant additions are the following:

- *Disjunctive definitions.*

  The disjunction in (18), i.e. "...NP or S ...," has been represented using the two clauses of `gcDomain/1`. This introduces a subtle but important discrepancy between the two definitions. The PROLOG definition implies that the category label of a GC domain must be *either* `i2` *or* `np`. On the other hand, the natural language definition has the possibility of being interpreted as an inclusive-or; in particular, allowing the possibility of having both disjuncts simultaneously satisfied. In this case, there is no effective distinction because linguistic categories have unique labels. However, this is a potentially important difference because Horn clause logic (on which PROLOG is based) cannot represent such inclusive-or disjunctions (that can be represented in 1st order logic). We will return to consider whether this is an important limitation for the representation of principles in chapter 2.

- *Minimality.*

  The `smallest_configuration` form implements definition (18). It states that:

  > "The GC of `X` is the smallest configuration `CF` such that `CF` has category label `C` and `C` is a GC domain, and `CF` contains three com-

> ponents: (1) `X` itself, (2) `G` where `G` governs `X` in `CF`, and (3) `S` where `S` is a SUBJECT accessible to `X` in `CF`."
>
> This statement defines the binary relation `gc/2` referenced in the definition of condition A. In general, `gc(CF,GC)` will hold if `GC` is the GC of `CF` as specified above.

Observe that nowhere in the definitions presented have we specified how a principle should be ordered or whether it should be interleaved. Those decisions need not concern the grammar writer since the intended semantics of the definitions are independent from such control considerations.[22] This concludes our overview of the representation used for phrase structure rules and principles. In the next section, we will describe the architecture of the underlying system.

1.6.3 System Organization

The grammar writer need only be concerned with a fraction of the underlying system. For example, the particular phrase structure recovery algorithm used and details of how changes in the control parameters are reflected in the underlying architecture have been (deliberately) kept hidden as much as possible. In this section, we will briefly describe each of these components in turn.

Figure 1.8 shows the major components of the implemented system. The basic inputs to the system as provided by the grammar writer are shown on the right-hand-side. The "output" of the system is a parser specialized with respect to: (1) the linguistic theory in the form of the principles and S-structure rules, (2) the particular language in the form of the parameters and lexicon (plus part of the S-structure rules), and (3) the parser control specified by the user. Note that the system initially recovers phrase structure at the level of S-structure (rather than at Logical Form or D-structure — the other two syntactic levels described earlier in section 1.4.1) for efficiency reasons to be outlined in chapter 4.[23] One disadvantage to specialization is that portions of a parser may have to be re-built if any of these elements change. In the current system, adding a new entry to the

[22] Actually, not all the principles we have described can be interleaved. In particular, any principle that operates on LFs cannot be interleaved with phrase structure construction because the system (initially) recovers phrase structures at the level of S-structure.

[23] Although the S-structure rules are parameterized with respect to head–complement and specifier–head order, there still remains some work to be done with respect to movement before the phrase structure rules can be truly abstract. For example, as described earlier one of the major differences between Japanese and English is that *wh*-movement is reflected at S-structure in English but not in Japanese. In the current system, the possible consequences in terms of what can appear where at S-structure are still spelled out manually in the rule set. (See chapter 4 for the details.) The solution is, of course, to have the grammar writer only specify the rules of D-structure and add an extra compilation step that automatically computes the possible S-structure rules given an appropriate setting of the whInSyntax parameter.

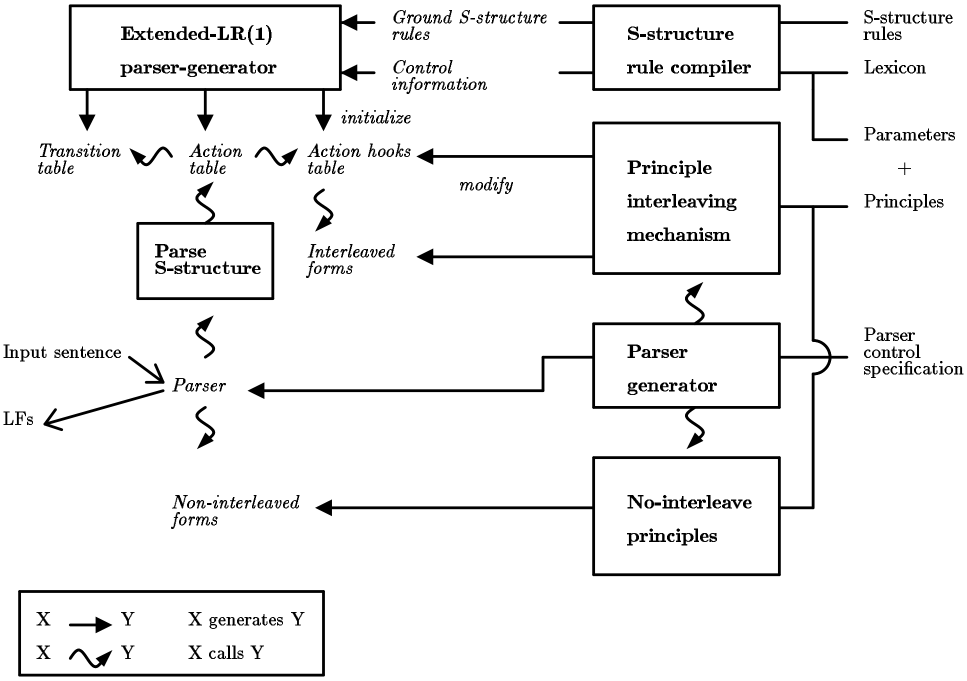


Figure 1.8 The components of the system.

lexicon could require re-computing the entire LR(1)-based automaton. Note that the system makes use of a variety of internal representations (shown in italics) that are normally not visible to the user or grammar writer. We will now briefly explain the significance of each of these:

1. *The phrase structure components.*

   The rules that encode the phrase structure component will be "grounded" with respect to the constituent order parameters [±`specInitial`,±`headInitial`]. Also, $\overline{\text{X}}$ phrase structure rules (introduced previously in section 1.6) will be grounded with respect to particular choices for complements and specifiers for each category. (The compiler will also detect when some of these choices are unavailable for some categories due to the lack of lexical entries of the right type.) The resulting set of ground rules together with extra control information for disambiguation will be input to a canonical LR(1)-based parser generator. The generator produces LR(1)-style transition and action tables. The major differences in the action table are (1) multiple actions for table entries are permitted, and (2) each action may have pre-conditions attached.

These pre-conditions are accessed indirectly via a mapping table *Action hooks table* that can be dynamically modified by other components of the system (as we shall see below).

The phrase structure components have been designed to be modular. It is a simple matter to replace the "back-end" (i.e. the LR(1)-based parser-generator), with an alternate parser-generator using a different algorithm. For example, in an earlier version the intermediate representation, i.e. the ground rules, was not compiled, but simply interpreted.

2. *The principle compilation components.*

   The parser control specifies which principles are to be called independently or interleaved with structure building. The compiler will generate the appropriate forms in either case:

   (a) If a principle is not to be interleaved, a parser operation that accepts a complete phrase structure as input will be generated. This operation will be directly invoked by the top-level parser control (to be described below).

   (b) Otherwise, when a principle is to be interleaved, a predicate that accepts only partial structures as input will be generated. The type inference mechanism will then compute where to "splice in" links to the generated predicate in the LR(1) action table. Note that only one copy of the action table is ever generated. That is, the interleaver does not directly modify action table entries, but instead inserts the relevant calls via the mapping table *Action hooks table*. Note also that this implies that interleaved principles will not be directly called by the parser control. As the dependencies in the system diagram indicate, parser control invokes the operation *Parse S-structure* which will call the appropriate action table entries which, in turn, will invoke the interleaved forms (via the mapping table).[24]

3. *The parser control component.*

   Given a parser control specification, the parser control component generates a top-level control predicate *Parser* that simply calls *Parse S-structure* to obtain phrase structures and passes them to the non-interleaved forms in the order specified.[25] If the non-interleaved forms are to be dynamically ordered, the top-level predicate also calls a secondary control predicate (not shown) for

[24] Note that *Parse S-structure* must also accesses the lexicon (the link is not shown in the system diagram).

[25] The actual implementation is a bit more complicated. Each principle is actually called indirectly via a mapping table similar to that for the LR(1) action table. Generally, there will be a

each structure recovered by *Parse S-structure*. This control predicate manages the list of non-interleaved operations and invokes the operations in the order most likely to rule out the given structure.

4. *The parser.*

   The top-level predicate generated by the parser control component is a two-place predicate that accepts a sentence from the user and returns zero or more Logical Forms.

This completes our review of the components of the system. In the next and final section of the chapter, we will briefly outline the topics covered in each of the remaining chapters.

1.7 Roadmap of the Thesis

The main body of the thesis consists of the following six chapters:

- *Parser Design.*

  Chapter 2 examines the important design decisions and points out where necessary tradeoffs have been made in the current implementation. For instance, it considers the consequences of adopting a simple representation of linguistic principles built on top of PROLOG. The chapter will also make use of comparisons with other principle-based systems to highlight particular design points.

- *The Representation of Linguistic Principles.*

  Chapter 3 describes how linguistic principles are represented and identifies the limitations of the current representation using case studies involving Case theory, chain formation, free indexation and Binding theory.

- *The Recovery of Phrase Structure.*

  Chapter 4 describes how the system copes with multiple levels of phrase structure, ambiguity, and infinite structures. It also documents how the phrase structure rules are compiled into an efficient canonical LR(1)-based machine by specializing the rules with respect to the language and the lexicon. Also, the suitability of the canonical LR(1)-based procedure and the effect of varying language parameters will be examined. Comparisons will also be made to

---

variety of preconditions and postconditions defined on each principle including the updating of various counters of the type described earlier in the chapter, calls to display the structure passed to the principle — for debugging purposes, extra control that the user can specify, e.g. to just return the first parse instead of looking for all solutions, to return only (or no) structures satisfying some form, and so on.

the LALR(1) procedure, a model that trades the amount of disambiguation possible from lookahead for reduced table size.

- *Principle Ordering.*

  Chapter 5 considers the question of ordering principles for efficient parsing. More precisely, the following questions will be addressed: (1) Will re-ordering principles make a big difference in parsing speed? That is, is principle ordering something parser designers should even be worried about? (2) If so, which orderings are better than others? Can we explain why? (3) Finally, is there an "optimal" ordering of principles that parsers should adopt? Empirical data will be presented to validate the adopted computational model of principles as simple filters and generators. The chapter will also describe and evaluate the use of an ordering mechanism that attempts to dynamically specialize the filters-and-generators model with respect to particular inputs.

- *Principle Interleaving.*

  Chapter 6 will describe the type inference procedure and other components that make up the off-line principle interleaving mechanism. The notion of the size of the type of a principle will be introduced as a basis for predicting the effectiveness of interleaving on particular principles. Empirical data will be presented to correlate the type size with parsing efficiency and to determine which principles are appropriate candidates for interleaving.

- *Conclusions.*

  Chapter 7 discusses the lessons learned from building the system, things that should have been done differently, and points out areas where future work could be profitably applied.

Finally, we will briefly outline the contents of the two appendices also included in the thesis:

- *The Linguistic Theory.*

  Appendix A contains a compact, but complete description of the linguistic theory that was formalized. Since we do not repeat the linguistic arguments for each principle listed, it is not meant to be a substitute for Lasnik & Uriagereka. We have attempted, however, to provide a reference (where possible) to the original source for each definition.

- *The Notation for Encoding Principles.*

  Appendix B contains a complete description of the "programming language" (i.e. the linguistically-motivated primitives and combinative elements), that were used to encode the theory described in appendix A. Type equations or type inference rules used by the principle interleaving mechanism will also accompany all appropriate entries.

Chapter 2

Parser Design

The main chapters of the thesis will be devoted to describing the specific features and components of the implemented system. The purpose of this chapter will be to discuss general design issues for principle-based parsers and to highlight the differences between this system and the approaches taken in other principle-based systems. The remainder of the chapter will be organized around the following topics:

- *Modifying linguistic theory.*

  We will discuss various reasons why grammar writers might want to modify linguistic definitions. We will also outline the position taken in the current system as to the circumstances under which modifications can be made.

- *The level of representation of linguistic principles.*

  We will discuss the consequences of adopting PROLOG as the underlying language for the representation of principles. In general, there seems to be a tradeoff between the abstraction level of the representation and the efficiency of the resulting parser. In the context of this tradeoff, we will consider the question of whether a more powerful formalism such as 1st order logic, as adopted by Stabler [52], is really necessary in order to parse with principle-based theories.

- *Free overgeneration, separation of principles, and efficiency.*

  Next, we will discuss the reasons for, and the problems of, adopting a policy of maintaining a strict separation between principle definitions in the present system.

- *Derived principles.*

  Correa's attribute grammar-based formulation ([16]) is an example of a parser that has not imposed a strict separation between principles. Because Correa's parser and the system described here are based on a common theory of functional determination, we shall directly compare our representation to Correa's reformulated version.

- *Licensing parsers.*

  We will briefly discuss the concept of a *licensing parser*, i.e. a parser that only build structures in accordance with a distinguished subset of principles, as exemplified by Abney [2] (also Frank [24]). We will relate part of Abney's system to ours by illustrating how the control of the current system can be set up in a similar fashion so that the phrase structure recovery component only recovers structures satisfying a designated subset of principles.

- *Methods for efficient parsing.*

  Next, we will briefly survey some of standard compilation and interpretation methods that have been used for parsing. In each case, we will point out the specific advantages and disadvantages of the method for principle-based parsing, and whether the method is utilized in the current implementation.

- *Other issues.*

  Finally, we will finish the chapter with by briefly discussing the role of the notions *directness* and *faithfulness* in the current implementation.

2.1 Modifying Linguistic Theory

- *Linguistic theory is always changing.*

  In recent years the principle-and-parameters framework has proved to be a very productive area of research in linguistics. It is probably safe to assume that linguistic theory will continue to be updated at a fairly rapid rate in years to come. This state of affairs has obvious important implications for parser design. Given the inevitable lag in time that occurs between publication of a theory and its possible realization, most implementations are probably obsolete before they are completed, or perhaps even begun. As we will see below, this will lead us to reject certain techniques such as hand-optimization of principles in favour of less efficient, but more easily maintainable representations.

- *Linguistic theory should not be changed.*

  It can be tempting for the grammar writer to alter principle definitions for a number of reasons:

  1. Principles are usually not written in the most efficient form for computation. In some cases such as *free indexation* (the assignment of indices

to establish coreference relations), as will be discussed in chapter 3, definitions may not be directly *constructive*. That is, principles may not be stated in a form that immediately suggests a corresponding computational procedure.

2. Some linguistic theories may make use of powerful operations on phrase structure that may lead to parser non-termination. A good example of this is the γ-marking proposal of Lasnik & Saito [34]. This scheme makes use of an operation that allows certain elements in phrase structure to move to another position (and possibly back again) without generating an trace. Thus, it becomes possible to derive certain phrase structures in an infinite number of ways.

3. Restatement may also be forced by peculiar limitations in the underlying architecture of the parser. For example, the current system makes extensive use of enumerative procedures that generate all possible hypotheses to be filtered out at a later stage. A problem occurs if an enumeration procedure defines an infinite number of hypotheses. Although, it may be the case that there are only a finite number of valid hypotheses, i.e. all but a finite number of hypotheses will be filtered out at some later stage, the simple generate-and-test strategy will not terminate. For example, the Verb Phrase (VP) adjunction operation (introduced in the previous chapter) may be used to implement the 'scrambling effects' present in languages such as Japanese. Unrestricted application of VP-adjunction will result in an infinite number of (distinct) phrase structures, e.g. $[_{VP}\ [_{NP}\ \textit{nani-o}\ ]_i\ [_{VP}\ t_i\ \ [_{VP}\ t_i\ \ldots\ [_{VP}\ t_i\ \textit{katta}\ ]]]]$ can be generated from $[_{VP}\ [_{NP}\ \textit{nani-o}\ ]\ \textit{katta}\ ]$.

Hence, an important question to ask is: Under what circumstances should it be acceptable to recast definitions to better suit parsing needs?

This thesis takes the position that the grammar writer should avoid changing the definition of a principle unless it is crucial for ensuring parser termination, as in the case of VP-adjunction and γ-marking. In particular, we will argue below that rewriting an already terminating operation simply for efficiency should not be done manually, but only if there exists some well-defined automatic and transparent procedure that accomplishes the task. We are assuming, of course, that systems constructed in this fashion should be more robust. Also, optimizations for efficiency are less likely to prove to be idiosyncratic (and transient) in nature when the principles are modified.

We should mention that one limitation that results from forcing parser operations to be finitely enumerable is that correctness of the restricted operation is not guaranteed. For example, the current system allows only one iteration of repeated

VP-adjunction.[1] So [*VP* [*NP* *nani-o*]$_i$ [*VP* t_i *katta*]] would be allowed, but not [*VP* [*NP* *nani-o*]$_i$ [*VP* t_i [*VP* t_i *katta*]]]. Hence, the parser will (incorrectly) miss a parse if there happens to be one that involves two consecutive traces of this type. Basically, what is needed is some sort of *inductive proof capability* in the sense of showing that an ill-formed parse cannot be 'repaired' by (repeated) VP-adjunction.[2] The current system lacks this capability.[3]

2.2 The Level of Representation of Principles

We have found PROLOG ([51]) to be a reasonable compromise between a completely declarative representation such as 1st order logic (FOL) (for which practical general search strategies do not exist) and a more efficiently executable but less abstract model such as that of PASCAL or LISP. By building a suitable collection of predicates that operate on constituent structure, as chapter 3 will show, it is possible to produce executable specifications with little effort for many principles. However, PROLOG does have several flaws that make themselves felt in the principle definitions we have used, thereby preventing it from being an ideal representation language. We will briefly list the major ones:

1. PROLOG is based on Horn clause logic (a limited subset of FOL). In particular, Horn clauses are restricted to 1st order formulas of the form $\forall x_1, \ldots, \forall x_n(L_1 \wedge \neg L_2 \wedge \neg L_m)$ where $x_1, \ldots, x_n$ are the variables occurring in the formula, and $L_1, L_2 \ldots, L_m$, $m \geq 1$, are all positive literals, i.e. no negated atoms. (PROLOG shorthand for the above formula is $L_1 : -L_2, \ldots, L_m$.) Because of the stipulation that all Horn clauses must have at exactly one positive literal, Horn clauses cannot be used to express negative information, e.g. $\neg$*whInSyntax* would not be a Horn clause. However, the same stipulation allows the use of the efficient SLD-resolution (Linear resolution with Selection function) strategy (which has been shown to be both sound and complete).

[1] The LR(1)-based parser-generator automatically deals with simple cases of infinite loops. The current system detects and repairs simple cases in the CFSM (characteristic finite state machine) when it is possible to loop back to the same state without consuming any input, as will sometimes occur with λ-rules such as NP$\rightarrow$ λ or Adv$\rightarrow$ λ. (See the screen snapshot in chapter 7 for an example.) The default 'repair' is to only allow it to loop once. This parameter can be altered on a global or state by state basis to allow an arbitrary upper limit.

[2] Also see section 12 in Stabler [52].

[3] A speculative comment is in order here. Note that a system that implements a constraint propagation scheme may be able to avoid having to impose the finite enumerability restriction, and therefore also avoid having to implement the inductive proof tactic. More concretely, a forward pruning strategy (see van Hentenryck [28]) may be able to eliminate the infinite branches in the search space corresponding to, say, repeated VP-adjunction. We have not investigated the application of this technique in principle-based parsing.

2. PROLOG does not properly implement SLD-resolution. Standard PROLOG systems have two major defects. (1) The full unification algorithm (with the occurs check) is not usually implemented. The consequence is that soundness is not preserved. (2) PROLOG uses an unfair computation rule, namely depth-first search. The consequence is that completeness is not preserved.

3. PROLOG does not properly implement NAF (Negation as Failure). PROLOG allows Horn clauses to be extended with negated goals in the body of a clause, e.g. $L_1 : -\backslash\text{+}L_2, \ldots, L_m$ where '$\backslash\text{+}L_2$' represents a negated goal. It is well-known that SLDNF-resolution (SLD-resolution with NAF), Lloyd [36], is a sound and complete computation rule. However, standard PROLOG systems only implement an unsound version of NAF.

The reason why standard PROLOG implementations have so many compromises is that it is much more efficient *in practice* to omit the occurs check, to use depth-first search (rather than, say, breadth-first search), and to implement a non-safe negation rule.[4] However, this efficiency comes at the cost of the grammar writer having to worry about procedural details to avoid getting into trouble. For example, the grammar writer can avoid the incompleteness of depth-first search and the unsoundness of PROLOG negation in many cases by re-ordering goals. However, it is unreasonable to expect potential grammar writers to deal with such defects.

On the other hand, better PROLOG implementations, in the sense of having fewer compromises with respect to soundness and completeness, whilst not yet in widespread use, do exist. For example, there are systems, e.g. Naish [40], that automatically delay negated goals until all unbound variables have been instantiated (an example of a sound implementation of NAF). The current principle definitions can be interpreted directly by such systems without any changes. Hence, we conclude that many of the present deficiencies in PROLOG are not fundamental, but simple artifacts of current implementations that will probably be remedied in the near future.

However, the restriction of Horn clause logic to a single positive literal is potentially a more serious limitation. The natural question to ask is: Are there are any instances of principles where two (or more) positive literals are required? Equivalently, do we ever need to express something of the form, say, $p \vee q \leftarrow r$? Stabler [52] suggests one example of such a situation:

(1) moveA(Tree0,Tree) $\leftrightarrow$ (substitute(Tree0,Tree) §*6(7)*
$\vee$ adjoin(Tree0,Tree)
)

[4] We have ignored the so-called 'extra-logical' features such as the 'cut' that can also cause problems for completeness.

The $\rightarrow$ direction of rule (1) contains two positive literals. Note that the following two PROLOG clauses are *not* equivalent to the $\rightarrow$ form:

(2)
```
substitute(Tree0,Tree) :- moveA(Tree0,Tree).
adjoin(Tree0,Tree) :- moveA(Tree0,Tree).
```

The reason, of course, is that the two clauses in (2) constitute an *exclusive-or* disjunction, whereas the 1st order logic statement allows the possibility that both disjuncts might be simultaneously satisfied, i.e. an *inclusive-or*. Hence, at first glance, it would seem that the PROLOG representation is inadequate. However, it turns out that (1) is an incorrect formalization of the English definition in *Barriers* Chomsky [11]. From pg.4, we have:

(3) "Assume that there are two types of movement: substitution and adjunction."

In (3), the *intended* interpretation is that substitution and adjunction are *distinct* operations. Hence the PROLOG definition *does* capture the intent of Chomsky's definition. Moreover, an informal survey of the 108 1st order sentences in Stabler's axiomatization of *Barriers* suggests that the intended interpretation of all such disjunctive statements is the exclusive-or one. Finally, there also seems to be no need for inclusive-or disjunctions in the definitions employed in the current system. To conclude, there seems to be no need to go to a more powerful representation (and pay the associated performance penalty) than Horn clause logic extended with NAF.[5]

2.3 Free Overgeneration, Separation of Principles and Efficiency

We have chosen to use as straightforward and direct an implementation of principle definitions as possible. In particular, each principle has been encoded separately. We have deliberately chosen not to attempt to combine principles or allow one principle to, partially or otherwise, abrogate another purely for the sake of efficiency. Depending on the linguistic theory, this can result in a very inefficient parser. For example, in the type of theory assumed by Lasnik & Uriagereka, the basic model is

[5] Stabler also suggests other theoretical reasons for using FOL. He observes (§2.9) that there may be sentences *which are not in the completion of any Horn theory.* He also points out the possibility that selective SLDNF computation rule may abort *so much as to make the theorem proving system worthless.* Finally, he also notes that the NAF rule can fail to terminate. However, despite these theoretical limitations, it is not clear that a non-contrived, linguistically-motivated example that illustrates these deficiencies can be found.

one that allow certain operations such as movement (Move-α) to operate as freely as possible; for example, to allow any syntactic element to move anywhere. (See also section 1.2.) The essential assumption here is that principles will interact so as to eliminate all but the well-formed instances of movement. Even standard assumptions about movement such as "traces are obligatorily left by movement" and "a trace and an antecedent are obligatorily coindexed" are considered to be stipulations best reduced from being obligatory to being merely optional, with other principles forcing the stipulation to hold only when necessary. In this way, the theory becomes a stronger one with fewer redundancies because (optional) stipulations will have independent justification.

Let us briefly consider the *LF movement* operation, an *optional* process (motivated by semantic interpretation considerations) that moves quantifiers and *wh*-elements to positions of sentential scope at LF. For example, *everyone* and *nani* ('what') both undergo movement at LF in (4a) and (4c) (= (11) in chapter 1), respectively:

(4) a. John saw everyone
    b. everyone$_i$ [John saw x_i] (LF)
    c. watashi-wa Taro-ga nani-o katta ka shitte iru
       (= I know what Taro bought)
    d. watashi-wa nani$_i$ [Taro-ga x_i katta] ka shitte iru (LF)

In general, such elements, unless they appear in sentential scope positions, will be ruled out by principles that operate at LF, such as the *wh*-Comp requirement (which requires [+*wh*], i.e. questioned, clauses to have *wh*-elements head them) or perhaps operator–variable licensing (operators must Bind some variable). Hence, one temptation when encoding LF movement is to make movement obligatory, say, for all *wh*-elements and quantifiers not already in positions of sentential scope. This would avoid much unnecessary work from ruling out obviously ill-formed LFs. But we take the position that this is wrong, not only because it changes the theory and introduces redundancy, but it is also dangerous in the sense that the formulation of the *wh*-Comp requirement, for example, will not be properly tested. This can lead to insidious problems when the theory is updated. For example, if the *wh*-Comp requirement is incorrectly altered for some reason, the parser will continue to produce the correct parses (for the wrong reason). Admittedly, this is a relatively trivial example.[6] However, as we shall see below, reformulation has led to major (probably unintended) deviations from the theory for at least one existing parser.

[6] Incidentally, in the current system, the theory as implemented is underconstrained. The LF principles fail to force both *wh*-elements in *kimi-wa nani-o doko-de katta no* (example (60) from figure 1.3) to move to Comp. This discrepancy would not have been apparent if LF movement had been stipulated to be obligatory for *wh*-elements not in Comp.

It should be pointed out that we should have no objection to the alteration of principles for efficiency reasons provided that such operations can be carried out automatically and transparently, i.e. without missing well-formed parses, accepting ill-formed sentences, or altering the structures that the user sees in any way. Unfortunately, the current system (or any other existing parser as far as we are aware) cannot automatically deduce theorems such as: *wh*-elements must move to position of sentential scope (if they are not already at such a position). Because of the lack of such a compilation procedure, the *LF movement* operation in the current system faithfully mimics the *free overgeneration* that is a *property* of the linguistic theory described in Lasnik & Uriagereka. However, principle-based theories need not have this freely-generate-and-test characteristic. For example, Chomksy [14] has described a theory that explains important properties of verbal inflection (that we will not cover here), but relies on a 'guiding' principle to prune the space of possibilities for Move-α. The essential difference is that the guiding principle is a 'least effort' condition on derivations. This can considerably simplify the problem for parsing. That is, we no longer have an obligation to fully explore the space of possible derivations, but simply to produce all the derivations of shortest length.[7]

However, many existing parsers have been designed with no such prohibition against combining or reformulating principles. Such systems will avoid the performance penalty due to overgeneration that occurs in the system described here. We will now turn to discuss one example of such a system.

2.4 Problems with Derived Principles

Correa [15] (also [16]) describes an attribute grammar formulation that has several interesting features. One of the features that Correa's parser has in common with the one described in this thesis is that both parsers employ a theory of empty categories (ECs) based on *Functional Determination*. In this theory, empty noun phrases are initially devoid of syntactic features. However, their distribution are restricted depending on whether or not they are classified as anaphors or pronominals (or both or neither). For example, if an EC is classified as an anaphor, then just like an overt anaphor such *himself*, it must be subject to Binding condition A, as discussed in chapter 1. The Binding status of such elements is indicated by the values assigned to two features [±A] (anaphoric) and [±P] (pronominal) — discussed earlier in section 1.2. Let us examine in more detail how these features are assigned.

Normally, these values are determined by the function that the EC plays in syntactic structure. For example, if an EC is bound by a *wh*-element (an operator)

[7] Note that the actual theory is slightly more complicated because not all derivation steps have the same 'cost'.

Trace Theory	
(non-deterministic)	
↑	
Free Indexation	*Case Assignment*
(non-deterministic)	(deterministic[8])
↑	↑
Functional Determination	*Structural Determination*

Figure 2.1 Operational dependencies for functional and structural determination.

occupying a clause-initial position, as in *which report*$_1$ *did you file* e_1, then the EC, e_1 in this case, should be determined as a variable ([$-$A,$-$P]). The precise details of how the assignments are made will not be important for the purpose of this discussion. We will just note that functional determination is a complex procedure that depends on many factors, including: (1) whether the EC is bound or not; (2) the position that the binder occupies if the EC is bound, and (3) whether the bound EC is a trace formed originally by movement of the binder.

For parsing, this implies that a great deal of information must be recovered before functional determination can be applied. In particular, we should apply the principles that help to identify which ECs are bound to what elements. Consider again the dependency graph that was shown earlier in figure 1.6. Here, each Binding condition, *A*, *B* and *C*, is directly dependent on *Functional Determination*. In turn, *Functional Determination* is dependent on *Free Indexation* (an operation that can establish Binding relations by freely coindexing, i.e. assigning the same indices to, syntactic elements). *Functional Determination* is also dependent on *Trace theory* (an operation that can also establish Binding relations by recovering trace-antecedent relations, i.e. the occurrences of movement). (No other operation can establish Binding relations, so *Functional Determination* does not depend on anything else, apart from *Parse S-structure* to recover phrase structures.)

However, Correa's parser uses a reformulation of functional determination, called *Structural Determination*, that assigns [$\pm$A,$\pm$P] features to ECs without needing to identify the possible binders first. Instead, these features are assigned by inspecting properties of the EC itself. (The properties used are whether the EC is: (1) in an A-position or not, (2) governed or not, and (3) assigned Case or not.) This reformulation greatly simplifies the computation of Binding features for ECs since there is now no need to recover the trace-antecedent relations or do free indexation first. (The relevant dependencies for both operations are summarized in figure 2.1.)

[8]Not strictly true. See the discussion of optional Case assignment in section 3.3.2. However, Case assignment is relatively deterministic compared to trace theory and free indexation.

Moreover, this also reduces the amount of non-determinism in the system since, in general, there will be many assignments of trace-antecedent and coindexing relations compatible with any particular structure. However, we do not adopt the (more efficient) structural determination approach for the following reasons:

1. *Problems with theory maintenance.*

   Correa is able to bypass having to compute coindexing relations because he has pre-computed whether various assignments of [$\pm$A,$\pm$P] are allowed for certain positions. For example, one of the cases of structural determination is the following rule:

   (5) If an EC occurs in an A-position, and is assigned Case, and is governed; then it *must* have features [$-$A,$-$P].

   Any other assignment of [$\pm$A,$\pm$P] will be ruled out by some filter. To reach this conclusion, information from many other principles such as the Case filter and the the various Binding conditions must have been applied. In other words, (5) must be a logical consequence of the linguistic theory. Now, consider the problem of maintaining (5). What happens when the theory is updated? Because it relies on many assumptions about other principles, how can we be sure (5) still holds? Neither Correa's parser nor the system described in this thesis has the capability to automatically derive this rule. Hence, we would be forced into the unsatisfactory position of having to to re-derive the theorem by hand.

2. *Problems with representability and explanation.*

   Another problem with adopting structural determination is that the system described here has the goal of being able to, or at least have the option of, building structures that violate principles of Case and Binding theory. (Linguistics texts including Lasnik & Uriagereka often make use of such examples). It would be very hard to implement such an option if the consequences of a given principle were embedded (along with those of other several principles) in different operations. The problem of not being able to represent violations of particular principles also translates into a problem for explanation. If a structure is eliminated by structural determination; then how can we tell if it is a Case or Binding theory violation? There is no easy way to distinguish betweeen the two. This would be a serious problem for the system described in this thesis because one of the tests used to inspire confidence in the implementation is that it should report exactly the same violations as predicted by Lasnik & Uriagereka.

3. *The correctness of the reformulation.*

Actually, a more serious question is: How do we know that the reformulation is correct in the first place? Correa does not mention any incompatibilities nor does he provide a formal proof that structural determination produces the same assignments of [±A,±P] as functional determination taken together with whatever other principles needed to derive a rule like (5). However, we can show by means of a simple example that structural determination is inconsistent with, and therefore cannot replace, functional determination. Consider the following *strong crossover* example (example (3:1) in Lasnik & Uriagereka):

(6) *Who$_1$ does he$_1$ think [Mary likes e_1]

(The impossible intepretation is "for which person x, x thinks Mary likes x.") According to Lasnik & Uriagereka, functional determination will assign [+A,+P] to e_1. Being an anaphor, e_1[+A] must be bound in the embedded clause. But since the nearest binder he_1 occurs outside this clause, this is an example of a Binding condition A violation. However, structural determination makes a different prediction. Since e_1 is in an object position, it satisfies all the preconditions of rule (5).[9] Hence, it will be assigned [−A,−P]. Since, e_1[−A] is not an anaphor, condition A will be trivially satisfied, and the sentence will be incorrectly analysed.[10]

There is a similar problem with another derived rule used by Correa. This is the *Chain Formation* algorithm which recovers the trace-antecedent relations produced by movement. We will not go into the details of this rule here except to mention that it subsumes Subjacency and integrates information from a variety of sources including parts of Case theory, Theta theory, and, surprisingly enough, the features assigned by structural determination.[11] However, Is chain formation correct? For example, does it recover the correct antecedent-trace relations as predicted by the principles that have been integrated? Correa mentions that, *inter alia*, it cannot handle parasitic gap sentences such as *which report did you file without reading*. However, the theory (in particular the functional determination analysis) as assumed by

[9] (1) Object positions are A-positions, i.e. positions to which θ-roles may be assigned (e.g. *John likes Mary*). (2) Objects are always governed (in this case, by the verb). (3) Case is assigned under government and *like* is a Case assigner in this context, so e_1 will get Case.

[10] For completeness: under a non-functional determination account, Condition C can be used instead to rule out the strong crossover example: "R-expressions (overt and non-overt [−A,−P] elements) must be A-free (in the domain of the head of its non-trivial chain if one exists)."

[11] The latter dependency is surprising because functional determination as formulated by Chomsky [10] is dependent on trace-antecedent relations being identified first. That is, the dependency in Correa's formulation is the opposite of what we might expect — see the dependency graph in figure 1.6.

> Correa does correctly permit such parasitic gap sentences. Actually, structural determination does make the right prediction in this case for the empty object of *reading* to be analysed as a variable. Hence, the fact that parasitic gap sentences are not handled, must be due to some peculiarity of the chain formation algorithm. We should add that this one case does not imply that structural determination correctly handles other parasitic-gap-type examples. For example, structural determination as well as chain formation must be fixed before the parser can handle an example like **The report was filed without reading.*[12]

To conclude, although it is often tempting to manually compile principles for efficiency, it should be avoided if possible because of the ease of introducing errors and the difficulty of ensuring correctness. Furthermore, we have argued that such reformulations also come with a loss of explanatory power. Perhaps it is not surprising that it is easy to get derived rules wrong; the free and complex interactions between principles means that it is easy to miss cases where incompatibilities can result. In the current system, principles are formulated independently of each other for these very reasons. However, it can be argued that principle interleaving causes a similar loss of explanatory power. Still, this is not fatal since the user always has the option of determining filter violations by re-configuring the parser to operate without interleaving. Note also that reformulation may lead to redundancies in the parser, which will not occur with interleaving. For example, the structural determination and chain formation algorithms must have much of the same force as the Case filter and θ-criterion, two of the principles that they integrate information from. That is, the latter two principles may not actually do much work because many such violations will have already been accounted for by the derived rules. However, it remains a mystery under what circumstances, if any, the original formulation of the two principles will remain independently useful. (Correa does not mention discarding these rules.) By comparison, when a principle is interleaved, it is wholly merged into the phrase structure recovery procedure. There is no possibility of introducing such unnecessary redundancy; that is, there is no need to keep the non-interleaved version around.

In the next section, we will turn our attention from the topic of principle representation to that of control organization.

2.5 Licensing Parsers

Abney [1] (also [2]) describes a parser that builds *licensing structures*. The basic idea here is that the parser builds structure only in accordance with certain licensing

[12] The object of *reading* will be (incorrectly) licensed as [−A,−P] by (5).

relations. Now, licensing relations form a distinguished subset of the set of general linguistic relations. For example, one way for a noun phrase to be properly licensed is to appear as an argument of some verb that will assign it a θ-role. The noun phrase is also free to participate in other (non-licensing) linguistic relations such as Case assignment or Binding; however, irrespective of what other relations are formed, only θ-role assignment will license the presence of the noun phrase in syntactic structure. The licensing of non-arguments by *predication* is another example of such a relation. For example, a relative clause such as *who arrested John* in the noun phrase *the man who arrested John* is usually considered to be an modifier rather than an argument of the noun *man*. Because modifiers are not arguments, they must be licensed in some way other than via θ-role assignment. In some theories (see van Riemsdijk & Williams [46] for example), relative clauses are properly licensed if they happen to be one-place predicates. For example, this can be used to explain the difference between the following two noun phrases:

(7) a. The man [who arrested John]
    b. *The man [(that) Bill arrested John]

The two noun phrases will have the following forms at LF:

(8) a. (the man) ($\lambda x, x$ arrested John)
    b. *(the man) (Bill arrested John)

That is, the relative clause in (7a) may be interpreted as a simple λ-expression, but the lack of an operator in (7b) prevents it from having the same analysis.[13]

Predication and θ-role-assignment are two of the four fundamental licensing relations that Abney describes.[14] (We will not describe the other two here.) There are two main reasons why we have not adopted a licensing approach:

1. *Representation of ill-formed sentences.*

   We are primarily interested in parsing both well-formed and ill-formed sentences such as those found in Lasnik & Uriagereka. The licensing approach does not allow structures that violate its restrictions to be represented. Examples like (7b) and (8b) do violate Abney's licensing restrictions. There is no reason why we should prevent these examples from being represented, especially since, unlike Abney, we make no claims with regard to psycholinguistic plausibility.

[13] Here, it is assumed that the operator *who* has moved to a clause-initial position to get scope, therefore leaving behind a trace in subject position to be interpreted as a bound variable.

[14] This view has some intuitive appeal in the sense that identification of the thematic and predication relations in a sentence, be it ill-formed or well-formed, seems to be a necessary pre-condition if semantic interpretation is to take place.

2. *Licensing is largely subsumed under the principle interleaving model.*

   In any case, irrespective of whether we need to represent licensing violations or not, the licensing model represents a case of unnecessary 'hardwiring' of control strategy. In the current system, the user can construct a 'licensing' parser with predication and θ-role assignment as privileged operations simply by specifying that the two operations, *FI: License Syntactic Adjuncts* (which requires that all adjuncts must satisfy predication) and *Assign Theta-Roles*, should be interleaved. Recall that if an operation is interleaved, it is applied (where relevant) to individual phrases as they are recovered. In particular, *FI: License Syntactic Adjuncts* will be applied as a precondition on any phrase that is about to be attached as a modifier. Thus, [*CP (that) Bill saw John*] can never be attached as a relative clause to a noun phrase such as [*NP the man*].[15]

2.6 Designing for Efficient Parsing

In this section, we will address the issue of selecting methods for efficient parsing. We will survey the standard methods that can be applied to parsing and point out those that have been utilized in the current implementation. Additionally, we will comment on any specific problems that the methods might pose for implementation.

Let us begin by outlining the preconditions for adopting any particular method:

- *Must be completely automatic.*

  For example, it is unreasonable to expect the grammar writer or user to have to guide or otherwise intervene to make the selected method terminate. Furthermore, the method must not expect the grammar writer to have to annotate, say, the description of the principles in any way.

- *Must preserve linguistic judgments.*

  That is, the correct parses or explanations of ill-formed sentences must be maintained.

To summarize, any method chosen must be completely transparent to the grammar writer. However, as in the case of the principle ordering and interleaving models, there may exist simple parameters of application that will be controllable by the user.

[15] Note that Abney also discusses the application of non-licensing principles such as chain formation and Subjacency after recovering a licensing structure. The same strategy can be used in this system by simply not interleaving these operations. However, there are many aspects of Abney's system pertaining to performance such as ambiguity resolution that we cannot mimic so easily.

In the following discussion, we will distinguish between *off-line* and *on-line* methods. For our purposes, off-line methods, i.e. methods that can be applied to improve the efficiency of the grammar independently of the input, will be preferable to those that are on-line, i.e. to be applied during parsing. The obvious reason is that off-line methods extract no penalty during parsing. On the other hand, on-line methods need to be carefully evaluated since any overhead caused by bookkeeping or extra evaluation can, especially unless tightly coded, swamp any benefits derived from the application of the method itself. Having said that, one possible disadvantage of an expensive off-line method is that re-parameterizing the grammar for different languages may take a considerable amount of time. This could also prove to be inefficient if the parser were to be embedded within a system that required rapid re-parameterization, such as in the case of a system that attempts to learn parameter values through iteration.

1. *Use existing grammar compilation techniques.* [Off-line]

   Although one might be tempted to abandon traditional parsing techniques at the same time as abandoning traditional theories of syntax, it would be unconscionable to ignore the large body of well-understood and efficient parsing algorithms for theories such as context-free grammars (CFGs). General CFGs can be parsed using a push down automata (PDA), or any number of tabular methods including the Cocke-Young-Kasami (CKY) and Earley's algorithms. Also, from the domain of programming languages where linear time parsing is obviously vital, tabular methods with different tradeoffs between generative capacity and program size have been developed for the SLR, LL, LALR, and LR subsets of context-free languages. (See Aho & Ullman [7].)

   An obvious candidate for such techniques is the core set of grammar rules based on $\overline{\mathrm{X}}$-theory that was described in section 1.6. Of the basic methods outlined above, the extended LR-based parsing methods, on the basis of the on-line/off-line distinction as observed by Tomita [56], seem to be preferable to other general methods such as Earley's algorithm, as used by Dorr [20]. In the current system, such an extended-LR(1) scheme is used for the S-structure grammar.

2. *Dynamic programming.* [On-line]

   This is a general technique that is useful for problems that can be recursively subdivided into smaller problems of the same type. If this results in many instances of the same sub-problems having to be solved, then it is usually more efficient to solve the sub-problem just once and 'save' the answers for subsequent occasions. This is also sometimes known as (simple) lemma generation. The essential tradeoff is one of the cost of computing each solution and the cost of saving and subsequent matching and retrieval of the solutions.

Tomita's extended-LR(1) *graph-structured stack* and *shared packed-forest* representations are an example of where this could be exploited to good effect in LR-style parsers. However, experiments with straightforward implementations of optimizations of this sort so far have not been promising. We will not go into the details here, save to mention that the cost of maintaining large phrase structure representations seemed to drastically outweigh any benefits.

3. *Partial Evaluation.* [Off-line]

   Partial evaluation, or the off-line partial execution of programs, has been widely discussed in the logic programming literature. There have been some notable successes such as the more-or-less automatic derivation of RETE-like optimizations for production systems (described in Takeuchi & Fujita [54]), and the derivation of a version of the efficient KMP-type (Knuth-Morris-Pratt) string-matching algorithm from a 'naïve' string-matching algorithm (Fujita [26]). (The latter case is especially interesting since it also incorporates off-line constraint propagation techniques.) However, there remain a number of outstanding problems (discussed in Fujita [26]) that need to be addressed before such methods will be generally applicable:

   (a) Simple partial evaluation, or just simply 'unravelling' goals and using substitution propagation, will not generally lead to a more than linear improvement in run-time performance.

   (b) Shortcutting (or 'folding') can produce large, e.g order-of-magnitude, improvements, but seem to rely on fortuitous circumstances to work. For example, the KMP-type shortcuts only seem to hold for certain matching patterns. Moreover, the string matching process has the advantage of a pre-determined partial input (an option not available in parsing), which provides additional information for program specialization.

   (c) Detailed control of the partial evaluation process seems necessary to achieve good results. In particular, annotations are generally used to indicate to the evaluator when it can safely completely execute code, or avoid non-termination. This violates our working assumption that adopted methods be completely automatic and transparent.

   In the area of principle-based parsers, Johnson [31] has shown that partial evaluation (including shortcutting) can produce parsers with different control strategies. However, our own experiments with Johnson's parsers suggests that the observations listed above seem to hold. For example, a wide range of performance including both significant slow-downs and speed-ups over an unevaluated parser are possible. However, it seems difficult to predict exactly which combinations will produce significant speed-ups, or whether the same results will hold for more substantial parsers, as Johnson's system only implements a small fraction of a principle-based theory.

At the moment, as will be described in chapter 4, the current implementation employs only simple partial evaluation, chiefly applied to the LR(1) parser-generation process.

4. *Constraint satisfaction methods.* [On-line]

   Because of the relatively free interaction between principles and the common use of finite domains, e.g. [$\pm$A,$\pm$P] features, indices (bounded with respect to a structure), constraint satisfaction methods should find wide application in principle-based systems. Standard techniques of propagating constraints offer the possibility of pruning the search space ahead of time to reduce the number of possibilities that have to be examined by standard backtracking schemes such as that employed in our system. The tradeoff with respect to efficiency is the added time that has to be spent periodically checking constraints. However, there are several other drawbacks. Perhaps the most important is that this usually changes the 'natural', or simplest formulation of programs (as observed by van Hentenryck [28]). It is unreasonable to expect a grammar writer to explicitly program in, say, forward checking, into principle definitions. Assuming that the system could be configured to recognize and transform such problems into one that uses constraints (a capability that the present system does not have), the lack of general (low-level) programming language support for even simple forms of constraints such as simple inequality in standard PROLOG makes explicit checking expensive. For instance, a built-in facility for inequality constraints would be useful for implementing the *i-within-i* condition used in the definition of chain formation (described in section 3.4.3). Currently, a non-transparently programmed constraint checker is used.

5. *Theorem proving.* [Off-line]

   We have already seen examples of hand-derived theorems, e.g. Correa's Structural Determination (and Chain Formation) definitions which integrate knowledge from various principles to reduce non-determinism. Another example where theorem proving can be useful employed is the theorem that *PRO* must be ungoverned. In Lasnik & Uriagereka, this is shown to be a consequence of the fact that *PRO* has Binding features [+A,+P] (and therefore must satisfy both Binding condition A and B). The present system does not make direct use of the *PRO* theorem, but effectively re-derives instances of the theorem each time it applies the two Binding conditions to *PRO*. Since the current system is oriented towards parsing rather than doing general theorem proving, it does not have the capability to generate such theorems.[16]

2.7 Other Issues: Directness and Faithfulness

The definitions of commonsense notions such as directness and faithfulness are somewhat unclear in the context of parser implementation. We will briefly comment on what they might mean in the framework of the implementation described in this thesis.

As discussed earlier, we have adopted a practical definition of correctness. That is, a parser must return the same parses for well-formed sentences and produce the same explanations in terms of the underlying principles as the reference, in our case Lasnik & Uriagereka. This is the minimum requirement that we can expect any implementation to satisfy. Let us now consider the other issues:

- *Directness.*

  As we have seen in the previous sections, the level of representation 'actually used to parse' is largely an illusionary one due to the possibility of having many layers of abstraction with automatic and transparent procedures to compile or interpret each succeeding level. As long as correctness with respect to the original specification, i.e. the high-level definition of grammar, is preserved at each stage, the grammar writer need only be concerned with the topmost level. That is, it only makes sense to 'ground' terms such as directness or faithfulness with respect to the level of representation seen by the grammar writer. Thus, perhaps one reasonable measure of the directness of an implementation is the amount of translation required to encode a given principle. For example, a 1st order logic definition would be more 'direct' than an set of attribute grammar rules (as used by Correa), or LISP or PASCAL code. Ideally, the framework we have adopted, i.e. a strict separation of principles and a representation that minimizes the amount of translation effort required of the grammar writer by using similar combinative and primitive elements, should be considered to be maximally direct. However, the current implementation does fall short of this goal in several respects. As we have mentioned earlier, in some cases, e.g. free indexation, linguistic definitions are not directly constructive. Also, the grammar writer has to 'program around' the deficiencies of standard PROLOG with respect to completeness and soundness.

- *Faithfulness.*

[16] Note that, as mentioned earlier, Stabler does have a 1st logic theorem prover that, presumably, has the capability to prove the *PRO* theorem; although, designing a system that could 'discover' such useful facts is probably another matter entirely.

The relevant question here is: Given the many layers of abstraction possible, what stronger practical requirements on parsing can be made other than simple correctness? Let us consider one such requirement. In the generate-and-test framework of the current system, we can distinguish between *linguistically-motivated choice points* and all others. The idea being that the parsing problem can be viewed abstractly as a general search problem. In general, at each point in the search space, there many be many choices of particular actions to take. Certainly, if we wish to take ambiguity into account by generating all possible parses, then parsing must be a non-deterministic process. Other theory-internal examples will be discussed in later chapters; let us briefly consider one of these. For example, in Lasnik & Uriagereka Case assignment is assumed to be an optional process in order to be able to account for examples such as:

(9) a. I want John to win
    b. I want *PRO* to win

(The subject *John* must get Case in (9a), but *PRO* must not get Case in (9b).)

The optionality of Case assignment is represented using two PROLOG clauses, one for each situation, (described in section 3.3.2). Thus during parsing, a (linguistically-motivated) choice point will be created for Case assignment. In general, implementations of a theory must implement all linguistic choice points or else risk getting the wrong analysis for some omitted case. However, all other choice points, call them *computational choice points*, may be eliminated without affecting the linguistic judgements rendered by the parser. Generally, computational choice points may be generated from 'supporting' definitions such as bookkeeping operations and maintenance of internal data structures. To summarize, we might then say that an implementation with only linguistic choice points is more faithful to the theory than one that has additional computational choice points. More specifically, the relative performance, e.g. in terms of parsing time, of the former implementation with respect to different inputs will more faithfully reflect the non-determinism of the theory than the latter implementation.[17]

Note that in the case of a parser that includes principles of performance — provided we regard performance constraints as being part of the original specification of grammar and correctness is maintained — then the same basic observations should hold.[18]

[17] We assume here that the branching factor in a given search space is the dominant factor that affects the time taken to explore that search space.

[18] One interesting question does arise. For example, if the specification requires the use of a

stack representation, then does this mean that a stack must be used at every level of abstraction? Or would it suffice to use a stack, say, only at the lowest level?

CHAPTER 3

The Representation of Linguistic Principles

One of the most important tasks in implementing a linguistic theory is how to turn the natural language, or semi-formal, definitions that are commonly found in the literature into the precise, formal definitions required for computation. The problem is to choose an appropriate level of representation between the abstract definitions used in the linguistics literature and the low-level, but efficient machinery used in computation. We will present a high-level notation for encoding principles that is abstract from certain control strategy choices, such as whether to interleave principle application with phrase structure construction or not, yet may be efficiently targeted to a variety of different machine architectures. As preliminary groundwork, first, let us review the important issues for representation design and how they are addressed in this system.

3.1 Design Issues

In this section, we discuss the reasons for adopting the following three design goals for our system:

1. A perspicuous representation that is fairly 'close' to linguistic definitions.
2. Efficiently executable definitions.
3. A representation that is abstract from control strategy choices.

An important goal of representation design is to select a representation that minimizes the amount of translation effort required of the grammar writer. There are many reasons why a 'direct' encoding should be adopted. Perhaps the most important is the correctness problem: that is, the problem of ensuring that all encoded forms are faithful to the original definitions. This is made all the more difficult by the lack of a standardized formal notation in the framework. Minimizing the encoding effort is also important from the point of view of theory maintenance. The rapid evolution of linguistic theories implies that frequent revisions will be necessary if parsing systems are not to become obsolete. The large number of extant versions of 'standard' theory is yet another motivating factor. Ideally, it should be possible to re-target a system for comparison purposes with minimal effort. All these factors suggest that the level of representation should be pitched as 'close' to that of the original definitions as possible.

The most obvious route to achieving a direct encoding is to design the representation to have the same vocabulary, i.e. the same primitive and combinative elements, as that used by linguists. For example, informal definitions often make use of pseudo-logic notation. Consider the following θ-marking uniformity condition on D-structure from Chomsky [12]:

(1) "If a position X is T-governed by α, then X is occupied by an argument if and only if X is θ-marked by α." (from (3:86))

For the purpose of this discussion, we will ignore the precise definition of linguistic terms such as T-government, θ-marking and argument. Instead, let us focus on the natural language connectives in the condition. They resemble their logical counterparts. For example, *"if...then..."* and *"if and only if"* are basically the logical implication and equivalence connectives, respectively. Hence, a natural step towards formalizing such a definition would be to substitute the formal logic connectives in place of the natural languages ones. To complete the formalization, we will also have to make explicit the quantifiers that range over the variables X and α, determine their relative scopes, and introduce a logical variable variable (plus an accompanying quantifier) to represent the element that occupies X. Linguistic terms like *"argument"* and *"D-structure position"* may be represented as one-place predicates. Similarly, binary linguistic relations such as *"T-governed by"*, *"occupied by"* and *"θ-marked by"* may be represented as two-place predicates. Hence, a formalized version of (1) might look something like:

(2) $$\begin{aligned}&\forall X\ \exists\alpha[(\ \text{D-structure-position}(X) \wedge \text{T-governed-by}(X,\alpha)\ ) \rightarrow \\ &\qquad (\exists Z[\ \text{occupied-by}(X,Z) \wedge \text{argument}(Z)\ ] \leftrightarrow \theta\text{-marked-by}(X,\alpha)\ )]\end{aligned}$$

The full 1st-order logic (FOL) notation employed by Stabler [52] is an example of this approach. However, there are well-known tractability problems for computing directly with FOL. Also, it is by no means clear that such a powerful notation is required to encode principle-based theories. (See the discussion of PROLOG in chapter 2 for details.)

At the other end of the spectrum, one could use a imperative programming language such as PASCAL (Wehrli [58]) or LISP (Kashket [32] and Dorr [20]) to provide a lower-level, but immediately operational specification. For example, a corresponding formulation in LISP might look something like:

(3)
```
(defun satisfies-condition (node)
   (if (T-governed-p node)
       (let ((alpha (T-governor-of node)))
          (if (argument-p (element-occupying node))
              (theta-marked-by-p node alpha)
              (not (theta-marked-by-p node alpha))))
       't))

(defun DS-filter (node)
   (if (satisfies-condition node)
       (if (has-subconstituents-p node)
           (every #'DS-filter (subconstituents node))
           't)))
```

Although the judgement is ultimately a subjective one, intuitively speaking, this definition is significantly less direct and perspicuous than the FOL formulation. In general, it will be harder to convince oneself both that the translation is correct and that changes will be propagated correctly should the original definition have to be updated. Also, unlike the FOL formulation, the LISP version (unnecessarily) commits the parser to a top-down traversal of phrase structure. This 'hardwiring' of control strategy is not flexible enough for our purposes: recall that one of the major motivating factors in building the system was to explore different control strategies. In the current implementation, principles may be interleaved with phrase structure construction, or applied only after skeletal S-structures are built, or some combination in between. As chapters 5 and 6 will describe, these two strategies operate using quite different methods. To insulate the grammar writer from such concerns, principles should be encoded 'declaratively' such that the semantics of the encoded forms can be understood independently without reference to the actual operational realization.

3.1.1 The Approach

In the remainder of the chapter, we will describe a representation that attempts to retain as much of the perspicuity of the FOL approach as possible without incurring the associated performance penalty. The gap between an abstract representation and the underlying operational specification will be bridged (albeit, somewhat

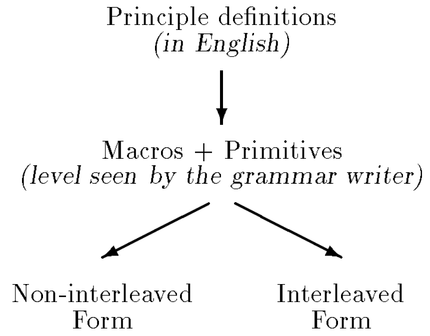


Figure 3.1 Multiple-targeting of linguistic principles.

incompletely) by an automatic translation procedure. This translation step, shown in figure 3.1, will allows us to 'multiply-target' different control strategy options in a manner that is transparent to the grammar writer. Basically, the representation will consist of two components:

1. *Two top-level principle-defining macros.*

   The simpler and more widely used macro is `in_all_configurations` which is used to encode universally quantified conditions over tree structures. (See figure 3.2 for a list of principles in the current system encoded using this macro.) The condition has the basic form $\forall X((X \in T \wedge p(X)) \rightarrow q(X))$, where X ranges over the constituents of a tree T, and p and q represent predicates that involve at least X. The second macro `compositional_cases_on` is used to encode compositionally defined operations on n-ary trees. For example, let Z be a constituent with two immediate sub-constituents X and Y. The macro may be used to define some function f over constituents such that $f(Z)$ is defined (recursively) in terms of the corresponding values for its children X and Y, i.e. $f(Z) = g(f(X), f(Y))$ for some function g.

2. *Linguistically-motivated primitives.*

   A set of linguistically-motivated primitives covering the access and manipulation of constituent features, category labels and sub-constituents.

Both components are written on 'top' of the underlying implementation language, PROLOG. Although the grammar writer has access to the full generality of the programming language, the multiple-targeting procedure is only defined for the two macro forms. This effectively constrains linguistic principles to be written using either one of these two forms. Note however that we deviate from this scheme for the principles of $\overline{\mathrm{X}}$-theory that define phrase structure. Such principles are treated

Operation	Encoded Using
θ-role assignment	`in_all_configurations`
θ-criterion	`in_all_configurations`
D-structure θ-condition	`in_all_configurations`
Subjacency	`compositional_cases_on`
Inherent Case assignment	`in_all_configurations`
Structural Case assignment	`in_all_configurations`
$\overline{\text{S}}$-deletion	`in_all_configurations`
Case filter	`in_all_configurations`
Case condition on traces	`in_all_configurations`
Trace theory (chain formation at S-structure)	`compositional_cases_on`
Wh-movement in syntax	`in_all_configurations`
Coindex Subject and INFL	`in_all_configurations`
Free Indexation	`compositional_cases_on`
Functional Determination	`compositional_cases_on`
Condition A	`in_all_configurations` (transformed)
Condition B	`in_all_configurations` (transformed)
Condition C	`in_all_configurations` (transformed)
Empty Category Principle (ECP)	`in_all_configurations` (transformed)
Control	`compositional_cases_on`
LF movement	Hand-coded
ECP at LF	`in_all_configurations` (transformed)
License operator-variables	`in_all_configurations`
License syntactic adjuncts	`in_all_configurations`
Wh-Comp requirement at LF	`in_all_configurations`

Figure 3.2 Parser Operations.

separately so that they may be compiled into an efficient machine for building phrase structure (to be described in chapter 4).

3.2 Overview of the Chapter

In the remaining sections of the chapter, we will illustrate the capabilities and limitations of the adopted representation by providing step-by-step examples of how principles in the current system are encoded. Let us briefly review the contents of the sections:

- *Using the* `in_all_configurations` *Form.*

  Section 3.3 contains a case study of how to directly encode English definitions of principles from Case theory using the universally quantified form `in_all_configurations`. We will address how *optionality* in Case assignment and access to constituents at D-structure (a level not directly computed by the phrase structure recovery procedure) are handled.

We will also take the opportunity to point out a (non-transparent) limitation of the underlying representation of phrase structures. In the current implementation, accessing constituents dominated by a larger constituent can be accomplished in a straightforward manner using various primitives supplied. However, due to the simplicity of the representation, there is no built-in method for accessing dominating constituents from a smaller constituent. The burden of ensuring that enough information gets passed to a parser operation for the operation to be carried out falls to the grammar writer. This is reflected in the definitions of many structural relations as an extra argument that holds 'context' information. For example, roughly speaking, government (see [Defn. 11] in appendix A) is normally considered to be a binary structural relation that obtains between two constituents that do not dominate one another. However, there is also the condition that there should be no intervening maximal projection, i.e. any phrasal node, between them. For example, the verb *sees* governs *Mary* but not *John* in [*John* [*VP* [*V sees*] *Mary*]]. To apply government, we must not only pass the two constituents but also enough surrounding context to determine if there is an intervening phrasal node. Hence, government is actually implemented as a ternary predicate `governs(X,Y,CF)` that holds if `X` governs `Y` *in* configuration `CF`.

- *Compositional Definitions. Part I*

  Certain parser operations (but not all as we shall see later) that require the establishment of *non-local*, i.e. potentially unbounded, structural relations pose difficult problems for two reasons:

  1. *Non-constructive definitions.*

     For example, the principle of free indexation in Lasnik & Uriagereka has the form:

     (4) "Freely assign indices" *(3:47b)*

     The relevant problem for computation is: How to turn an abstract definition of the form in (4) into a procedure that *sensibly* computes all possible indexings, i.e. all distinct assignments of indices, without emitting duplicate indexings? As it turns out, as we shall show later in the chapter, computing (4) basically amounts to enumerating the possible assignments of the *n-set partitioning problem*, a well-known problem in computational combinatorics. In the current system, an algorithm that enumerates the possible assignments is used.

     Note that this raises an important formalization issue. As will we see, the description of the algorithm cannot be claimed to be close to the notation used by linguists, i.e. (4). Also, it is not clear how one might attempt to raise the level of the representation to close the abstraction gap. For example, it is not clear how a compilation procedure could be constructed that automatically recognizes instances of well-known combinatorial problems from such abstract definitions.

2. *Interleaving.*

   There is also a conflict between the non-local nature of free indexation and the incremental processing employed by the interleaving model. Free indexation is obviously non-local because it allows any element to be coindexed with another element, or elements, anywhere in a sentence. Hence, if free indexation is to be interleaved with phrase structure construction, it must either: (1) simply wait until all elements to be indexed are available, i.e. have been recovered, before partitioning those elements into all possible combinations of distinct sets. Unfortunately, waiting to collect all possible elements would be tantamount to not interleaving at all. (2) An alternative is to compute the possible partitionings independently for each phrase as it is built. Then, as phrases are combined to become part of larger phrases, the partitionings of smaller phrases must be updated in accordance with the possible partitionings of the larger phrases. (Of course, care must be taken to ensure that all possible and no duplicate partitionings are produced.) This approach is the one adopted in the implemented system. General support for inductive definitions over tree structures — we will adopt the term *compositional definition* in this chapter — is provided by the `compositional_cases_on` form to be described in section 3.4.

Section 3.4 will illustrate the use of the `compositional_cases_on` form for two parser operations, free indexation and chain formation (recovery of the history of movement). Because neither operation is defined using compositional definitions in the linguistics literature, ensuring that the implemented definitions correctly cover all and only the possible cases is an especially important concern. Hence, to partially rectify the situation, we will prove that the definitions of both operations are *complete* in the sense that they are guaranteed to produce all possible assignments (of indices and chains), and *non-redundant* in the sense that they cannot produce duplicate assignments. Furthermore, as a side effect of the proof, we will also obtain a combinatorial analysis of the general problem of free indexation.

However, this does not constitute a completely satisfactory answer to the problem of representing non-constructively defined principles (or at least, definitions that do not immediately lead to constructive forms). A more satisfactory approach should include some procedure that recognizes non-constructive statements that are instances of well-known combinatorial problems, and then produce efficiently executable definitions automatically. Unfortunately, the current system does not have this capability. However, in section 3.5 (to be reviewed immediately below), we will discuss other cases involving non-local relations that the system does automatically transform into compositional definitions.

- *Compositional Definitions. Part II*

  Finally, section 3.5 will investigate the representation of principles that operate on non-local structural relations, but involve *minimality* in the sense of defining some smallest *domain* where those relations may or may not hold. For example, pronoun Binding is a non-local relation as illustrated in (5). However, there exist locality restrictions on Binding, namely the Binding conditions as discussed in chapter 1.

  (5) [John$_i$ knew [that [Mary [told her [she [thought [she [saw him$_i$]]]]]]]

  We will illustrate how such principles can be succinctly expressed using the `in_all_configurations` form. The system will automatically transform such definitions into equivalent compositional definitions. (See figure 3.2 for the list of parser operations that can take advantage of this transformation.)

  Apart from allowing certain principles to be more naturally expressed using the universal quantification form, an important advantage of the transformation procedure is that the transformed definitions also avoid certain kinds of repeated computation. Because a compositional definition computes structural relations in an incremental fashion, we can avoid any unnecessary repetition involved in finding minimal but *non-local* structural relations. For example, each time a larger phrase is built, a routine that looks for a Binder should not have to repeat its search in the smaller phrases it has already examined. The transformation procedure automatically transforms non-local relations such as Binding into local versions that can be incrementally applied in an efficient manner.

3.3 Using the `in_all_configurations` Form

A Case Study: Case Theory

Consider the following two sentences:

(6) a. *it it is likely John to be here
    b. it is likely that John is here

```
:- caseFilter in_all_configurations CF where
       lexicalNP(CF) then assignedCase(CF).

lexicalNP(NP) :- cat(NP,np), \+ ec(NP).
assignedCase(X) :- X has_feature case(Case), assigned(Case).
```

Figure 3.3 The definition of the Case filter.[2]

The usual explanation for the contrast between these two similar-looking examples is that *John* receives abstract Case in the latter example, but not in the former.[1] The key difference being that the non-matrix (or embedded) clause in (6b) is tensed, whereas the corresponding clause in (6a) is not. To exploit this distinction, we can say that a tense element in (6b) (which is missing or inactive for untensed clauses) is available to assign Case to, or Case-mark, *John*. Now, the Case filter, a principle of Case theory, requires that all lexical NPs such as *John* must receive Case. Hence, (6a) will violate the Case filter.

3.3.1 Case Filter

How can we implement this filter? The definition of the filter given in Lasnik & Uriagereka has the following semi-formal form:

(7) At S-structure, every lexical NP needs Case. *(1:20)*

Here, the basic elements being named are constituents (NPs) and features (lexicality, i.e. the property of having a morphological realization, and Case slot features). The combinatory terms *needs* and *every* imply that the representation, as a minimum, must include the ability to state feature membership conditions over constituents, and to universally quantify over particular classes of S-structure constituents.

[1]The term 'abstract' here refers to the fact that Case may not always be morphologically realized. For example, in *John saw him/*he* and *he/*him saw John*, the form of the pronoun *he* changes depending on whether it is in a subject or an object position, whereas the form of *John* remains invariant.

[2]For convenience, we will repeat the description of PROLOG notation introduced in chapter 1:

1. Logical variables are distinguished from constants by the use of an uppercase letter as the initial character. The scope of a variable is a clause. Also, '_' is used to indicate an anonymous or 'don't care' variable. The scope of '_' is itself.
2. Elements enclosed in square brackets denote lists. In particular, `[]` denotes the empty list. `[Y]` denotes a list with one element `Y`, `[Y1,Y2]` a list of two elements, and so on. `[X|L]` denotes a list with first element `X` and with `L` representing the remainder of the list.
3. Outside of lists, (`,`) is used to denote conjunction. (`:-`) — to be read as 'if' — is used to denote implication and (`\+`) is used to denote negation as failure to prove.

Compare (7) with the formulation shown in figure 3.3. The first statement represents a universally quantified formula over constituent structures. `caseFilter` names the following condition: "in every configuration `CF` where `lexicalNP(CF)` holds, then `assignedCase(CF)` must also hold." The next two clauses define what we mean by the terms *lexical NP* and (to be) *assigned Case*:

1. The first clause states that: "`NP` is a lexical NP if: (1) it has the category label `np`, and (2) it is not an empty category (`ec/1`)." Both `cat/2` and `ec/1` are primitives in the system.[3]

2. Two other two primitives are used in the second second, namely `has_feature/2` (which tests whether a constituent has the syntactic feature in question) and `assigned/1` (which tests if some feature slot has been bound). Hence, `assignCase/1` is meant to be read as: "`X` is assigned Case if: (1) `X` has a Case feature with slot `Case` and (2) that slot has been filled."

There are two important observations that we should make at this point:

- PROLOG uses a simple depth first (left-to-right) search strategy. Also, goals are matched with each clause in the order that clauses are written. The order of execution is unimportant for `lexicalNP/1`. However, `assignedCase/1` relies on the Case slot being identified before it is tested.

- Note also that the universally quantified form may be understood in a control-independent manner. In particular, the grammar writer need not know what PROLOG form or forms it expands into. For example, in the case when the principle is not being interleaved, the macro expands into a 'tree-walker' that recursively descends a given phrase structure representation testing each subtree using the specified conditional form. However, no tree-walking is done in the case when the principle is interleaved. Since the macro may been understood at a level abstract from its realization, there is no need to commit to any particular search strategy, as in, say, a direct operational encoding in an imperative programming language.

3.3.2 Structural Case Assignment

In the current implementation, the Case filter, i.e. testing for the presence of Case, can only operate after Case has been assigned. Now, according to the grammatical theory, Case may only be assigned to NPs at S-structure by certain constituents, called Case assigners, or markers. For simplicity, we will restrict the discussion of Case assigners to the following classes:

---

[3] Note: `p/n` is standard notation for a predicate `p` of arity `n`.

1. *AGR (Agreement).*

   Consider example (6) again, repeated below as (8):

   (8) a. *it it is likely John to be here
   b. it is likely that John is here

   Here, the (non-vacuous) agreement element of inflection (Infl) will assign nominative Case to *John* in the latter sentence. However, only finite, but not infinitival clauses have a non-vacuous agreement element (AGR). This implies that *John* will not get Case in the former case. As it turns out, there are no other possible Case assigners in the sentence. Hence, *John* will violate the Case filter.

2. *Verbs.*

   (a) Verbs usually assign accusative Case to their complements (if they exist) as in *John hit him* (cf. **John hit he*).[4]

   (b) Certain verbs, known as Exceptional Case Markers (ECM), may also assign Case to subjects of clausal complements as in *i believe John to be here* (cf. the ungrammatical example (8a)). However, this does not hold when the verb is passivized: for example, as in **it was believed John to be here.*

   (c) However, there are also certain 'technically intransitive' verbs such as *seem* or *ask* that do not assign Case to their complements. (See section 3.3.3.2 in Chomsky [12] for further discussion.) In the current implementation, such verbs (as well as the passivized ones) are marked with the lexical feature `noCasemark`.[5]

3. *Complementizers.*

   A complementizer such as *for* in English may assign oblique Case to the subject of a clause (another case of Exceptional Case Marking) . For example, *John* is Case-marked by *for* in *i am eager for John to be here*, thereby avoiding a violation of the Case filter — cf. **i am eager John to be here.*

[4] The actual implementation is slightly more complicated. Verbs sometimes may also assign dative Case to direct objects — see the Japanese examples shown previously in figure 1.3.

[5] Passivized verbs in English are implemented as follows: the passive auxiliary *be* subcategorizes for a verb phrase to be headed by a verb with the condition that the verb must take the past participle form, as in $[_{VP}\ be\ [_{VP}\ [_{V}$`[+en,noCaseMark]` $arrest\ ] \ldots\ ]]$. It also marks the subcategorized verb with the feature `noCasemark`. For technically intransitive verbs, `noCasemark` will appear directly in their lexical entries.

[6] In the current implementation, the two components of Infl, i.e. agreement and tense, are not represented as separate constituents, but simply as features of Infl.

```
caseAssigner(Infl,nom) :-  % Class 1: AGR assigns nominative Case.
        cat(Infl,i),
        Infl has_feature agr(_).6
caseAssigner(Verb,acc) :-  % Class 2: V assigns accusative Case
        cat(Verb,v),       % unless marked otherwise.
        \+ Verb has_feature noCasemark.
caseAssigner(ECM,obq) :-   % Class 3: ecm complementizer ('for') assigns
        cat(ECM,c),        % oblique Case.
        ECM has_feature ecm.
```

Figure 3.4 The definition of a Case assigner.

These cases are summarized in the definition of the predicate `caseAssigner` shown in figure 3.4. Here, `caseAssigner(X,C)` holds only if constituent `X` is a Case assigner that assigns Case of type `C`. Each case that we have mentioned above is encoded separately by a different clause that may be regarded as a single disjunct of an aggregate formula for the predicate. That is, if any clause holds, then the predicate also holds. (As discussed in chapter 2, the disjunction represented by the separate clauses encodes an exclusive-or rather than an inclusive-or.)

Having identified the possible Case assigners, Lasnik & Uriagereka define the conditions under which an element β can receive Case from an assigner α as follows:

(9) α assigns Case to β if *(1:26)*
  a. *α is a Case assigner;*
  b. *α governs β;*
  c. *α is adjacent to β.*

α assigns Case to β only in configurations of *government* where there cannot be "too much intervening" structure. Configurations that contain no intervening maximal projection, i.e. a phrasal boundary, between the assigner and assignee, as in $[_{\bar{V}}$ *V NP* $]$, i.e. a head-complement configuration, or $[_{IP}$ *NP* $[_{\bar{I}}$ $[$*I(AGR)*$]\ldots]]$, i.e. a specifier-head configuration, will satisfy government. Obviously, there must be certain exceptions to this rule such as the case where the subject of a clause may be Case-marked by an element outside the first maximal projection, as in *i believe* [*John to be here*]; the bracketing here is used to indicate the relevant phrasal boundary. (For brevity, we omit the formal definition of government. See [Defn. 11] in appendix A.) Finally, in English there is the additional constraint that the elements involved must be adjacent. For example, Lasnik & Uriagereka cites **i believe sincerely John to be here* as evidence where intervening material, in this case the adverb *sincerely*, may block Case assignment.

```
sCaseConfig(CF,Assigner,Case,NP) :-
    governs(Assigner,NP,CF),         % Assigner governs NP in CF, and
    cat(NP,np),                      % NP is a noun phrase, and
    caseAssigner(Assigner,Case),     % Assigner is a Case-marker assigning
                                     % Case of type Case, and
    adjacent(Assigner,NP,CF)         % Assigner and NP must be adjacent
            if caseAdjacency.        % sub-constituents in CF if parameter
                                     % caseAdjacency holds.
```

Figure 3.5 Configurations of structural Case assignment.

```
:- sCaseAssign in_all_configurations CF
       where sCaseConfig(CF,Assigner,Case,NP)
         then assignCase(Assigner,Case,NP).
```

Figure 3.6 The top-level definition of structural Case assignment.

These conditions may be encoded in a parallel fashion to (9), as shown in figure 3.5. `sCaseConfig/4` identifies a a structural Case assignment configuration `CF` as one that contains the following three components:

1. `Assigner`: the Case-marking element.
2. `Case`: the type of Case, e.g. nominative, accusative, etc., to be assigned.
3. `NP`: the noun phrase to receive Case.

We have assumed the existence of a three-place predicate `governs` such that a goal `governs(X,Y,Z)` holds only if `X` and `Y` are sub-constituents (note: not necessarily immediate sub-constituents) of `Z`, such that `X` governs `Y`. Similarly, `adjacent` is a three-place predicate such that `adjacent(X,Y,Z)` holds only if `X` and `Y` are adjacent sub-constituents of `Z`. Finally, `caseAdjacency` is a language parameter with the following interpretation: `caseAdjacency` holds if and only if the language requires Case adjacency.

We can now use `sCaseConfig/4` in the definition of structural Case assignment as shown in figure 3.6. As with the Case filter, the top-level definition makes use of the universal quantification macro. This form states that in all configurations satisfying the pre-conditions of Case marking, the operation of assigning Case must be applied. The act of assigning Case, encoded using the three-place predicate `assignCase`, is implemented by unifying the Case slot of the NP with `Case`.

Optionality in Case Assignment

One complication for the definition of `assignCase/3` is that Case marking may sometimes be optional. For example, consider the difference between *believe* and *want* when the non-matrix subject is non-overt (*PRO* being an empty pronoun):

(10) a. * I believe *PRO* to be here
b. I want *PRO* to be here
c. I believe John to be here
d. I want John to be here

One explanation for this contrast, discussed in Lasnik & Uriagereka, is that *want* optionally assigns Case whereas *believe* is an obligatory Case marker. In (10d), *want* must Case-mark *John*; otherwise the Case filter will be violated. Now, one property of *PRO* is that it is a special element that must not be Case-marked.[7] Since *want* is an optional Case marker, we can avoid violating this condition by simply not assigning Case in (10b). On the other hand, *believe*, being an obligatory Case marker, must assign Case to the forbidden element; hence the ungrammaticality of (10a). This distinction is encoded in the lexical entries of *want* and *believe*. Although both verbs are classified as ECM verbs, i.e. they both have the feature `ecm(Type)`, `Type` will be instantiated to `opt` (for optional) for *want*. Now, consider the two clauses that define `assignCase/3` as shown in figure 3.7. The first clause handles the situation when no Case is actually assigned, and the other clause handles the situation when Case is assigned (by unifying the Case to be assigned with the Case slot feature of the assignee). Note however, that these two clauses have 'overlapping scope'; that is, a Case assigner with the feature `ecm(opt)` can satisfy *either* clause, whereas an element without this feature, e.g. `ecm(oblig)` (for obligatory) for *believe*, can only satisfy the second clause. The parser accommodates both possibilities by trying each clause in the order in which they appear.

Configurations and Phrase Structure Representation

Consider again the configurations of structural Case assignment shown in figure 3.5. Note that *binary* structural relations such as 'governs' and 'adjacent' have been defined using three-place predicates. This situation arises because the phrase structure representation adopted effectively allows only 'one-way' access between constituents. More precisely, the representation of a phrase will contain all of its sub-constituents,

[7] In the account adopted in this thesis, this condition is not explicitly stated anywhere in the set of principles used by the parser. Instead, it falls out from the condition that *PRO* cannot be governed, since Case-marking presupposes government. This latter condition, which is known as the *PRO* theorem, is not explicitly stated anywhere either. In fact, it is a consequence of the need for *PRO* to satisfy two fundamental principles that are explicitly stated, namely, conditions A and B of the Binding theory.

```
% assignCase: Assigner × Case × Assignee
assignCase(ECM,_,_) :-                  % Case 1: Optional ECM
        ECM has_feature ecm(opt).       % do not assign Case.
assignCase(_,Case,NP) :-                % Case 2: assign Case.
        NP has_feature case(Case),
        morphCaseAgreement(NP,Case).    % 'he/him', 'they/their'
```

Figure 3.7 The optionality of Case assignment.

but there will be no (direct) way to access any other part of a structure that does not form some sub-constituent of the phrase. For example, given a structure [*IP* [*NP I*] [*VP pointed out* [*NP the man who I saw*]]], the object of the matrix clause is represented not as a single 'node' NP, but as the entire 'sub-tree' rooted at the object position, i.e. [*NP the man who I saw*]. From this sub-tree, it is a simple and direct procedure to (recursively) access any node that it dominates (i.e. any descendant). However, there is insufficient information to access any dominating node, e.g. the matrix VP node; or indeed, any node that does not dominate it nor is dominated by it, e.g. the subject *I*. Hence, it is important that predicates that encode structural relations be provided not only with the elements on which some relation is supposed to hold, but also with enough of the 'surrounding' phrase structure, or context, so that the structural connection between the elements can be established. For example, *the man* is adjacent to *who* in [*NP* [*NP the man*] [*CP* [*NP who*] [$\bar{C}$ *I saw*]]]. But the predicate `adjacent/3` requires a third argument, i.e. the context in which two elements are to be adjacent. This argument must be filled by a node that dominates both arguments; in this case, (at least) the NP node that represents the entire relative clause. In general, it will be up to the grammar writer to supply a configuration that dominates all arguments of a structural relation. Returning to figure 3.5, we have that `CF`, a configuration of structural Case assignment with three components (namely, a Case assigner, the type of Case to be assigned, and a NP to receives Case), must be a node that includes both the assigner and the assignee. For example, [*CP for* [*IP John to be here*]] is a Case assignment configuration, but not [*IP John to be here*].

Conceptually speaking, it would be relatively straightforward to augment the current representation with the necessary 'back-pointers' to parent nodes, and thus avoid having to 'manually' pass the extra parameter.[8] However, to avoid the com-

[8] Mark Johnson *(p.c.)* has brought to the attention of the author an interesting phrase structure representation (in PROLOG) that does not require explicit pointers to accomplish the doubly-linked scheme. Basically, each node with category `C` is represented as a triple `n(Ith,C/L,P)` where `L`

plexity involved in devising an efficient scheme for mainpulating and updating back-pointers in PROLOG, the present system has remained with the simple and non-transparent representation.[9] Actually, this unfortunate situation is remedied to some extent because the system also records certain kinds of positional information in the feature sets of constituents that would otherwise require back-pointers to verify. For example, constituents that occupy A-positions, short for *argument positions*([Defn. 2]) and/or complement positions have the features `apos` and/or `compl` automatically recorded in their feature sets, as a side-effect of the phrase structure recovery routine. Hence, instead of having to be able to access dominating nodes to check whether a certain element X occupies an A-position, we can simply test whether X `has_feature apos` holds.

This completes our discussion of structural Case assignment. Up to now, the principles outlined operate on phrase structure at the level of S-structure only. As chapter 4 will describe, for efficiency reasons, complete phrase structure representations are only recovered for the levels of S-structure and LF. In particular, D-structure constituents are recovered from S-structure only on a demand basis (to be made clear below). To illustrate how principles that operate on D-structure constituents are encoded, we now turn to the formalization of another instance of Case marking, that of inherent Case assignment.

Accessing D-structure constituents

Consider the nominal form [*NP* [*NP their*] [$\bar{N}$ [*N destruction*] [*NP of the city*]]]. Following Chomksy [12], we assume that both *the city* and *they* (*their* being the corresponding genitive form) are overt NPs that must be assigned Case at D-structure in order not to violate the Case filter. As with the corresponding non-nominal form *they destroyed the city*, the arguments *the city* and *they* are assumed to play the part of the 'target of destruction' and 'agent of destruction', respectively. Hence, they will receive a 'patient' and an 'agent' θ-role (respectively) from the noun *destruction* that heads the construction. In general, we have:

(11) *"Inherent Case is assigned by α to NP if and only if α θ-marks NP."*
from (pg. 193, Chomsky [12])

---

is a list of the nodes that it immediately dominates, `P` being the node that immediately dominates the current node, and `Ith` being the (unique) index of the current node among its siblings. Then the parent relation and i-th child relations can be simply defined as `P` *parent_of* `n(_,_,P)` and `n(Ith,T,Node)` *is* `Ith` *child_of* `Node`, respectively, where `Node=n(_,_/L,_)` and `T` is the `Ith` member of `L`.

[9] To properly implement a transparent representation will also require 'delays' in the case of principle interleaving. For example, the parent of a node X may not be known at the same time the node X is first made available by the phrase structure recovery routine. Hence, a goal such as Y `immediately_dominates` X would have to be delayed until Y has been generated.

```
:- inCaseAssign in_all_configurations CF where
       inCaseConfig(CF,Case,Items) then assignInCase(Case,Items).

% CF is a configuration of inherent Case assignment with two components:
% (1) Case, and (2) a list of constituents Items to receive Case.
inCaseConfig(CF,Case,Items) :-
    thetaConfig(CF,_,Items),          % CF is a θ-config. with arguments
    Head head_of CF,                  % Items, and Head heads CF, and
    cat(Head,C),                      % Head is of category C, and
    inherentCaseAssigner(C,Case).     % C is an inherent Case marker that
                                      % assigns Case of type Case.

% Category × Case
inherentCaseAssigner(n,gen).
inherentCaseAssigner(a,gen).
inherentCaseAssigner(p,obq).
```

Figure 3.8 Inherent Case assignment.

This uniformity condition ensures that the two arguments will not violate the Case filter. The correspondence between the nominalized and non-nominalized forms extends to passive constructions. For example, the nominalized form of *the city was destroyed* [*NP-t*] will be [*NP* [*NP the city's*] [$\bar{N}$ [*N destruction*] [*NP-t*]]], in which Move-α has (optionally) caused the *the city* to move from the complement position at D-structure. Of course, the other form [*NP* [$\bar{N}$ [*N destruction*] [*NP of the city*]]] is also available in the situation where there is no movement.

In general, adjectives such as *proud*, as in [*AP* [*A proud*] [*NP of John*]], as well as nouns assign genitive Case. Genitive Case is morphologically realized by affixing a marker that depends on the position of the NP at S-structure. For instance, in English it is realized as the semantically empty marker *of* for NPs in complement position (*of*-insertion), as in *destruction of the city* and *proud of John* (cf. **proud John*). However, in subject position, genitive Case is realized as the possessive marker *'s*, as in the passive form *the city's destruction*. For *their destruction of the city*, we assume that *'s* will combine with the subject *they* to form *their*.[10] Finally, prepositions are also assumed to Case-mark the NPs that they assign a θ-role to, as in *the city was destroyed* [*PP* [*P by*] [*NP John*]]. However, in English, oblique Case (unlike genitive Case) is not morphologically realized.

[10]For simplicity, in the present system, no derivation *they* + *'s* $\rightarrow$ *their* is actually done. The genitive form *their* has its own lexical entry with a special feature indicating that it 'already has' genitive Case.

```
assignInCase(Case,SSItem) :-
        recoverDsConstituent(DSItem,SSItem),
        DSItem has_feature case(Case) if cat(DSItem,np).
```

Figure 3.9 Case assignment at D-structure.

To summarize, inherent Case assignment can be encoded in a parallel fashion to structural Case assignment, as shown in figure 3.8. Here, we assume a predicate `thetaConfig/3` that defines the configurations of θ-marking, `CF`, where `Items` holds a list of the elements in the configuration to be θ-marked. (We will not go into the details of `thetaConfig` here; the definition of θ-role assignment will be described in detail in chapter 6.) For example, `Items` would be a list containing the two elements *their* and *the city* for the θ-configuration *their destruction of the city*. Then, the configurations of inherent Case assignment are simply those θ-marking configurations that are headed by an inherent Case assigner. But because the phrase structure recovery routine produces structures at the level of S-structure, not D-structure, the θ-configuration items returned are the elements that occupy θ-positions at S-structure. For example, the complement position in the phrase [*NP the city's* [$\bar{N}$ *destruction* [*NP-t*]]] is occupied by the trace of *the city* at S-structure.[11] These S-structure constituents are passed to the predicate `assignInCase` that will unify `Case` with the Case slots of the relevant constituents at D-structure. To retrieve the corresponding constituent that occupies a given θ-position at D-structure, we make use of a primitive operation `recoverDsConstituent` (to be described in section 4.3.2), as shown in figure 3.9. (For simplicity, we have only shown the definition of `assignInCase` for the instance where only a single item is to be assigned Case.[12])

In the current implementation, it is left to the grammar writer to specify when to access D-structure constituents. An alternative (but equivalent procedure) is to first re-express a principle that operates on D-structure as a (derived) principle

[11] Chomsky [12] distinguishes between positions and the elements that occupy those positions. Strictly speaking, a θ-role is assigned to a position, whereas Case is assigned to the element that occupies a given position. Of course, the effects of such a distinction will only be apparent under movement. In the current representation of phrase structure, both kinds of syntactic features such as Case and θ-slots are held in the feature set associated with an element. This can probably be regarded as a design flaw in the current implementation. Hence, when elements are moved by Move-α, we should be careful to copy certain features to the new destination, e.g. Case, and leave others, e.g. θ-slots, behind in the feature set of the trace that occupies the original position.

[12] Note that we have retreated considerably from the strict uniformity condition (11). Not all configurations of θ-marking are configurations of inherent Case assignment, e.g. verbs θ-mark their complements, but verbs assign structural, not inherent, Case. Also, clausal arguments must be θ-marked, e.g. *the belief* [*that Mary left*], but it is not clear whether clauses should also be Case-marked (see note 9 of chapter 1 in Lasnik & Uriagereka).

that operates at S-structure. This is often done in the literature. For example, Chomsky [12] re-expresses the uniformity condition shown in (11) as:

(12) *"If α is an inherent Case-marker, then α Case-marks NP if and only if [it] θ-marks the chain headed by NP."* *(from ch.3, no.272)*

The reformulated definition which operates at S-structure makes explicit the use of the chains constructed through movement that the call to the D-structure constituent recovery primitive encoded.

This concludes our discussion on the use of the `in_all_configurations` form. This universal quantification macro is used to encode many of the principles in the current implementation. In fact, as figure 3.2 shows, a majority of the current parser operations are encoded directly in this form. In the next two sections, we will turn to illustrate the second principle-defining form `compositional_cases_on` which accommodates definitions that involve structural induction over tree structures.

3.4 Compositional Definitions Part I: Enumeration of Combinatorial Operations

In this section, we will describe how inherently combinatorial operations such as free indexation and chain formation are encoded in the current system. These operations share the general characteristic of having many logically possible assignments, of indices and chains, respectively, for any given phrase structure. In general, there are two major issues to be considered in enumerating such assignments:

- How can we guarantee that the implementation will recover each logically possible assignment? Unless this can be proved, there will be the risk of failing to produce an assignment that corresponds to a well-formed structure for some particular sentence, or an assignment that is ill-formed for a particular sentence will be ruled out for the wrong reason.

- How can we construct a procedure that works irrespective of whether the operation is to be interleaved or not? As will be explained below, the obvious procedures for enumerating such assignments do not accommodate the interleaved model. In fact, it will be necessary to define such procedures compositionally over tree structures in order for assignments to be produced incrementally as partial structures are recovered.

In this section, we will describe compositional definitions for free indexation and chain formation that provably enumerate all and only those assignments, i.e. without duplicates, that are logically possible.

Section Roadmap

In section 3.4.1, we will use linguistic examples to introduce the operations of free indexation and chain formation. We will also take the opportunity to expand on the two problems for enumeration that were briefly mentioned above. Next, sections 3.4.2 and 3.4.3 will each present a compositional definition together with a proof of correctness for free indexation and chain formation, respectively.

3.4.1 The Problems of Chain Formation and Free Indexation

Completing Phrase Structure Representations

As chapter 4 will describe, the phrase structure recovery procedure initially recovers representations at the level of S-structure for reasons of computational efficiency. These representations are incomplete in many respects. For example, the phrase structure recovery procedure does not recover the chains formed by syntactic movement between D- and S-structure. Also, indices that indicate coreference possibilities for noun phrase constituents such as pronouns and anaphors are missing. Empty noun phrases are particularly underspecified. Although they are inserted (where possible) to occupy positions normally occupied by overt noun phrases, e.g. subject and object positions, during the initial phrase structure recovery stage, they lack many syntactic features. For instance, their status as being empty elements generated through Move-α, or not, as may be the case, remains undetermined. For example, the phrase structure recovery procedure will assign essentially the same structure to (13a) and (13b):

(13) a. John seems [[*NP*-e] to be happy]
b. John wants [[*NP*-e] to be happy]

In particular, the empty element [*NP*-e] will occupy the subject position of the clausal complement in both cases. However, in order to satisfy θ-theory, [*NP*-e] must be a trace in the former case (in fact, it must be the trace of *John*, thus forming a chain $\langle$*John*, [*NP*-e]$\rangle$), but not in the latter case where it is usually assumed that *John* has not been subjected to Move-α. For example, compare (13) with (14):

(14) a. *John seems that he is happy
b. John wants Mary to be happy

To capture both possibilities shown in (14), the [*NP*-e] constituent must be allowed to *optionally* participate in movement. The task of recovering the possible movement chains falls under the province of the chain formation algorithm. Principles of grammar from other modules such as θ-theory will restrict the possibilities to those that are well-formed.

As mentioned above, noun phrases also lack the indices that are necessary to indicate various coreference possibilities. For example, the sentence *John believes Bill will identify him* is ambiguous. Here, the pronoun *him* can be interpreted as being coreferential with *John* or with some person not named in the sentence, but not with *Bill.* We can represent these various cases by assigning indices to all noun phrases in a sentence together with the interpretation that two noun phrases are coreferential if and only if they are coindexed, that is, if they have the same index. Hence the following indexings represent the three coreference options for pronominal *him*:

(15) a. John_1 believes Bill_2 will identify him_1
b. John_1 believes Bill_2 will identify him_3
c. *John_1 believes Bill_2 will identify him_2

Once indices have been assigned, Binding conditions that state constraints on the locality of reference of pronominals and R-expressions, i.e. *referential* elements such as *John* and *Bill*, will conspire to rule out the impossible interpretation (15c). At the same time, the constraints will allow the other two (valid) interpretations. *Free indexation* is the process of assigning indices to noun phrases. (16) contains an updated version of the definition shown previously in (4): form:

(16) Assign indices freely to all noun phrases.[13]

Hence, free indexation accounts for the fact that we have coreferential ambiguities in language. Other principles of grammar will interact so as to limit the number of indexings generated by free indexation to those that are semantically well-formed.

Finally, it is worth mentioning at this point that the phrase structure recovery procedure also leaves the Binding theory status of empty noun phrases unspecified. As explained in chapter 1, some empty noun phrases exhibit the same locality restrictions as overt anaphors or pronouns (or both or neither of them). As explained in chapter 2, a theory of *functional determination* can be used to determine the appropriate status. By contrast, the status of overt nouns as anaphors or pronominals are specified in the lexicon.

The Problems of Formalization and Interleaving

One common aspect of both free indexation and chain formation is that, in general, there are many possible assignments of indexings and chains to a given phrase structure at S-structure. In general, free indexation must be capable of

[13]The exact form of (16) varies according to different theories in the principles-and-parameters framework. For example, in Chomsky [10] (pg.59), free indexation is restricted to apply to A-positions at the level of S-structure, and to $\bar{\text{A}}$-positions at the level of logical form.

5 indexings for sentences with 3 noun phrases:

(111)	John_1 wanted PRO_1 to forgive himself_1
(112)	John_1 wanted PRO_1 to forgive him_2
(121)	John_1 wanted Mary_2 to forgive him_1
(122)	John_1 wanted Mary_2 to forgive herself_2
(123)	John_1 wanted Mary_2 to forgive him_3

15 indexings for sentences with 4 noun phrases:

(1111)	John_1 persuaded himself_1 that he_1 should give himself_1 up
(1222)	John_1 persuaded Mary_2 PRO_2 to forgive herself_2
(1112)	John_1 persuaded himself_1 PRO_1 to forgive her_2
(1221)	John_1 persuaded Mary_2 PRO_2 to forgive him_1
(1223)	John_1 persuaded Mary_2 PRO_2 to forgive him_3
(1233)	John_1 wanted Bill_2 to ask Mary_3 PRO_3 to leave
(1122)	John_1 wanted PRO_1 to tell Mary_2 about herself_2
(1211)	John_1 wanted Mary_2 to tell him_1 about himself_1
(1121)	John_1 wanted PRO_1 to tell Mary_2 about himself_1
(1232)	John_1 wanted Bill_2 to tell Mary_3 about himself_2
(1123)	John_1 wanted PRO_1 to tell Mary_2 about Tom_3
(1213)	John_1 wanted Mary_2 to tell him_1 about Tom_3
(1231)	John_1 wanted Mary_2 to tell Tom_3 about him_1
(1234)	John_1 wanted Mary_2 to tell Tom_3 about Bill_4

Figure 3.10 Example of various indexings

generating each possible indexing or else risk missing a valid interpretation for some sentence. For example, there are five and fifteen possible indexings for a sentence containing three and four noun phrases, respectively; all of which correspond to some valid interpretation as shown in figure 3.10. As discussed at the beginning of the chapter, the relevant problem for computation is how to turn an abstract definition of the form shown in (16) into a procedure that provably generates all possible indexings? Moreover, in the interest of computational efficiency, we would also like the procedure to (provably) generate *only* the possible indexings without redundancy. Finally, the system should derive such a procedure *automatically*, i.e. without the intervention of the grammar writer, for reasons of perspicuity and abstraction.

An additional problem is that the chain formation procedure must accommodate the possibility of interleaving chain formation. Let us briefly review the interleaving model. Basically, the idea here is to apply the well-formedness conditions that represent linguistic principles to partial structures 'as they are assembled' by the

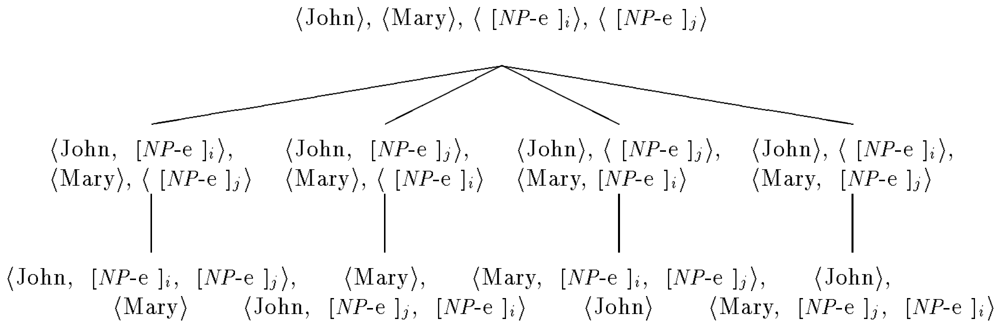


Figure 3.11 Computation tree for chain formation.

phrase structure recovery procedure. The motivating factor for this approach (as opposed to a scheme in which well-formedness principles are only applied after the structure for a complete sentence has been recovered), is that it will be possible to avoid building a complete, but ill-formed structure if some principle can be applied to rule out some portion of that structure. Let us now briefly consider the problems faced by chain formation in the light of this model.

Perhaps the simplest method for assigning chains to a structure is to first collect all the constituents that could possibly be members of a chain into a list. (We will consider only the movement of NPs in this discussion.) There is only one possible assignment in which chains may contain at most one element. That is, by having each element of the list head its own (trivial) chain. Then, the possible chain combinations with at most two elements can be found by optionally adding any [*NP*-e] element (only empty NPs can be traces) to any chain of length one. The possible combinations for chains with at most three elements can be incrementally derived from each combination of length up to two by optionally adding any [*NP*-e] not already part of a non-trivial chain, i.e. a chain with more than one member, to any chain with two members. In general, we only need to repeat this last step up a maximum of $n + 1$ times where n is the number of elements in the original list. Figure 3.11 illustrates this procedure for the list *John*, *Mary*, [*NP*-e]$_i$ and [*NP*-e]$_j$. Here, each of the nine nodes in the computation tree represents one of the possible chain assignments. This simple procedure produces each possible chain assignment without any duplicates and works well in the case where chain formation applies after structure building has been completed. Unfortunately, this procedure is not flexible enough to be used in the interleaved model because all noun phrases in a sentence must be identified before the computation tree can be constructed. But, to delay chain formation until all noun phrases in the sentence have been recovered by the phrase structure recovery procedure would negate much of the advantage of the interleaved model.

A similar procedure also works for free indexation. All and only the possible indexings can be easily found by, first, collecting all noun phrases into a list. Then, simply take each element in turn and optionally coindex it with each element following it in the list. However, this method also suffers from the same problem with delayed evaluation as the corresponding method for chain formation.

However, a procedure that is defined *compositionally* on phrase structures can be effectively interleaved. That is, chain formation, for example, should be defined so that the chains present in some phrase are some function of the chains present in that phrase's sub-constituents. Then, chains can be computed incrementally for all individual phrases as they are built. Such a method would have the advantage of being *unbiased* in the sense that it will be applicable irrespective of whether the operation is to be interleaved or not. In the remainder of this section we present compositional definitions for both chain formation and free indexation. However, despite the additional flexibility, these definitions also have the property of being very different in flavour from the 'original' specifications. As mentioned at the start of the chapter, this difference cannot be made transparent to the grammar writer because the current system does not have the capability to automatically synthesize compositional definitions from abstract specifications. This obviously compromises our goal of separating concerns about control options from principle definitions. Moreover, the correctness of representation becomes an important concern because of the extra 'derivation step' that has to be performed manually by the grammar writer.

In section 3.4.2, we will prove the 'correctness' of a compositional definition for free indexation, in the sense established at the beginning of the chapter. Furthermore, as a side-effect of the proof procedure, we will derive a tight bound for its complexity in terms of the number of referentially distinct index assignments as a function of the number of noun phrases in a sentence. Next, section 3.4.3 will repeat the process for chain formation.

3.4.2 Compositional Free Indexation

Before going on to discuss specific details of the free indexation operation, first let us review the model of indexation adopted by the present system (following (50) in Lasnik & Uriagereka(pg.82)):

(17) a. Coindex via movement.
b. Freely assign indices to NPs in A-positions only at S-structure.
c. Freely assign indices at LF.

We should also add the condition:

(17d) Indices can only be assigned through (17)a–c.[14]

Here, we have assumed that syntactic elements do not inherently possess indices. That is, the only way (apart from free indexation) that a constituent can acquire an index is via movement (condition (17a)). This condition ensures, *inter alia*, that resumptive pronoun constructions such as (19) cannot license parasitic gaps under the theory of functional determination adopted in the current system:[15]

(19) *The article which$_i$ I filed it$_i$ yesterday without reading e_i is over there
Lasnik & Uriagereka (3:46)

Here, the parasitic gap e_i will be (incorrectly) licensed by being functionally determined as a variable, since it is locally bound by the relative operator *which* in an $\overline{\text{A}}$-position. According to Lasnik & Uriagereka, condition (17b) prevents e_i from being licensed as a parasitic gap, since functional determination is assumed to apply at S-structure only. Finally condition (17c) permits elements that are base-generated in $\overline{\text{A}}$-positions to receive the indices that are necessary for interpretation in cases of resumptive pronoun constructions not involving parasitic gaps, such as *the man who I don't believe the claim that anyone saw him* (example [3.2.2:12(i)] in Chomsky [9]). For the remainder of this section, we will be concerned with the implementation of free indexation as defined in condition (17b). In particular, we will address the following three issues:

1. *The combinatorics of free indexation.*

   First, we will investigate the combinatorics of free indexation. By relating the problem to the well-known n-set partitioning problem, we will show that free indexation must produce an exponential number of referentially distinct phrase structures given a structure with n (independent) noun phrases.

[14] Of course, the subject of a clause and Infl will also need to be coindexed for agreement. Note that this constraint does not affect how indices are initially assigned because the subject position is an A-position and head movement will ensure that Infl will always have an index. The same reasoning will also apply to other processes such as the linking of non-arguments (not implemented in the current system), as in *there$_i$ is* [*a man*]$_i$ *in the room.*

[15] Condition (17b) also prevents overgeneration in cases such as (18):

(18) a. who left?
b. [$_{CP}$ who$_i$ [$_{IP}$ t_i left]]
c. [$_{CP}$ who$_i$ [$_{IP}$ e_i left]]

In the standard account for (18a), *who* is assumed to have been moved from the subject position to COMP, as shown in (18b). However, there is one other possibility that is consistent with a simple model of indexation, as shown in (18c). Here, *who* (base-generated in COMP) binds an empty NP in subject position. However, e_i is not a trace. Furthermore, at S-structure, e_i (as in the case of t_i) will be functionally determined as a variable. However, note that (18c) will not be generated under (17). (*who* does not receive an index and hence cannot bind anything. As a result, e_i will be functionally determined as *PRO*. Hence, 18c will be ruled by condition A of the Binding Theory.)

2. *The enumeration of indexings.*

   Secondly, we will describe a non-deterministic procedure for enumerating the logically possible indexings. This procedure will be compositionally defined over general tree structures. We will use the results of the first part to show that the procedure is correct, both in terms of completeness and non-redundancy, in the senses discussed earlier.

3. *The representation of the enumeration procedure.*

   Finally, we will introduce a general notation for encoding compositional definitions. We will illustrate its use by showing how compositional free indexation may be represented in a perspicuous fashion.

The Combinatorics of Free Indexation

Consider the general problem of partitioning a set of n elements into m *non-empty* (disjoint) subsets. For example, a set of four elements $\{w, x, y, z\}$ can be partitioned into two subsets in the following seven ways:

$$\begin{array}{ll} \{w,x,y\}\{z\} & \{w,x\}\{y,z\} \\ \{w,x,z\}\{y\} & \{w,y\}\{x,z\} \\ \{w,y,z\}\{x\} & \{w,z\}\{x,y\} \\ \{x,y,z\}\{w\} & \end{array}$$

The number of partitions obtained thus is usually represented using the notation $\left\{ {n \atop m} \right\}$ (Knuth [33]). In general, the number of ways of partitioning n elements into m sets is given by the following formula. (See Purdom & Brown [44] for the derivation of (20).)

(20)
$$\left\{ {n+1 \atop m+1} \right\} = \left\{ {n \atop m} \right\} + (m+1)\left\{ {n \atop m+1} \right\}$$

$$\text{for } n, m \geq 0$$

The number of ways of partitioning n elements into zero sets, $\left\{ {n \atop 0} \right\}$, is defined to be zero for $n > 0$ and one when $n = 0$. Similarly, $\left\{ {0 \atop m} \right\}$, the number of ways of partitioning zero elements into m sets is zero for $m > 0$ and one when $m = 0$.

We observe that the problem of free indexation may be expressed as the problem of assigning $1, 2, \ldots, n$ distinct indices to n noun phrases where n is the number of noun phrases in a sentence. Now, the general problem of assigning m distinct indices to n noun phrases is isomorphic to the problem of partitioning n elements into m non-empty disjoint subsets. The correspondence here is that each partitioned subset represents a set of noun phrases with the same index. Hence, the number of indexings for a sentence with n noun phrases is:

(21)
$$\sum_{m=1}^{n} \left\{ {n \atop m} \right\}$$

(The quantity in (21) is commonly known as Bell's Exponential Number B_n; see Berge [5].) The recurrence relation in (20) has the following solution (Abramowitz [3]):

(22)
$$\left\{ {n \atop m} \right\} = \frac{1}{m!} \sum_{k=0}^{m} (-1)^{m-k} \binom{m}{k} k^n$$

Using (22), we can obtain a finite summation form for the number of indexings:

(23)
$$B_n = \sum_{m=1}^{n} \sum_{k=0}^{m} \frac{(-1)^{m-k}}{(m-k)!\,k!} k^n$$

It can also be shown (Graham [27]) that B_n is asymptotically equal to (24):

(24)
$$\frac{{m_n}^n e^{m_n - n - \frac{1}{2}}}{\sqrt{\ln n}}$$

where the quantity m_n is given by:

(25)
$$m_n \ln m_n = n - \frac{1}{2}$$

That is, (24) is both an upper and lower bound on the number of indexings. To provide some idea of how fast the number of possible indexings increases with the number of noun phrases in a phrase structure, figure 3.12 exhibits the values of B_n for $n = 1, 2, \ldots, 10$.

A Procedure for Enumerating Indexings

For each node X in a tree structure, we will associate a set I_X of indexed noun phrases dominated by node X. Each index set I_X will represent the range of indices assigned within the phrase X. For example, figure 3.13 illustrates the set of indices associated with each constituent in *who*$_i$ *did John*$_j$ *see* t_i.[16]. Basically, the procedure

[16] Ignoring head movement. See section 4.4.1.

NPs	Indexings	NPs	Indexings
1	1	7	877
2	2	8	4140
3	5	9	21147
4	15	10	115975
5	52	11	678570
6	203	12	4123597

Figure 3.12 Table of the first dozen values of B_n.

$$\begin{array}{llllllll} [_{CP} & [_{NP}\ \text{who}_i] & [_{\overline{C}} & [_{C}\ \text{did}] & [_{IP} & [_{NP}\ \text{John}_j] & [_{VP}\ \text{see} & [_{NP}\ t_i]]]]] \\ \{i,j\} & \{i\} & \{i,j\} & \{\} & \{i,j\} & \{j\} & \{i\} & \{i\} \end{array}$$

Figure 3.13 Index sets for *who did John see?*

operates recursively by calculating the possible index assignments, or *indexings*, for a phrase in terms of the possible indexings for its immediate sub-constituents. There are two cases to consider:

1. Base Case: X is a leaf node.

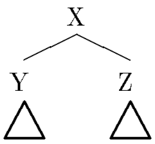
   (a) X is a noun phrase in an A-position. By condition (17b), X should be assigned a (new) index *unless* it already has one (through movement) by condition (17a). In either case, let i be the index. Then, $I_X = \{X_i\}$. For the remainder of the section, it will be convenient to represent a set of indexed NPs using just the indices present. Hence, $I_X = \{i\}$.

   (b) X is not a noun phrase in an A-position. In this case, X should not receive an index, i.e. $I_X = \emptyset$.

2. Compositional Case: X is an interior node.

   (a) Unary branching: Suppose X has a single immediate constituent Y. Since no additional elements are introduced by X, the range of indices in X will be the same as that for Y. In other words, X will inherit the index set of Y: that is, $I_X = I_Y$.

   (b) Binary branching: Suppose X has two immediate constituents Y and Z:



$I_Y = \{i_1, \ldots, i_m\}, m \geq 0$ $\qquad$ $I_Z = \{i_{m+1}, \ldots, i_n\}, n \geq m$

In order to compute the possible index sets of X, given particular index sets for Y and Z, we must allow for the possibility that elements in one phrase, say Y, could be coindexed with elements from the other phrase, Z. This will be accomplished by allowing indices from one set to be (optionally) merged with distinct indices from the other set. Let us call this process *cross-indexing*. For example, the phrases [*NP John*]$_i$ and [*VP likes* [*NP him*]$_j$] have index sets $\{i\}$ and $\{j\}$, respectively. Free indexation must allow for the possibilities that *John* and *him* could be coindexed or maintain distinct indices. Cross-indexing accounts for this by optionally merging indices i and j. Hence, we obtain:

(26) a. John$_i$ likes him$_i$, *i merged with j*
b. John$_i$ likes him$_j$, *i not merged with j*

More formally, let us assume for the moment that phrase structures are binary branching. Then:

(27) Operation: `crossIndex`: $I_Y \times I_Z$
Let $\bar{I}_Y$ represent those elements of I_Y which are not also members of I_Z, that is, $(I_Y - I_Z)$. Similarly, let $\bar{I}_Z$ be $(I_Z - I_Z)$.
i. If either $\bar{I}_Y$ or $\bar{I}_Z$ are empty sets, then stop.
ii. Let y and z be members of $\bar{I}_Y$ and $\bar{I}_Z$, respectively.
iii. *Either* merge indices y and z *or* do nothing.
iv. Repeat from step (i) with $\bar{I}_Y - \{y\}$ in place of $\bar{I}_Y$.
Replace $\bar{I}_Z$ with $\bar{I}_Z - \{z\}$ if y and z have been merged.

Next, we must find the index set of the larger phrase X. Hence:

(28) Index Set Propagation: $I_X = I_Y \cup I_Z$.

That is, the index set of an aggregate phrase is just the set union of the index sets of its sub-phrases *after* cross-indexation. Returning to our earlier example, *John likes him*, here, (26a) and (26b) will have index sets $\{i\}$ and $\{i, j\}$, respectively.

Note that the above procedure must have an additional step in cases where the larger phrase itself may be indexed; for instance, as in [*NP*$_i$

[NP_j *John's*] *mother*]. In such cases, the extra step will be simply to merge the singleton set consisting of the index of the larger phrase with the result of cross-indexing in the first step. In other words, a structure of the form [NP_i [$NP_j \ldots \alpha \ldots$] $\ldots \beta \ldots$] can always be handled as [P_1 [NP_i] [P_2 [$NP_j \ldots \alpha \ldots$] $\ldots \beta \ldots$]].

(c) n-ary branching: The algorithm generalizes to n-ary branching, for $n \geq 3$, using iteration. For example, a ternary branching structure such as [$_P$ X Y Z] can be handled in the same way as [P X [P' Y Z]].

The nondeterminism in step (iii) of cross-indexing will generate all and only all the possible indexings. We will show this in two parts. First, we will argue that the above procedure cannot generate duplicate indexings. As shown in the previous section, the combinatorics of free-indexing indicates that there are only B_n possible indexings. Secondly, we will demonstrate that the algorithm generates exactly that number of indexings. If the algorithm satisfies both of these conditions, then we have shown that it generates all the possible indexings exactly once:

1. Consider the definition of cross-indexing. Here, $\bar{I}_X$ represents those indices in X that do not appear in Y. (Similarly for $\bar{I}_Y$.) Note also that whenever two indices are merged in step (ii), they are 'removed' from $\bar{I}_X$ and $\bar{I}_Y$ before the next iteration. Thus, in each iteration, x and y must be 'new' indices that have not been merged with each other in a previous iteration. By induction on tree structures, it is easy to see that two distinct indices cannot be merged with each other more than once. Hence, the algorithm cannot generate duplicate indexings.

2. We now demonstrate why the algorithm generates exactly the correct number of indexings by means of a simple example. Without loss of generality, consider the right-branching phrase scheme shown in figure 3.14.

   Now consider the decision tree shown in figure 3.15 for computing the possible indexings of the right-branching tree in a bottom-up fashion.

   Each node in the tree represents the index set of the combined phrase depending on whether the noun phrase at the same level is cross-indexed or not. For example, $\{i\}$ and $\{i, j\}$ on the level corresponding to NP_j are the two possible index sets for the phrase P_{ij}. The path from the root to an index set contains arcs indicating what choices (either to coindex or to leave free) must have been made in order to build that index set. Next, let us just consider the cardinality of the index sets in the decision tree, and expand the tree one more level (for NP_l) as shown in figure 3.16.

   Informally speaking, observe that each decision tree node of cardinality i 'generates' i child nodes of cardinality i plus one child node of cardinality

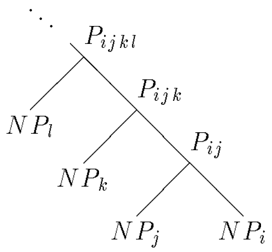


Figure 3.14 Right-branching tree.

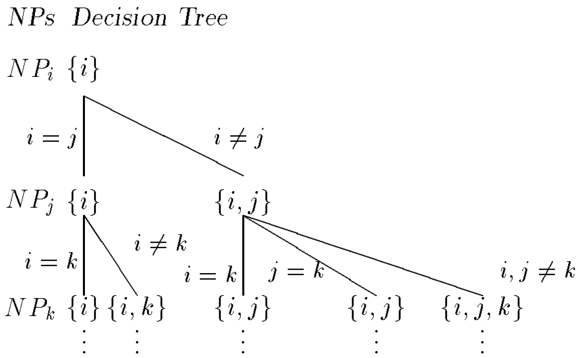


Figure 3.15 Decision tree.

$i+1$. Thus, at any given level, if the number of nodes of cardinality m is c_m, and the number of nodes of cardinality $m-1$ is c_{m-1}, then at the next level down, there will be $mc_m + c_{m-1}$ nodes of cardinality m. Let $c(n,m)$ denote the number of nodes at level n with cardinality m. Let the top level of the decision tree be level 1. Then:

(29)

$$c(n+1, m+1) = c(n, m) + (m+1)c(n, m+1)$$

Observe that this recurrence relation has the same form as equation (20). Hence the algorithm generates exactly the same number of indexings as de-

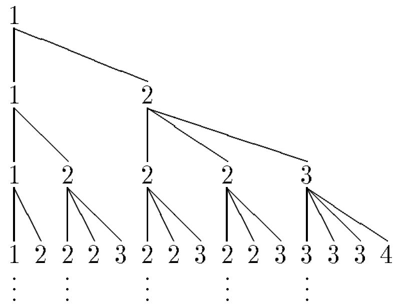


Figure 3.16 Condensed decision tree.

manded by combinatorial analysis.

The Representation of Free Indexation

Figure 3.17 contains the definition of free indexation. The first form is a direct instantiation of the compositional definition. The first line states that `freeIndexation` maintains a list of indexed noun phrases for each node of a phrase structure tree. The rest of the definition is divided into two parts:

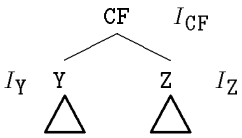
1. The first part encodes the case for binary branching. Here, `CF` names a constituent with two immediate sub-constituents `X` and `Y`. (The next three lines just names the indexed NPs associated with each of these three constituents.) It specifies that the indexed NPs for `CF` are defined by cross-indexing the corresponding sets for `X` and `Y` and then taking the union of the two sets, as required by the compositional definition. The system will automatically generalize the procedure to unary and ternary branching (and above) by iteration (using the method described on page 96). In any case, no matter how many branches there are, the additional step to take care of the case when `CF` itself may be indexed is handled by the final call to `freelyIndexIfNP/3`.

2. The second part encodes the base case, i.e. when there is a noun phrase in an A-position.

The remaining predicates are a straightforward encoding of the `crossIndex` and Index Set Propagation operations defined in (27) and (28).

[16]To make the boundary cases match, just define $c(0,0)$ to be 1, and let $c(0,m) = 0$ and $c(n,0) = 0$ for $m > 0$ and $n > 0$, respectively.

```
:- freeIndexation produces [INPs]
        compositional_cases_on CF where   % Compositional case:
          CF with_constituents Y and Z st
                Y produces [IY],
                Z produces [IZ],
                CF produces [ICF]
        then crossIndex(IY,IZ),
             unionWrtI(IY,IZ,ICF)
        finally freelyIndexIfNP(CF,ICF,INPs)
      else
        npInApos(CF) then index(CF,_), INPs=[CF]. % Base case.
```




```
npInApos(CF) :- cat(CF,np), CF has_feature apos.

crossIndex(IX,IY) :-
        diffWrtI(IX,IY,IbX),              % $\bar{I}_X = I_X - I_Y$
        diffWrtI(IY,IX,IbY),              % $\bar{I}_Y = I_Y - I_X$
        freelyCoindex(IbX,IbY).

% freelyCoindex: $I_X \times I_Y$
freelyCoindex([],_).                      % Do nothing,  $I_1 = \emptyset$
freelyCoindex([NP|IX],IY) :-
        optCoindex(NP,IY,IYp),
        freelyCoindex(IX,IYp).

% optCoindex: $NP \times I \mapsto I'$
optCoindex(_,I,I).                        % Do nothing, or ...
optCoindex(NP1,I,Ip) :-                   % Select an NP in I to coindex with.
        select(NP2,I,Ip),                 % $NP_2 \in I, I' = I - \{NP_2\}$
        agree(NP1,NP2),
        coindex(NP1,NP2).

coindex(X,Y) :- index(X,I), index(Y,I).

index(X,I) :-
        X has_feature index(I)            % Is X already indexed?
        -> true
        ;  addFeature(index(I),X).        % If not, give it one.

freelyIndexIfNP(NP,I,Ip) :-
        npInApos(NP)
        -> index(NP,_),                   % Give the NP an index feature.
           crossIndex([NP],I),
           unionWrtI([NP],I,Ip)
        ;  Ip = I.                        % Do nothing if not an NP
```

Figure 3.17 Representation of the free indexation algorithm.

This completes our formalization of free indexation. In the next section, we will turn to describe how chain formation, a similar but more complex procedure, can also be encoded using structural induction.

3.4.3 Compositional Chain Formation

In the following discussion, we will restrict the domain of application of Move-α to instances of syntactic movement, i.e. between D- and S-structure, of NPs only. Furthermore, we will assume that syntactic movement of NPs involves substitution only.[17] Movement of *wh*-adverbs such as *why* and *how* are handled by the same mechanism; but for simplicity of exposition, this will not be shown here. However, LF-movement of quantifiers, *wh*-elements, and operators will be covered by a separate parser operation. As chapter 4 will describe, cases of head movement are also handled by a different mechanism. Finally, we will also assume that movement must leave traces.[18]

First, let us formally define the notion of a movement chain:

(30) A (non-trivial) *movement chain* is a sequence of elements $\langle E_1, \ldots, E_n \rangle$, $n \geq 2$, such that E_n occupies the position occupied by E_1 before movement, and that each subsequence $\langle E_i, E_{i+1} \rangle$, for $i \geq 0$, represents a single instance of movement of E_1 from the position occupied by E_{i+1} to that occupied by E_i. Elements E_2 to E_n are *traces*. Furthermore, all elements of a chain will be coindexed. A *trivial chain* involving no movement will simply be a singleton sequence $\langle E_1 \rangle$.

(Unless otherwise stated, hereafter, we will use the term 'chain' to denote a 'non-trivial movement chain'.) The chain formation algorithm is based on the following conditions:

(31) a. [*NP*-e] optionally participates in a chain.
b. An overt NP optionally heads a chain.

[17] In the current system, a mechanism for adjunction of NPs was found to be only necessary for cases of multiple-*wh*-movement, e.g. as in *who will read what?* Hence, the current chain formation operation for syntactic movement lacks this capability. However, since such cases do occur at LF, the operation that implements LF-movement does indeed incorporate a mechanism for adjunction. (See Lasnik & Saito [34] for the proposal that adjunction to Comp is universally prohibited in the syntax, and only possible in LF.)

[18] However, a trace may be marked as being invisible to the ECP. For example, consider the (ill-formed) S-structure *who*$_i$ *do you think* [*CP* t'_i [$\bar{C}$ *that* [*IP* t_i *saw Bill*]]]. Here, we have assumed that *who* has been moved from the embedded subject position occupied by t_i in two steps (in order not to violate Subjacency). According to Lasnik & Uriagereka, this structure violates the ECP because t_i is not properly governed. For this to happen, it must be the case that the intermediate trace t'_i should not function as a proper governor. One way to guarantee this is to say that movement will optionally leave a trace. Since we have assumed that movement must leave traces, the same effect can be obtained by marking t'_i as being invisible to the ECP.

c. An element cannot participate in more than one chain.
d. A chain is *complete* if and only if it is headed by a non-trace element. All chains must be complete.

The first two conditions allows both possibilities for movement discussed above in example (13). Condition (31c) ensures that two movement chains cannot share common heads or traces. Note that condition (31d) prevents the movement of traces. However, the same condition does not forbid a chain from being headed by an empty element as long as it is not a trace. For example, the chain conditions are compatible with the following analysis for the relative clause construction *the man I saw* (along the lines of Chomsky [12]):

(32) $[_{NP}\ [_{NP}$ the man $]\ [_{CP} Op_i\ [_{\bar{C}}\ [_{C}\ ]\ [_{IP}$ I saw $[_{NP}$-t $]_i$]]]]

Here, the empty operator Op_i is assumed to head the chain $\langle Op_i,\ [_{NP}\text{-t}\ ]\rangle$.

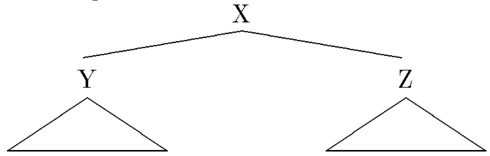
As with free indexation, the present system does not have the ability to automatically synthesize a constructive chain formation algorithm from the definitions given in (30) and (31). Hence, we must suply it with one. We now proceed to a description of an algorithm defined recursively over tree structures that builds chains in accordance with the chain conditions. For each constituent X in a structure, we will associate X with two sets: (1) the set of incomplete chains in X, to be denoted by C_X, and (2) the set of 'free' NPs in X, to be denoted by F_X, i.e. those NPs that do not participate in a non-trivial movement chain in X. There are two cases to consider. (For simplicity, we will omit for now the encoding of chain condition (31d), which only applies to the root node.

1. Base Case: X is a leaf node.
   (a) X is an empty noun phrase, i.e. $[_{NP}$-e]. In accordance with chain condition (31a), there are two possibilities:
      i. $[_{NP}$-e] may be free in X. Hence $C_X = \emptyset$ and $F_X = \{\ [_{NP}\text{-e}\ ]\}$, or
      ii. $[_{NP}$-e] may participate in a chain in X. Hence, $C_X = \{\langle\ [_{NP}\text{-e}\ ]\rangle\}$ and $F_X = \emptyset$.
   (b) X is not an empty noun phrase. Then $C_X = \emptyset$ and $F_X = \emptyset$.

   In either case, the chain conditions will be satisfied.

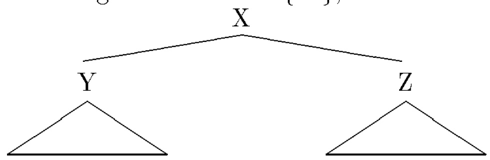
2. Compositional Case: X is an interior node.
   (a) Unary branching: Suppose X has a single immediate constituent Y. Then X inherits the chains and free elements of Y: that is, $C_X = C_Y$ and $F_X = F_Y$. Since no additional elements are introduced by X, it is clear that if Y satisfies the chain conditions, then so will X.

(b) Binary branching: Suppose X has two immediate constituents Y and Z:



$C_Y = \{c_1, \ldots, c_m\}$, $m \geq 0$ $\quad C_Z = \{c_{m+1}, \ldots, c_p\}$, $p \geq m$
$F_Y = \{f_1, \ldots, f_n\}$, $n \geq 0$ $\quad F_Z = \{f_{n+1}, \ldots, f_q\}$, $q \geq n$

The possible (partial) chains for X will simply be the union of the corresponding chains for Y and Z, possibly extended by the free elements of Y and Z. For example, consider the case when Y has a singleton chain $\langle e1 \rangle$ together with free elements $\{John, e4, e5\}$, and Z has a single chain $\langle e2, e3 \rangle$ and a single free element $\{e6\}$, where $e1$ to $e6$ are all empty NPs:



$C_Y = \{\langle e1 \rangle\}$ $\quad C_Z = \{\langle e2, e3 \rangle\}$
$F_Y = \{John, e4, e5\}$ $\quad F_Z = \{e6\}$

For example, one possibility would be to use $e6$ from Z to prefix the chain $\langle e1 \rangle$ from Y, thus obtaining the extended chain $\langle e6, e1 \rangle$. In general, the extensions, or prefixes, for either chain must be drawn from one of the 65 possible r-permutations of the available free elements shown in figure 3.18.[19] Here, each node corresponds to a distinct r-permutation of the free elements $John$, $e4$, $e5$ and $e6$. Of course, not every permutation need be a valid prefix. For example, $\langle e4, John \rangle$ is not a valid prefix because intermediate elements in a chain must be traces, as defined by (30). (Those r-permutations that are also valid prefixes have been marked in figure 3.18.)

We now proceed to define a non-deterministic operation `merge1` that extends chains using the possible r-permutations of the available free elements. Let `merge1` take as input a set of chains C and two sets of free elements F_1 and F_2. `merge1` will return C', the set of chains after extension, and F_1' and F_2', representing the 'unused' elements of F_1 and F_2, respectively:

(33)

[19] An r-permutation of n things is an ordered selection (without replacement) of r of them. Let P_n denote the sum of r-permutations for $r = 0, 1, \ldots, n$. In general, P_n will be the nearest integer to $n!e$ (Riordan [47]).

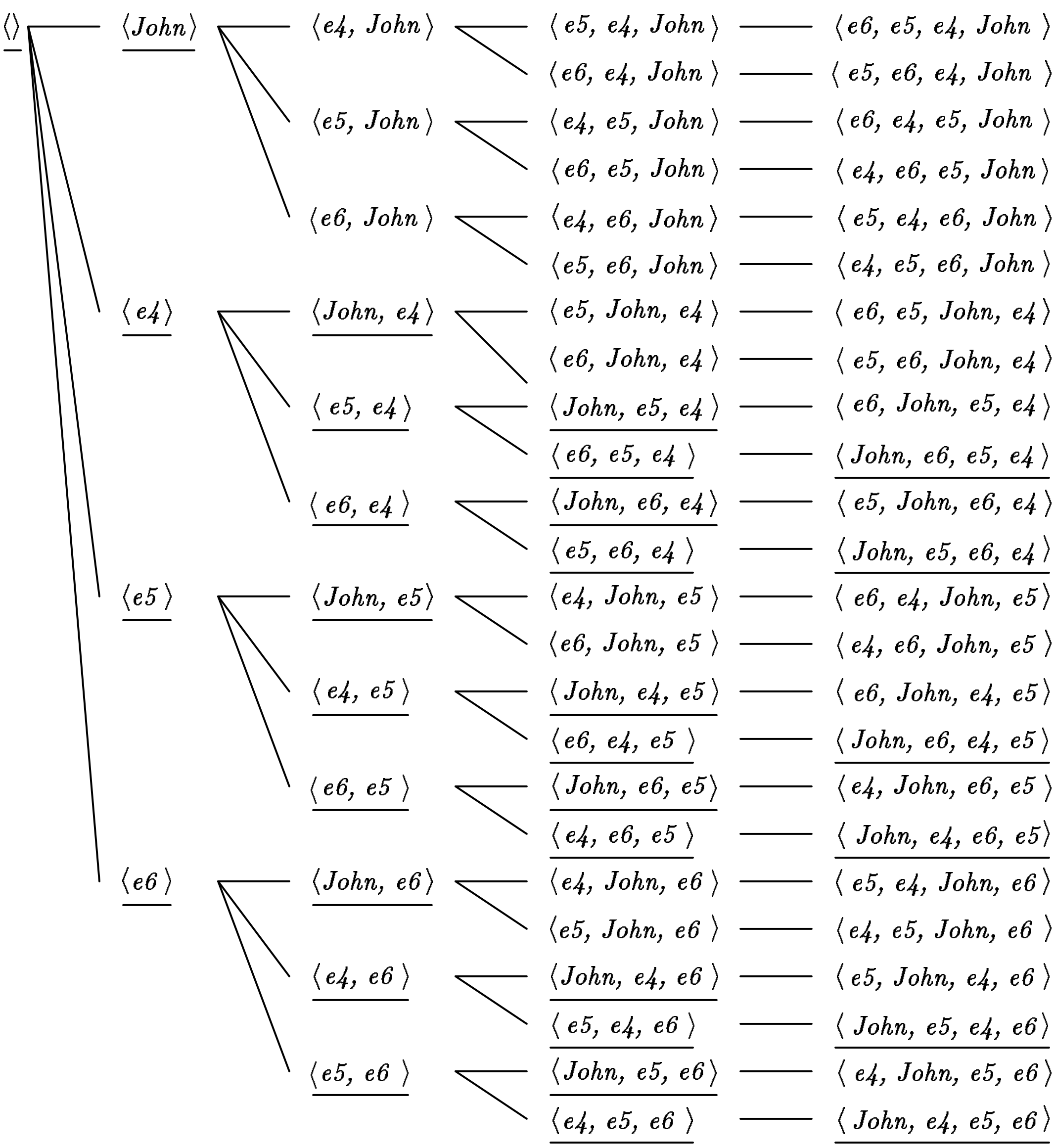


Figure 3.18 r-Permutations of $\{John, e4, e5, e6\}$.

Operation: `merge1` $: C \times F_1 \times F_2 \mapsto C' \times F_1' \times F_2'$.
<u>Either</u>:
1. Do nothing, i.e. let $C' = C$, $F_1' = F_1$ and $F_2' = F_2$,
<u>Or</u>:
2a. <u>Select</u> an element $f \in F_1 \cup F_2$, and
2b. <u>Select</u> a chain $c = \langle E_1, \ldots, E_n \rangle \in C$, and
2c. Let c' be the chain $\langle f, E_1, \ldots, E_n \rangle$, and
2d. Let C', F_1' and F_2' be the result of `merge1`(C'', F_1'', F_2'') where:
i. $C'' = (C - \{c\}) \cup \{c'\}$ if c' is incomplete,
otherwise let $C'' = C - \{c\}$,
ii. Let $F_1'' = F_1 - \{f\}$ if $f \in F_1$, otherwise $F_1'' = F_1$,
iii. $F_2'' = F_2 - \{f\}$ if $f \in F_2$, otherwise $F_2'' = F_2$,

`merge1` is non-deterministic because there are three choice points in its definition. First, we can either choose to extend a chain (step 2) or do nothing (step 1). Secondly, if we choose to extend a chain, we can form distinct chains by using different selections of free elements and chains at steps (2a) and (2b). Note also that `merge1` is guaranteed to terminate since F_1 and F_2 are finite sets. To show that `merge1` is correct, we need to show (1) that the chain conditions are preserved by `merge1`, and (2) that all possible extensions are permitted by the choice points in `merge1`:

i. Preservation of Chain Conditions:
Since X is an interior node, the requirement that [*NP*-e] be allowed to participate in a chain, (31a), is trivially satisfied. Ignoring (31b) for the present, obviously, the assignment of an element to at most one chain, condition (31c), is trivially preserved when step (1) is taken. Let us consider the case when step (2) is taken instead. Here, parts (ii–iii) of step (2d) will ensure that all free elements will be removed from further consideration once they have been selected to extend a chain. Hence, condition (31c) is satisfied in either case.

ii. All Possible Extensions:
In general, the possible r-permutations of a set can be generated recursively by a number of different methods (see Page & Wilson [41] or Reingold *et al.* [45]). The permutation tree shown previously in figure 3.18 illustrates one such method. Here, each r-permutation (of length r), $r \geq 0$, is extended to derive the possible $(r+1)$-permutations by prefixing it with one of the unused free elements. The recursive procedure for `merge1` is basically an implementation

of this method.[20] For simplicity, suppose there is only one chain c in C. If step (1) is chosen, then `merge1` will return c unmodified. This is equivalent to prefixing c with the 0-permutation, or the empty sequence, at the root of the permutation tree. If step (2) is chosen instead, then c will be prefixed with one of the free elements. This is equivalent to moving from the root down to one of the singleton prefixes in the permutation tree. Next, `merge1` will be applied recursively. In the second application, we can choose to either remain at the current node in the permutation tree or move to an descendant one level further down, and so on. Moreover, the removal of completed chains in part (i) of step (2d) corresponds to the pruning of invalid prefixes as illustrated in figure 3.18. By induction on the recursive step, it is clear that `merge1` can visit all the valid prefixes in the permutation tree. Finally, the selection step in (2b) generalizes the method for the case when C happens to contain more than one chain. Hence, each time a particular chain gets selected, we move to a node one level further down in the permutation tree for that chain.[21]

For binary branching, we must apply `merge1` twice: once to extend the chains in Y and once more to extend the chains in Z. Let `merge2` represent the following composite operation:

(34) Operation: `merge2` : $(C_Y, F_Y) \times (C_Z, F_Z) \mapsto (C_X, F_X)$
1. Let `merge1`(C_Y, F_Z, F_Y) produce C'_Y, F'_Z and F'_Y.
2. Let `merge1`(C_Z, F'_Y, F'_Z) produce C'_Z, F''_Y and F''_Z.
3. Then let $C_X = C'_Y \cup C'_Z$ and $F_X = F''_Y \cup F''_Z$.

Notation: For simplicity, we will sometimes represent a pair of chains and free elements, e.g. (C_W, F_W), by the node to which the pair belongs, e.g. W in this case.

The last step indicates that the set of chains for X is just the union of the sets of chains for Y and Z extended by `merge1`. Similarly, the free elements of X are simply those free elements of Y and Z not incorporated into chain extensions by `merge1`.[22]

[20] All possible selections may be explored by a backtrack algorithm that systematically explores each alternative at all three choice points.

[21] Of course, since a free element will be removed from the appropriate free set once it has been selected to extend a chain, the prefixes represented by the current nodes in the permutation trees will be disjoint at any given point during the application of `merge1`.

[22] Note that `merge1` must be applied serially to preserve the property that a free element cannot be used to extend more than one chain.

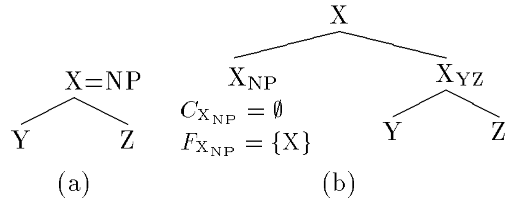


Figure 3.19 Expansion of binary branching when X=NP

We now return to consider chain condition (31c), which states that an overt NP has the option to head a (non-trivial) chain. If the interior node X itself happens to be a noun phrase, e.g. as in [*NP* [*NP who*] *that John knows*], then we must allow for the possibility that the noun phrase has been moved, as in [*NP who that John knows*]$_i$ *does he like* t_i (example [1:15a] from Lasnik & Uriagereka). In this case we can proceed as follows:

(35) 1. Let $C_{X_{NP}} = \emptyset$ and $F_{X_{NP}} = \{X\}$.
2. Let $C_{X_{YZ}}$ and $F_{X_{YZ}}$ be $\texttt{merge2}(Y, Z)$.
3. Then let C_X and F_X be $\texttt{merge2}((C_{X_{NP}}, F_{X_{NP}}), (C_{X_{YZ}}, F_{X_{NP}}))$.

Here, X will be allowed the option of participating in chain formation at step 3, as required by condition (31c). This procedure is equivalent to treating binary branching of the form shown in figure 3.19a as being equivalent (with respect to chain formation only) with the repeated binary branching structure shown in figure 3.19b.

(c) n-ary branching: Along the same lines, general n-ary branching, for $n > 2$, may be treated as instances of repeated binary branching. For example, suppose X has three constituents U, V and W. Then the chains and free elements of X will be given by $\texttt{merge2}(U, \texttt{merge2}(V, W))$.

We now return to consider the final chain condition, (31d), which states that all chains in a structure must be complete. Let the root node be denoted by S. Then, (31d) is equivalent to the following constraint:

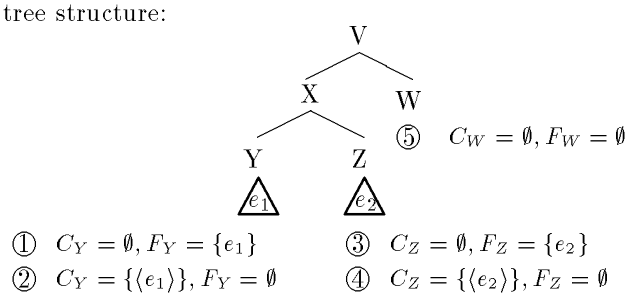
(36) $C_S = \emptyset$

That is, there can be no partial chains (left) at the root node.

To summarize, the chain formation procedure described so far admits all possible chain permutations that satisfy the chain conditions in (31). But, before going on to illustrate how the procedure is actually represented, we briefly consider two additional constraints to be imposed on chain formation.

Eliminating Redundancies in Chain Formation

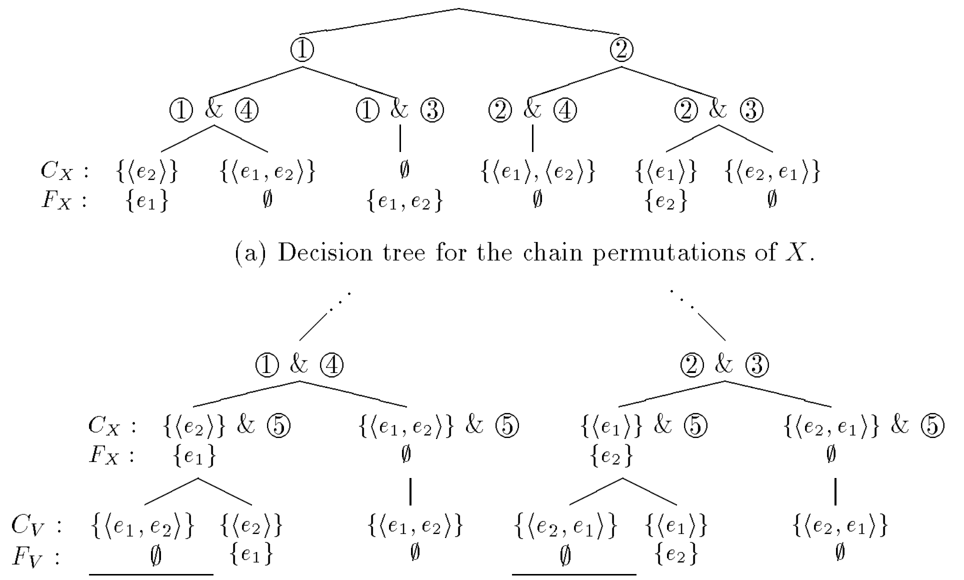
The chain formation procedure overgenerates in the sense that it is capable of producing some chain permutations in more than one way. For example, consider the followng tree structure:



In particular, consider the node X that dominates two leaf nodes, Y and Z, both containing a single empty NP. Note that each child of X has two possible chain permutations, labelled as ① and ② for Y, and ③ and ④ for Z. Applying `merge2` to each chain permutation for Y and Z in turn, we can deduce that X has six possible chain permutations, as the decision tree in figure 3.20a illustrates. Let us now consider the possible chain permutations for node V, i.e. X's parent. Here, V also dominates a leaf node W such that $C_W = F_W = \emptyset$. Since W does not contribute to chain formation, it should be the case that V directly inherits the six permutations for X. However, the algorithm described so far will generate eight chain permutations for V, two of which will be repeated, namely $C_V = \{\langle e_1, e_2 \rangle\}, F_V = \emptyset$ and $C_V = \{\langle e_2, e_1 \rangle\}, F_V = \emptyset$. Figure 3.20b.(i–ii) illustrates the expansion of the decision tree in figure 3.20a for these two cases. The redundancy arises because the algorithm allows e_1 (*resp.* e_2) to extend the chain $\langle e_2 \rangle$ (*resp.* $\langle e_1 \rangle$) either when the chains for Y and Z are merged, or at the next stage when the chains for X and V are merged. To avoid generating these duplicate permutations, a free element must be permitted *exactly* one opportunity to extend any given chain. In terms of the definition of `merge2` given in (34), this can be assured by adding the following requirement:

(37) The first element chosen to extend any chain from C_Y in step 1 (*resp.* C_Z in step 2) must not come from F_Y (*resp.* F_Z).

Returning to the example in figure 3.20b.(i), e_1 may be used to extend $\langle e_2 \rangle$ when the chains for Y and Z are merged because e_1 and e_2 belong to different branches. However, e_1 cannot be used to extend $\langle e_2 \rangle$ once Y and Z have been merged, e.g. when constructing the chains for V or for any ancestor of V, because they will then be part of the same sub-tree. Hence, e_1 has exactly one opportunity to extend $\langle e_2 \rangle$; that is, when the chains for Y and Z are merged. Hence, the duplicate chain



(i) Generating $C_V = \{\langle e_1, e_2\rangle\}, F_V = \emptyset$. (ii) Generating $C_V = \{\langle e_2, e_1\rangle\}, F_V = \emptyset$.

(b) Extensions of the decision tree in (a).

Figure 3.20 An example of redundant chain formation.

permutation in figure 3.20b.(i) (shown underlined) will be eliminated. A similar argument holds for the other case, i.e. the extension of $\langle e_1\rangle$ by e_2 in figure 3.20b.(ii).

In general, requirement (37) will not compromise the ability of `merge2` to produce all possible chain permutations.[23] In other words, the extra requirement only

[23] More formally, this can be proved by induction on tree structures:

1. *Base case:* We have seen that the algorithm correctly generates all possible chain permutations for leaf nodes. Since requirement (37) only applies to the compositional case, correctness for the base case is trivially preserved.
2. *Induction step:* Suppose that the algorithm correctly generates all possible chain permutations for two distinct sub-trees T_1 and T_2. Let node N be the parent of both T_1 and T_2. Now, consider the merging of the chains from T_1 and T_2 for node N. We know from earlier discussion that `merge2` produces all possible chain permutations for N. But requirement (37) will eliminate all instances generated by `merge2` that (initially) extends a chain with a free element from the same sub-tree. Let e and $c = \langle c_1, \ldots, c_n\rangle$ be any such free element and chain, respectively. Then (37) eliminates $c' = \langle e, c_1, \ldots, c_n\rangle$ generated for N in this manner. But, by the induction hypothesis, c' is also a chain permutation of T_1 (*resp.* T_2), assuming e is from T_1 (*resp.* T_2), since e and the elements of c all belong to T_1 (em resp. T_2). Hence, (37) eliminates duplicates.

$$[NP \cdots [NP\text{-t} \ ]_i \ \cdots]_i$$

Figure 3.21 An example of an antecedent that dominates its own trace.

eliminates instances of chain permutations that are also generated by the algorithm in other ways. In fact, the algorithm will now only generate distinct chain permutations.[24] Combining these two properties, we have that the augmented algorithm generates all and only the possible chain permutations.

Eliminating Antecedents with Embedded Traces

Finally, the chain formation procedure also overgenerates in the sense that it is capable of constructing certain chains that cannot be produced by Move-α. For example, Move-α cannot generate an instance of an antecedent that dominates its own trace, as shown in figure 3.21. Therefore, the actual procedure must also contain an *"i-within-i"* condition that must be satisfied by all chains. In terms of the definition of `merge1` given in (33), the extended chain c' (constructed in step (2c)) must satisfy the following extra condition:

(38) $\forall i, 1 \leq i \leq n, \quad f \not\preceq E_i$

where the symbol $\preceq$ denotes the dominates relation.

The Representation of Chain Formation

The chain formation algorithm described above may be encoded in a fairly direct manner, as shown in figure 3.22. Here, the predicate `chainFormation/3` takes as

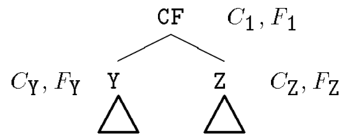
[24] As in the previous note, this can be proved by induction on tree structures. We augment the proof in the previous note as follows:

1. *Base case:* From the description of the algorithm, a leaf node can produce either one or two distinct chain permutations depending on whether it is an empty NP or not. Hence, the algorithm correctly generates all possible chain permutations *without duplicates* for leaf nodes.

2. *Induction step:* Suppose all possible chain permutations without duplicates are generated for T_1 and T_2. We now have to show that no duplicates can be introduced by extending the chains of T_1 and T_2. Since the first element, say e, added to any chain, say $c = \langle c_1, \ldots, c_n \rangle$, in T_1 (*resp.* T_2) must come from T_2 (*resp.* T_1), it is clear that the extended chain $\langle e, c_1, \ldots, c_n \rangle$ could not have been generated previously within either T_1 or T_2. Of course, subsequent extensions of this 'new' chain using free elements from either sub-tree does not alter the fact that it could not have been generated at a lower level. Also, since a free element cannot be selected twice, all prefixes, i.e. r-permutations of free elements, generated by `merge2` must be distinct. Hence all chain permutations for N must be distinct.

```
% (31d) All chains must be complete.
:- chainFormation(CF) top chainFormation(CF,[],_). % (36)  $C_S = \emptyset$

:- chainFormation produces [Chains,FreeSet]
        compositional_cases_on CF where    % Compositional case:
          CF with_constituents Y and Z st
                Y produces [CY,FY],
                Z produces [CZ,FZ],
                CF produces [C1,F1]
          then  merge2(CY,FY,CZ,FZ,C1,F1)
          finally extendChainIfNP(CF,C1,F1,Chains,FreeSet)
        else
          emptyNP(CF) then optStartChain(CF,Chains,FreeSet). % Base case.
```




```
emptyNP(NP) :- cat(NP,np), ec(NP).

% (optionally) start a chain
optStartChain(NP,[[NP]],[]).                % $C_{NP} = \{\langle NP \rangle\}, F_{NP} = \emptyset$
optStartChain(NP,[],[NP]).                  % $C_{NP} = \emptyset, F_{NP} = \{NP\}$

% (33) merge1 : $C \times F_1 \times F_2 \mapsto C' \times F_1' \times F_2'$
merge1(C,F1,F2,C,F1,F2).                    % Do nothing, or ...
merge1(C,F1,F2,Cp,F1p,F2p) :-
        selectElement(F,F1,F2,F1pp,F2pp),   % $F \in (F_1 \cup F_2), F_1'' = F_1 - \{F\}, F_2'' = F_2 - \{F\}$
        extendAChain(F,C,C1),               % C1 is C with some c $\in$ C extended by F
        merge1(C1,F1pp,F2pp,Cp,F1p,F2p).

% (34) merge2 : $(C_Y, F_Y) \times (C_Z, F_Z) \mapsto (C_X, F_X)$
merge2(CY,FY,CZ,FZ,CX,FX) :-
        merge1(CY,FZ,FY,CYp,FZp,FYp),       % Extend chains in $Y$.  $C_Y \mapsto C_Y'$
        merge1(CZ,FYp,FZp,CZp,FYpp,FZpp),   % Extend chains in $Z$.  $C_Z \mapsto C_Z'$
        union(CYp,CZp,CX),                  % $C_X = C_Y' \cup C_Z'$
        union(FYpp,FZpp,FX).                % $F_X = F_Y' \cup F_Z'$

% (35)
extendChainIfNP(NP,CYZ,FYZ,CX,FX) :-
        cat(NP,np)
        -> merge2([],[NP],CYZ,FYZ,CX,FX)    % merge2$((\emptyset, \{NP\}), (C_{X_{YZ}}, F_{X_{YZ}}))$ if NP
        ;  CYZ = CX, FYZ = FX.              % Do nothing if not an NP
```

Figure 3.22 Representation of the chain formation algorithm.

input a constituent `CF` in which chains are to be assigned, and produces as output an assignment of partial chains `Chains` and free elements `FreeSet` for that constituent. The top-level interface to chain formation is formed by a predicate with the same name, but of arity one instead. This top-level predicate (given a complete structure as input) will hold only if `chainFormation/3` holds with no partial chains (left) at the root. Hence, this 'top' condition encodes chain condition (31d), which states that all chains must be complete.[25] The remaining predicates shown directly encode each of the various chain formation operations in turn, namely chain condition (31a), `merge1`, `merge2` and chain condition (31c) (as realized by (35)). As with free indexation, the non-determinism present in these definitions will be automatically handled by the underlying PROLOG search mechanism. For example, chain condition (31a) is encoded (in part) by the two clauses that constitute predicate `optStartChain`. The choice between the two clauses represents the non-determinism present in the condition; that is, the option to participate in a (non-trivial) chain or not. Similarly, `merge1` is defined by two clauses, representing the choice between step 1 and step 2 in the procedure given in (33). The remaining two choice points in that procedure are contained in `selectElement` and `extendAChain` (neither definition shown here). Condition (37), which prevents the generation of duplicate chain permutations, is also enforced within these two predicates.[26] Finally, condition (38), which eliminates cases of impossible movement, is also enforced within `extendAChain`.[27]

This completes our discussion of how compositional definitions are used to represent combinatorial operations. In the next section, we will go on to illustrate how principles that involve *minimality* such as the Binding conditions may be con-

[25] Note that no such constraint is placed on the corresponding set of free elements, as indicated by the *don't care* '_' variable. In other words, any number of elements may remain free.

[26] There are numerous ways to encode condition (37). The actual implementation makes use of a simple marking feature associated with both chains and free elements. All chains and elements are initially unmarked when `merge1` is called in `merge2`. Also, marking will be local to `merge1`, i.e. the chains and free elements returned to `merge2` will not be marked. All elements selected from F_2 are marked. A marked element can only extend a marked chain. An unmarked element can extend any chain. If a chain is extended by an unmarked element, then the extended chain is marked. Hence, once a chain has been initially extended by an element from F_1, it can be further extended by any element from either F_1 or F_2.

[27] The actual implementation makes use of the property that all elements in a sequence will be coindexed. Indices are implemented as syntactic features of the form `index(I)`, where `I` is (always) a logical variable. Condition (38) implies that all NPs, e.g. $\mathrm{NP}_{j_1}, \mathrm{NP}_{j_2}, \ldots, \mathrm{NP}_{j_n}$, dominated by a larger NP cannot be coindexed with the larger NP, call it NP_i. Hence, for each large NP of this sort, we maintain the following list of constraints: $\{i! = j_1, i! = j_2, \ldots, i! = j_n\}$. Here, $j! = k$ means that variable j can never be unified with variable k. Hence, every time a chain is extended by some free element `F` within `extendAChain`, these constraints will be checked when the index of `F` is unified with the index of the head of the (partial) chain. (If any constraint is violated, then backtracking will ensue.) The implementation of PROLOG used in the current system does not directly support the use of 'anti-unifiers'. Hence, the aforementioned checks have to be manually implemented. To keep the definition of chain formation simple, we have omitted the code that implements anti-unification.

veniently encoded using the `in_all_configurations` form, but are automatically transformed into a compositional fashion.

3.5 Compositional Definitions Part II: Binding Conditions

The fundamental reason why compositional definitions are necessary for free indexation and chain formation to work with principle interleaving is that in these cases the relevant syntactic relation being expressed is essentially a *non-local* (or unbounded) one. For example, any noun phrase may be potentially coindexed with another noun phrase, regardless of the 'distance' between them. The same situation holds for chain composition. (Of course, there are many principles such as the Binding conditions, Control of *PRO* and Subjacency that may or may not impose locality restrictions on the subsequent assignments.) By contrast, the definitions of structural and inherent Case assignment are local ones because Case must be assigned under government, a *local* relation. In this section, we turn to the formulation of Binding conditions that express locality restrictions on the possible referents of anaphors, e.g. *himself*, pronominals, e.g. *him*, and referential expressions, e.g. names like *John*. In chapter 1, we introduced the following two Binding conditions:

(A) An anaphor must be coreferent with an element in its clause.
(B) A pronoun must ***not*** be coreferent with any element in its clause.

We also saw that this (purely local) definition proved sufficient to explain some simple facts, repeated here as (39):

(39) a. *[John$_1$ likes him$_1$] — *cf.* [John$_1$ likes him$_2$]
b. John$_1$ believes that [Mary likes him$_1$]
c. John$_1$ believes that [Mary likes him$_2$]
d. [John$_1$ likes himself$_1$] — *cf.* *[John$_1$ likes himself$_2$]
e. John believes that [Mary$_1$ likes herself$_1$]
f. *John$_1$ believes that [Mary likes himself$_1$]

Of course, the simple notion of the relevant Binding domain being the local clause cannot account for more complex examples such as the following (taken from Lasnik & Uriagereka):

(2:26a) John likes [Mary's pictures of him]
(2:26b) *John likes [Mary's pictures of himself]
(2:29) [John likes pictures of himself]
(2:91) [The men think that pictures of each other will be on sale]
(2:103) *The men think that [Mary's pictures of each other will be on sale]

Here, as in (39), we have used brackets to delimit the extent of the relevant Binding domain in each case. The problem now is that the domain can have an arbitrarily large extent. As with free indexation and chain formation, we will need to use compositional definitions if the Binding conditions are to be compatible with the principle interleaving control model. The difference being that it will be possible to retain the transparency of a universally quantified condition over tree structures for the Binding conditions, whilst maintaining compatibility with principle interleaving by using an automatic transformation between the two representations.

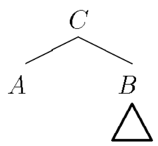
3.5.1 Using the `smallest_configuration` form

To handle the examples shown in the previous section, Lasnik & Uriagereka introduces the notion of a *governing category* (GC) as follows:

(40) (A) An anaphor must be A-bound in its GC.
(B) A pronominal must be A-free in its GC. *(adapted from (2:27))*

To be *bound* (equivalently, not *free*), an anaphor or pronominal must be coindexed with another noun phrase that *c-commands* the former noun phrase. (As with the previous definition, the same interpretation rule still hold; that is, if two elements are coindexed, then they are assumed to be coreferential.[28]) The canonical case in which the structural relation c-command obtains is the following:

(41)



Here, the constituent A c-commands B and any constituent dominated by B, where C is the first 'branching node' that dominates A. Note that the linear order of A and B is not significant. Finally, the term *A-bound* refers to being bound by an element in an A-position, i.e. a subject or object position. (Similarly, being A-free is the same as not being A-bound.)

We are now in a position to define the notion of a governing category:

(42) The GC of α is the minimal NP or S containing α,a governor of α, and a SUBJECT accessible to α. *(from ch.2, no.105)*

[28] There is one exception. The subject of a clause and inflection will be coindexed for agreement, not binding purposes. See the definition of `binds/3` in figure 3.28.

(43) β is accessible to α if *(from ch.2, no.106)*
  a. β c-commands α, and
  b. β is not coindexed with any category containing α

Here the notion SUBJECT (introduced by Chomsky [9]) extends the standard notion of subject, as in the subject of a clause, e.g. *pictures* in *pictures are on sale*, or the subject of a noun phrase, e.g. *Mary* in *Mary's pictures of him*, to include (non-overt) inflection (actually, the agreement component of inflection) present in tensed clauses.

The representation of the preceding definitions is shown in figure 3.23. The only new feature is the `smallest_configuration` construct introduced to define the notion of a governing category. It is meant to be read as follows: "`gc(X)`, the governing category of a constituent `X`, is defined to be the smallest configuration `CF` such that `CF` is a `gcDomain` (defined for `i2` and `np`) with the following components: (1) `X` itself; (2) an element `G` such that `G` governs `X` in `CF`; and finally (3) a SUBJECT `S` accessible to `X` in `CF`." (The `smallest_configuration` form is used to define many other cases of (minimal) domains. We shall see one such example shortly in the definition of Condition C.) We will omit discussion of the other predicates shown in the figure 3.23 since they are fairly straightforward translations of the appropriate definitions.[29]

The `smallest_configuration` form define an operation `gc(X,GC)` that can be called within the context of a universally quantified condition over tree structures, as will be explained below.

3.5.2 Reintroducing the `in_all_configurations` Form

Condition A of the Binding theory (repeated below) expresses a universal condition on anaphors:

(44) An anaphor must be A-bound in its GC.

Compare (44) with its representation shown in figure 3.24. The universal condition has the following reading: "in every configuration `CF` such that `CF` is an anaphor, then *if* `GC` is identified to be the governing category for `CF`, then `CF` must be A-bound in `GC`." The definition of an anaphor as a [+anaphoric] noun phrase and an A-bound constituent as an element that is bound by another element `A` in an A-position are both straightforward translations. However, the feature that distinguishes such a definition from the parallel (but strictly local) definition of, say, the Case filter, as shown earlier in figure 3.3, is the presence of the goal `gc(CF,GC)` that identifies a

[29] As with the *i-within-i* condition for chain formation (described in note 27), we have omitted the explicit constraint checks necessary to fully implement the notion of an accessible SUBJECT from the description in figure 3.23.

```
gcDomain(i2).  gcDomain(np).              % theory parameter.

:- gc(X) smallest_configuration CF st cat(CF,C), gcDomain(C)
        with_components
                X,
                G given_by governs(G,X,CF),
                S given_by accSubj(S,X,CF).

accSubj(Subj,A,CF) :-                     % Subj is accessible to A.
        bigSubject(Subj,A,CF),
        c_commandsBT(Subj,A,CF).

bigSubject(SUBJ,_,XP) :- subject(SUBJ,XP). % ordinary subject.
bigSubject(AGR,A,IP) :-                   % I(AGR)
        cat(IP,i2),
        IP has_feature agr(_),
        ISpec specifier_of IP,
        \+ properlyDominates(ISpec,A),
        AGR head_of IP.

% An ordinary subject.
subject(SUBJ,IP) :- cat(IP,i2), SUBJ specifier_of IP.
subject(SUBJ,NP) :- cat(NP,np), SUBJ specifier_of NP, cat(SUBJ,np).

properlyDominates(X,Y) :- \+ X=Y, X dominates Y. % defined in figure 3.28.

% Normal c-command + I(AGR) c-commands subject of IP.
% Compare this definition to the one shown in figure 3.28.
c_commandsBT(AGR,NP,IP) :-
        cat(IP,i2),
        IP has_feature agr(_),            % ..using inheritance of features.
        NP specifier_of IP,
        AGR head_of IP.
c_commandsBT(A,B,S) :-
        S has_constituents Cs,
        in(A,C1,Cs),                      % A,C1∈Cs, A≠C1.
        C1 dominates B.
c_commandsBT(A,B,S) :-
        S has_constituent C,
        c_commandsBT(A,B,C).
```

Figure 3.23 Governing Category.

```
:- conditionA in_all_configurations CF
        where anaphor(CF) then gc(CF,GC), aBound(CF,GC).

anaphor(NP) :- cat(NP,np), NP has_feature a(+).

aBound(B,S) :- binds(A,B,S), A has_feature apos.
```

Figure 3.24 The definition of condition A.

```
conditionA produces [CFs1]
        compositional_cases_on CF where
        CF with_constituents A and B st % Binary branching case.
                A produces [As],
                B produces [Bs],
                CF produces [CFs]
        then app1(As,Bs,CFs)          % append As to Bs ↦ CFs.
        finally conditionAml(CFs,CF,CFs2),
                (anaphor(CF)          % Is CF itself an anaphor?
                 -> add(CF,CFs2,CFs1) % If so, add it to the list.
                 ;  CFs1 = CFs2)      % If not, no change.
        else anaphor(CF) then initgc(CF,GC), CFs1 = [GC]. % Base case.
% GCs × CF ↦ GCs'.
% GCs' represents those elements of GC that remain incomplete in CF.
conditionAml([],_,[]).
conditionAml([GC|GCs],CF,GCs1) :-
        gc(X,GC,CF),                  % Try to complete the GC in CF.
        (completegc(GC)               % Is it now complete?
        -> abound(X,GC),              % If so, see if X is A-bound in it.
           GCs1 = GCs2                % If so, cond. A satisfied. Discard X.
        ;  GCs1 = [GC|GCs2]),         % If not, keep GC around.
        conditionAml(GCs,CF,GCs2).    % Go on to the next element...
```

Figure 3.25 The compositional version of condition A.

```
:- conditionB in_all_configurations CF
        where pronominal(CF) then gc(CF,GC), \+ aBound(CF,GC).

pronominal(NP) :- cat(NP,np), NP has_feature p(+).
```

Figure 3.26 Condition B.

configuration strictly larger than `CF`, i.e. `CF` is only an anaphor, but `GC` must include `CF`. In the current system, the definition of condition A is automatically transformed into the equivalent compositional definition shown in figure 3.25. We will not go into the details of the transformed definition except to note that the definition basically computes the list of anaphors with incomplete governing categories associated with every node in a phrase structure.[30]

We should also point out here that the grammar writer never needs to know (nor explicitly access) the reformulated version.

Finally, note that Condition B (repeated below) can also be encoded in a parallel fashion to Condition A. Compare the statement in (45) to the form in figure 3.26.

(45) A pronominal must not be A-bound in its GC.

In the next section, we will discuss one other example involving the `smallest_configuration` form, and outline how we can take advantage of the transformation process to produce not only a more perspicuous definition, but also a more efficient one.

Condition C of the Binding Theory

Transforming a definition into a compositional one has other benefits besides compatibility with the principle interleaving model. The chief benefit is the automatic elimination of some unnecesary work. To illustrate how this works, let us consider one last principle of the Binding theory, namely condition C which applies to referential expressions. The following definition is taken from Chomsky [12]:

(46) An r-expression must be A-free (in the domain of its operator)
(= (3:56))

[30] The compositional algorithm is actually much simpler than it appears. The `compositional_cases_on` form just wraps the standard inductive tree definition around the computation of the governing category in `conditionAml/3`. That is, `conditionAml/3` will be called for every interior node in a given structure (in a bottom up fashion). `conditionAml/3` will try to complete the GC for each unbound anaphor in each interior node. If a GC has been found, it calls `aBound/2` as required by the original definition in figure 3.24. Otherwise, to avoid unnecessary recomputation, it saves the partially computed GC for the next time around. Because of the bottom-up strategy used, the compositional definition is guaranteed to find the (smallest) GC.

```
:- conditionC in_all_configurations CF
        where rExpression(CF) then aBinderDom(X,D), aFreeCond(X,D).

rExpression(NP) :- cat(NP,np), NP has_feature_set [a(+),p(+)].

:- aBinderDom(X) smallest_configuration CF
        with_components
                X,
                A given_by aBinds(A,X,CF).

aBinds(A,B,S) :- binds(A,B,S), A has_feature apos.

aFreeCond(X,D) :-                   % only way R-expr can be A-bound is
        variable(X),                % outside the domain of its operator
        X has_feature operator(Op),
        D has_constituent D1,
        properlyDominates(D1,Op).% defined in figure 3.23.
```

Figure 3.27 Condition C.

(46) is understood to be a disjunctive condition as shown in (47) (= (3:57) in Chomsky [12]) below with (47b) only applying when (47a) is inapplicable:

(47) a. An r-expression must be A-free in the domain of its operator.
b. An r-expression must be A-free.

The first case applies to variables such as e_1 in (48):

(48) [*NP* [*NP* The man$_1$] [*CP* Op_1 I saw e_1]] decided to shoot John

Here, for purposes of strong Binding, Chomksy [12] (which we will not discuss here), e_1 must also be bound by *the man*. Hence, the domain of the operator (Op) must be CP. The second case applies to non-variables such as *John* in (49).

(49) *John$_1$ can't stand John$_1$'s teacher *(= (2:36a) in Lasnik & Uriagereka)*

Compare the definition in (46) with its realization in figure 3.27. Here, we have made use of the `smallest_configuration` form in the definition of `aBinderDom(X)` which finds the smallest constituent that dominates an A-binder of `X`. If such a domain can be found, then `aFreeCond(X,D)` must hold. Now, the only way that `aFreeCond(X,D)` can hold is if `X` is a variable such that its operator `Op` is properly dominated by some element `D1` within the domain `D` in which the variable happens to be A-bound.

```
binds(A,B,S) :-
        c_commands(A,B,S),
        coindexed(A,B),
        \+ A has_feature indexed(agr). % make sure not coindexed
                                       % for agreement.
% A c-commands B in S
c_commands(A,B,S) :-
        S has_constituents Cs,
        in(A,C,Cs),                    % A,C ∈ Cs, A ≠ C.
        C dominates B.
c_commands(A,B,S) :-
        S has_constituent C,
        c_commands(A,B,C).

coindexed(A,B) :-
        index(A,I),                    % defined previously in figure 3.17.
        index(B,J),
        sameIndex(I,J).                % primitive.

A dominates A.
A dominates B :-
        A has_constituent C,
        C dominates B.
```

Figure 3.28 Binding.

Now let us consider the computation of the (smallest) A-binder domain as defined by `aBinderDom`. Currently, because condition C is transformed into a compositional definition, each time a larger constituent is built that dominates an R-expression `X`, an attempt will be made to satisfy the conditions of `aBinderDom(X)`. That is, for each such constituent `CF` the parser will attempt to find (in a *bottom-up* fashion) an `A` such that `aBinds(A,X,CF)` holds. As soon as an A-binder is found, then the (smallest) domain is complete and `aBinderDom(X,D)` will succeed. Then, Condition C will succeed only if `aFreeCond(X,D)` holds. Now, consider the definition of `aBinds/3`. `aBinds/3` relies on the definition of `binds/3` shown in figure 3.28. In turn, `binds/3` is defined using `c_commands/3` which holds for all triples A, B and S such that A c-commands B for any constituents A and B within S. There are two clauses for `c_commands/3`:

1. The first clause implements the canonical case which was shown previously in (41)). Here, A (the c-commander) is an immediate sub-constituent of S.

```
:- aBinderDom(X) smallest_configuration CF
        with_components
                X,
                A given_by locallyABinds(A,X,CF).

locallyABinds(A,B,S) :- locallyBinds(A,B,S), A has_feature apos.

locallyBinds(A,B,S) :-
        locallyC_commands(A,B,S),
        coindexed(A,B),
        \+ A has_feature indexed(agr).
locallyC_commands(A,B,S) :-
        S has_constituents Cs,
        in(A,C,Cs),% A,C ∈ Cs, A ≠ C.
        C dominates B.
```

Figure 3.29 Localized version of `aBinderDom(X)`.

2. The second clause accounts for the case when A is not an immediate but a 'deeper' sub-constituent of S.

Hence, when `c_commands` is called in the course of computing an A-binder domain, first the immediate constituents of a candidate domain are tested, then if none of those immediate constituents turn out to be A-binders, the second clause is called to recursively pick out other possible candidates. However, this results in much redundant computation.[31] There is no reason to look any further (down) than the *local* sub-constituents for an A-binder for any particular domain because the bottom-up processing of the compositional definition ensures that previous (unsuccessful) attempts to locate an A-binder would have tried to use lower sub-constituents as candidate A-binders. In other words, the ability of the procedure to locate the smallest domain would not be affected if a minimal version of `c_commands` consisting of just the first clause was used instead. Hence, we can re-define `aBinderDom(X)` as shown in figure 3.29.

The current system automatically detects and produces these transformations. For example, an equivalent situation occurs with the definition of `c_commandsBT/3`. Returning to the governing category domain `gc(X)` (shown in figure 3.23), the same transformation is performed in the computation of an accessible SUBJECT. There, `c_commandsBT/3` has the same basic format as the definition of `c_commands/3` shown

[31] For binary branching, this will be proportional to the depth of the tree.

above.

Chapter 4

The Recovery of Phrase Structure

4.1 Motivation

The mechanism used to recover the phrase structure component of structural descriptions (SDs) is a crucial component of the parser. It must be efficient for the following reasons:

- First of all, linguistic principles that deal with syntactic constituents make broad use of structural relations such as 'subject-of', 'object-of', 'governs', etc. For simplicity, in the current implementation, parser operations require phrase structure descriptions to be recovered before structural relations can be evaluated. For example, the object of a verb *believe*, as in *believe Mary...*, will depend on the structure of the verb phrase (VP) built by phrase structure recovery, e.g. see (1a) and (1b):

  (1) a. [*VP* [*V* believe] [Mary]]
  (object is Mary)
  b. [*VP* [*V* believe] [Mary saw him]]
  (object is Mary saw him)

- Secondly, there may be many different phrase structure representations for any given input sentence. As is well known, there are two major sources of overgeneration: (1) *lexical ambiguity*, as in the surface form *saw* that may be syntactically realized as a noun or a verb; and (2) *structural ambiguity* in the sense that syntactic constituents may occupy many different positions in PS. For example, in English, a noun phrase that follows a verb may occupy various positions:

  (2) a. [I [*VP* believe [*NP* Mary]]]
  (object position)

b. [I [*VP* believe [[*NP* Mary] [*VP* saw him]]]]
(subject position of a complement clause)
c. [I [*VP* realized [*CP* [*NP* Mary] [*IP* [*NP*-e] [*VP* saw him]]]]]
(Comp — a position normally occupied by operators)
cf. [I [*VP* realized [*CP* [*NP* what time] [*IP* it was]]]]

The possibility of much overgeneration suggests that a substantial amount of the work spent in parsing will be spent on phrase structure recovery.[1]

- Thirdly, the cost effectiveness of some control strategies such as principle interleaving, and, to a lesser extent, principle ordering, is dependent on the *optimality* of the phrase structure recovery process in terms of error detection, in a sense to be made clear later.

For these reasons alone, it would seem worthwhile to expend significant effort to optimize phrase structure recovery. Moreover, as this chapter will describe, at least for the examples in Lasnik & Uriagereka, we can take advantage of well-understood and efficient parsing techniques developed for (subsets of) context-free languages (CFLs).

4.2 Roadmap

The remainder of the chapter is divided into four major sections. In the first section, we will discuss how phrase structure recovery is handled in a model of grammar that contains multiple levels of representation. Next, we will describe a phrase structure grammar for S-structure, the level of syntax that is recovered first; and outline an algorithm based on shift-reduce parsing techniques that will be sufficiently powerful to cope with the adopted grammar. Then, we will describe the implementation of the algorithm and how the rule schema specifying the grammar may be compiled into the tabular form required by the underlying machine. Finally, we will conclude by evaluating how well the shift-reduce machine matches the problem of recovering structure at the level of S-structure.

[1] As a rough guide, informal 'profiling' of an early version of the parser suggested that roughly 80% of the parsing effort went into phrase structure recovery for typical sentences from Lasnik & Uriagereka. This proportion dropped to the 20% level when grammar rules were compiled using the canonical LR(1)-based techniques described in this chapter; as opposed to on-line interpretation of the rules with no pre-compilation. Note that the 80% to 20% decrease translates into a twenty-five-fold improvement in phrase structure recovery times. Of course, the actual improvement will vary depending on the degree of ambiguity and the length of the input sentence.

$$\begin{tikzcd}[column sep=small] & \text{D-structure} \arrow[d] & \\ & \text{S-structure} \arrow[dl] \arrow[dr] & \\ \text{Phonetic Form} & & \text{Logical Form} \end{tikzcd}$$

Figure 4.1 The standard model of phrase structure.

4.3 Multiple Levels of Representation

In this section, we will begin by reviewing a standard model of phrase structure that consists of multiple levels of representation. Next, we will outline, and discuss the computational advantages of, a two stage method to recover these levels of representation.

4.3.1 A Standard Model of Phrase Structure

In general, the phrase structure component of grammar may contain several *levels* of representation related by transformation. For instance, figure 4.1 contains a typical generative model found in the principles-and-parameters framework. In this standard model, the level of D-structure is generated by simple syntactic rules that obey the properties of $\overline{\text{X}}$-theory (to be defined later). It is also the level at which grammatical relations such as 'subject-of' and 'object-of' are structurally defined. For example, consider the (partially specified) D-structure [[*NP the police*] [*VP arrested* [*NP John*]]]. Here, the NPs *the police* and *John* are understood to be the subject and object of the verb *arrest*, respectively. Semantic, or *thematic*, relations are also directly expressed at this level through these grammatical relations. For instance, we say that the subject *the police* bears the thematic role (hereafter abbreviated to θ-role) 'agent' of the predicate 'arrest'. Similarly, we say that the object *John* bears the 'patient', or 'theme', θ-role. Note that in some cases not all such positions need be filled. For example, [[*NP-e*] [*VP was arrested* [*NP John*]]] where the subject position is occupied by an empty NP is also a valid D-structure. Here, *John* still bears the patient θ-role of 'arrest'.

The level of S-structure is generated from D-structure by the operation of Move-α that allows the iterated, possibly null, *movement* of syntactic units. For example, NP-movement is an instance of Move-α where NPs, usually non-empty, are allowed to move to positions occupied by empty NPs. Ignoring morphological considerations, this is how the passive form *John was arrested*, where *John* is understood to bear the patient θ-role of 'arrest', may be generated from the underlying D-structure [[*NP-e*] [*VP was arrested* [*NP John*]]]. More precisely, NP-movement will substitute *John* for [*NP-e*] leaving behind an empty constituent [*NP-t*] in the position previously occupied by *John*, thus obtaining the S-structure [[*NP John*] [*VP was arrested* [*NP-t*]]]. In standard terminology, [*NP-t*] is known as a *trace* of movement.

From S-structure, phonological rules derive the level of phonetic form that forms the interface between language and the external systems that produce speech. Also, from S-structure, the movement of elements relevant to logical interpretation such as quantifiers and *wh*-phrases produces the level of Logical Form (LF) that forms the interface to systems that interpret language. For example, consider the sentence *Mary loves everyone*. Roughly speaking, the LF associated with this example should have the interpretation (3a), where x is a logical variable. Now, the S-structure will have the form (3b). The corresponding LF, (3c), is formed by 'raising' the quantifier *everyone* into a position where it may be interpreted as having sentential scope. In this case, Quantifier Raising (QR) leaves a trace [*NP-t*] that will be interpreted as the logical variable that *everyone* quantifies over.

(3) a. for all persons x, loves(Mary,x) (Interpretation)
b. [[*NP* Mary] [*VP* loves [*NP* everyone]]] (S-structure)
c. [*IP* [*NP* everyone] [*IP* [*NP* Mary] [*VP* loves [*NP*-t]]]] (LF)

4.3.2 Recovering Multiple Levels of Representation

The parser basically adopts the standard model of phrase structure as outlined above. The only significant deviation is that the level of phonetic form has been replaced by a level of Surface Form (SF). That is, instead of an abstract phonetic representation, the input for parsing is just a sequence of words.[2]

Now, let us consider the phrase structure model as a 'black box'. Of the four levels of representation, only the two interface levels, SF and LF, are *externally visible* — the other two, namely D- and S-structure, being purely theory-internal levels of representation. Hence, given some arbitrary SF, the parsing problem is to explicitly compute the corresponding LF structures that satisfy the constraints of LF, and indirectly, all other constraints that apply at the other levels. Note that this does not imply that all levels of phrase structure must be explicitly recovered. In the current implementation, we adopt the two stage approach shown in figure 4.2. In the first stage, we recover the phrase structure component of S-structure from an input SF. The other two levels, LF and D-structure, are not recovered simultaneously.

[2] Currently, morphological forms are mostly simply 'spelled-out' in the lexicon. One exception is for contractions. For example, a contraction like *okotteru* ('*be angry about*') in Japanese will be analyzed (on-line) as having the form *okotte* + *iru* (progressive '*be*').

$$
\begin{array}{c}
\text{D-structure} \\
\uparrow \\
\boxed{\text{Move-}\alpha} \\
\uparrow \\
\text{SF} \rightarrow \boxed{\text{Recover Phrase Structure}} \rightarrow \text{S-structure} \rightarrow \boxed{\text{LF-movement}} \rightarrow \text{LF}
\end{array}
$$

Figure 4.2 The implemented model of phrase structure recovery.

Instead, they are computed from initial S-structures at a later stage (as indicated by the dataflow in figure 4.2.) Partitioning the recovery of phrase structure in this manner has several computational advantages:

- *Recovering all levels simultaneously may involve doing unnecessary work.*

  For example, if an ill-formed structure may be eliminated by some constraint that applies at one particular level; then obviously, there is no need to recover any other level. Consider the (ill-formed) sentence **someone sleep*. The standard explanation for this is that the subject *someone* and the form of the verb *sleep* do not agree — cf. *someone sleeps*. In the theory adopted by the parser, the mechanism by which subject-verb agreement is enforced involves assigning an index feature to both the subject of a clause and an inflection element (hereafter abbreviated to Infl or I) that contains an agreement component (AGR) consisting of person, number and gender features. AGR is, in turn, associated with the verb through head movement. (The details of head movement (to be described in section 4.4.1 are not important for the purposes of this discussion.) Hence, as soon as a candidate S-structure has been assigned, e.g. $[_{IP}\ [_{NP}\ someone\ ]\ [_{\bar{I}}\ [_{I}\text{-}t\ ]\ [_{VP}\ [_{V}\ [_{I}\ ]\ [_{V}\ sleep\ ]]]]]$ — more precisely, once *someone* has been identified as the S-structure subject — no operation other than subject–Infl coindexation need be applied to rule out the ill-formed structure. In particular, there is no need to construct any part of a LF or D-structure representation.

- *Limiting overgeneration.*

  The decision to recover phrase structure at the level of S-structure first rather than at the level of D-structure or LF was largely motivated by the goal to limit overgeneration. Experience with an early version of the system that initially recovered phrase structure at D-structure suggested that overgeneration was a major bottleneck.[3] The basic problem in this case is that there is no simple

[3] Incidentally, one motivating factor for the D-structure generator was that the grammar would

correspondence between the order in which terminal elements of D-structure appeared and their relative positions in the SF. This is primarily caused by the power of Move-α to freely 'scramble' syntactic elements between D- and S-structure. Moreover, given the lack of a simple correspondence, modifying the recovery routine to allow for the effect of Move-α would require the addition of much detailed knowledge about how the Move-α operation works. Of course, this would severely compromise the goal of maintaining a strict separation between parser operations. Hence, the early version of the phrase structure recovery routine adopted the null hypothesis: that is, the free permutation, or scrambling, of elements between D- and S-structure.[4] Consequently, the routine hypothesized many structures that could not have generated the SF in question. For example, *John saw Mary* cannot be generated from a D-structure [[*NP Mary*] [*VP saw* [*NP John*]]]. Moreover, in order to exclude such impossible forms, Move-α must be applied to generate corresponding candidate S-structure forms. In turn, these candidates must be 'linearized' to check that the order of elements matches that of the original SF. By contrast, in the current theory, there is no powerful scrambling rule between SF and S-structure. In fact, apart from examples of genitive Case realization such as *John's book* and *pictures of themselves* where markers *'s* and *of* are not present at S-structure but inserted at SF, the 'yield' of the SF and S-structure consist of the same sequence of terminal elements. Of course, empty syntactic units such as [*NP-e*] in *John was arrested* [*NP-e*] may be inserted at the level of S-structure, but the linear precedence relations encoded in the SF will remain unaltered at S-structure. Hence, we can make of the invariance in linear precedence to avoid generating many of the aforementioned 'impossible' structures.[5] Oversimplifying somewhat, given a SF of length n, the old D-structure recovery routine would have to attempt to recover structures for $n!$ permutations of the SF, whereas the corresponding S-structure routine only has to deal with the original order.

---

be simpler, i.e. contain fewer terminal and non-terminal symbols, than the corresponding grammar at the level of S-structure. (Section 4.4.1 enumerates the extra cases that a grammar for S-structure has to handle.) Another potential advantage is that a smaller grammar may have a smaller characteristic finite state automata. That is, table size may be less of a problem when it comes to constructing shift-reduce machines.

[4] Although the other alternative of directly recovering LFs has not been explored, the presence of LF-movement parallels that of Move-α for D-structure.

[5] Note that there is no fundamental linguistic basis for this similarity between the level of SF and S-structure. In some theories linear precedence is irrelevant at S-structure. See Marantz (in Baltin *et al.* [8]) or Kashet [32] for an example of a purely 'hierarchical' version of S-structure.

Function: D-structure Constituent Recovery.
Input: A S-structure constituent X_S.
Output: The D-structure constituent at the position occupied by X_S.

1. If X_S occurs in a chain $(X_1, X_2, \ldots, X_n)$[6] then:
   (a) return X_1 if X_n is X_S, otherwise
   (b) return an empty category.[7]
2. Otherwise return X_S. (X_S has not been moved.)

Figure 4.3 D-structure constituent recovery algorithm.

- *Selective recovery of theory-internal levels of representation.*

  Complete LFs must be recovered since that is the designated output of the parser. However, there is no obligation to do the same for phrase structure at the level of D-structure since, as discussed earlier, such structures will not be externally visible. In fact, we can avoid redundant computation by only recovering fragments of structure at D-structure on a demand basis. Since D- and S-structure are related by Move-α, any portion of the phrase structure at D-structure may be deduced from S-structure once the derivation sequence, or *history of movement*, has been computed. In general, given any S-structure constituent we can deduce the element that occupies the S-structure position at D-structure using the algorithm shown in figure 4.3. In the current implementation, this algorithm is explicitly invoked to compute individual D-structure constituents as required. Complete D-structures are never recovered. For example, given a partially instantiated S-structure, such as (4a), together with the facts that: (1) *who* originated from the position occupied by [*NP*-*e*] (in standard terminology, *who* 'heads' the (movement) *chain* (*who*, [*NP*-*e*])), and (2) there are no other chains in the S-structure (ignoring the movement of heads such as verbs for the moment); then we can infer that the corresponding phrase structure at D-structure must be of the form (4b).

  (4) a. [*CP* [*NP* who] [[*C* did] [*IP* [*NP* John] [*VP* [*V* like] [*NP*-e]]]]]
  b. [*CP* [*NP*-e] [[*C* did] [*IP* [*NP* John] [*VP* [*V* like] [*NP* who]]]]]

[6] Notation: Let $(X_1, X_2, \ldots, X_n)$ denote a non-trivial movement chain of length $n, n > 1$ with head X_1. The head X_1 originates at the position occupied by the last element of the chain X_n known as a *base trace*. Elements $X_2, \ldots, X_{n-1}$ are *intermediate traces*. Each adjacent pair (X_i, X_{i+1}), or *link*, in the chain denotes a single instance of movement originating at the position occupied by X_{i+1} and terminating at the position occupied by X_i.

[7] Note that this algorithm will also return an empty category if X_S is an adjunct in an adjunction structure formed by movement.

PS Rules at S-structure =

PS Rules at D-structure:

- $\overline{\text{X}}$-rules: e.g. $\overline{\overline{\mathbf{X}}} \rightarrow \mathit{Spec}, \overline{\mathbf{X}}$
  $\mathbf{X} = N, \mathit{Spec} = \mathit{Det}$
  [*NP* [*Det the*] [$\bar{N}$ *destruction of the city*]]

- Empty category rules: e.g. $NP \rightarrow \lambda$
  [*IP* [*NP-e*] [*VP was followed John*]]

- Adjunction rules: e.g. $VP \rightarrow VP, Adv$
  [*VP* [*VP was followed John*] [*Adv why*]]

\+

Rules to account for movement:

- e.g. $CP \rightarrow Adv, \overline{C}$; $\quad Adv \rightarrow \lambda$
  [*CP* [*Adv why*] [$\bar{C}$ *was John followed*]]

Figure 4.4 Components of the phrase structure grammar for S-structure.

4.4 Recovery of Phrase Structure at S-structure

In the previous section, we have described a scheme for recovering phrase structure at the various levels of representation. Here, we focus on the first stage: the recovery of phrase structure at S-structure from input SFs. In this section, we will deal with two main issues: (1) obtaining a grammar for PS at S-structure, and (2) selecting a practical and efficient algorithm for building structures that conform to this grammar.

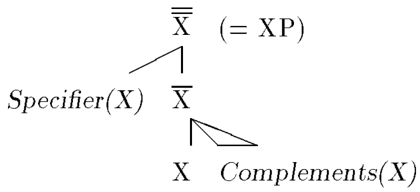
4.4.1 A Grammar of Phrase Structure at S-structure

In the generative model shown in figure 4.4, phrase structure at the level of S-structure is derived from corresponding phrase structure at D-structure through Move-α. How can we obtain a grammar for S-structure? Phrase structure at the level of D-structure is generated by rules that obey the principles of $\overline{\text{X}}$-theory together with additional rules to cover cases not handled by $\overline{\text{X}}$-theory, such as the introduction of empty categories and adjunction structures. We will first describe a grammar that covers the range of phrase structure at D-structure. Then we will consider the effect of Move-α on this grammar and revise the grammar accordingly to cover the range of phrase structure at S-structure.[8]

[8] 'Covering grammars' have also been employed for traditional transformational grammars. See Petrick [43], for example.

Phrase Structure Rules at D-structure

Phrase structure rules, with only a few exceptions, are defined according to the rules of $\overline{\mathrm{X}}$-theory. In $\overline{\mathrm{X}}$-theory, all phrases have the same basic internal structure. A typical *two-bar-level* prototype for a phrase might look something like the following:



Here, $\overline{\overline{\mathrm{X}}}$ represents the *maximal projection* that corresponds with the traditional notion of a complete phrase. For example, $\overline{\overline{\mathrm{N}}}$ and $\overline{\overline{\mathrm{V}}}$ denote a noun phrase (NP) and verb phrase (VP), respectively. The two other levels $\overline{\mathrm{X}}$ and X are known as the *intermediate projection* and *minimal projection*, or *head*, of the phrase. For example, the minimum projection of a noun or verb phrase will contain the noun or verb itself. The intermediate projection dominates both the head and the *complements* of the phrase. Typically, objects of a noun or verb will occupy the complement positions. Finally, the specifier position of a phrase may be filled by various elements such as subjects or determiners in the case of a noun, e.g. [*NP John's* [$\bar{N}$ [*N mother*]]] or [*NP the* [$\bar{N}$ [*N man*]]].

The relative order of the specifier and complements with respect to the head is parameterized according to different languages. For example, as we have described earlier, in Japanese heads will generally follow complements whereas the opposite is true for English. The $\overline{\mathrm{X}}$-prototype can also be extended to apply to sentences and full clauses:

1. In such theories, the maximal projection $\overline{\overline{\mathrm{I}}}$ (I for inflection), corresponds to the traditional category S (for sentence). The specifier position for I will be filled by the subject of the sentence, and its complement position by the main verb phrase. For example, *John saw Mary* will have the structure [*IP John* [$\bar{I}$ [*I*] [*VP saw Mary*]]]. Finally, it is assumed that the head of the sentence is occupied by inflection (I).

2. The prototype can be extended one level further to full clauses which are assumed to be headed by complementizers (sometimes abbreviated to Comp or C), e.g. *that* as in *that John saw Mary.* In this case, the complement position will be occupied by a sentence (IP), and the specifier position may be occupied by the usual clause-initial elements such as question operators. For example, *who did John see* might have the structure [*CP who* [$\bar{C}$ [*C did*] [*IP John see*]]].

```
head(n). head(v). head(a). head(p). head(i). head(c).
bar(n1). bar(v1). bar(a1). bar(p1). bar(i1). bar(c1).
max(np). max(vp). max(ap). max(pp). max(i2). max(c2).

proj(X,Y) :- head(X), bar(Y).
proj(X,Y) :- bar(X), max(Y).

spec(n1,np). spec(n1,det). spec(n1,[]).
spec(i1,np).
spec(c1,np). spec(c1,adv). spec(c1,[]).

noSpec(X1) :- \+ spec(X1,_).
noSpec(X1) :- spec(X1,[]).

compl(i,vp). compl(c,i2).
```

Figure 4.5 Predicates on category labels.[9]

Note that there are many variations of $\overline{\text{X}}$-theory; for example, the specifier position of C may not be available in some versions, more bar levels might be used for certain categories such as verbs in others, and so on.

We will now turn to the problem of encoding the $\overline{\text{X}}$-prototype. We will tackle this in two stages. First, we will define the categories that the prototype ranges over, together with fixed (as opposed to lexically determined) choices of specifiers and complements for appropriate categories. Then, we will define the grammar rules that represent the internal structure of the prototype.

We begin by defining a few simple predicates on category labels. Consider the definitions shown in figure 4.5. Here, the three one-place predicates `head/1`, `bar/1`, and `max/1` define a two bar-level scheme for nouns, verbs, adjectives, prepositions, inflection, and complementizers. The possible specifiers in each case are defined by the clauses of `spec/2`. Here, for example, we have allowed nouns to take a noun

[9] For convenience, we will repeat the description of PROLOG notation introduced in chapter 1:

1. Logical variables are distinguished from constants by the use of an uppercase letter as the initial character. The scope of a variable is a clause. Also, '_' is used to indicate an anonymous or 'don't care' variable. The scope of '_' is itself.

2. Elements enclosed in square brackets denote lists. In particular, `[]` denotes the empty list. `[Y]` denotes a list with one element `Y`, `[Y1,Y2]` a list of two elements, and so on. `[X|L]` denotes a list with first element `X` and with `L` representing the remainder of the list.

3. Outside of lists, (`,`) is used to denote conjunction. (`:-`) — to be read as 'if' — is used to denote implication and (`\+`) is used to denote negation as failure to prove.

```
rule XP -> [X1|specifiers(X1)] ordered specFinal st max(XP), proj(X1,XP).
rule specifiers(X1) -> [Y] st spec(X1,Y), \+ Y=[].
rule specifiers(X1) -> [] st noSpec(X1).

rule X1 -> [X|complements(X)] ordered headInitial st bar(X1), proj(X,X1).
rule complements(X) -> [] st lexicalProperty(X,grid(_,[])).
rule complements(X) -> [Y] st compl(X,Y).
rule complements(X) -> [Y] st lexicalProperty(X,subcat(Y$_,_),Y).
rule complements(X) -> [Y] st lexicalProperty(X,grid(_,[R]),csr(R,Y),Y).
rule complements(X) -> [Y1,Y2]
        st lexicalProperty(X,grid(_,[R1,R2]),csr(R1,R2,Y1,Y2),[Y1,Y2]).
```

Figure 4.6 $\overline{\mathrm{X}}$-prototype rules.[10]

phrase (`np`), or a determiner (`det`) as a specifier, or have no specifier at all (denoted by `[]`). We have also specified that no specifier should be assumed as the default if a specifier is not mentioned in `spec/2`. This is accomplished by the first clause of `noSpec/2` which is intended to be read as: "`X1` may take no specifier if there is no specifier clause for `X1`." Finally, `compl/2` states that I and C take a verb phrase and sentence as a complement. (The possible complements of lexical categories such as verb and nouns will be specified through lexical entries as we shall see below.)

We are now in a position to define the $\overline{\mathrm{X}}$-prototype. Consider the grammar rules shown in figure 4.6. The first group of rules defines the top half of the prototype. The first rule is intended to be read as: "`XP` derives `X1` followed by the specifiers of `X1` if `specFinal` holds," such that `XP` is a maximal projection and `X1` (immediately) projects to `XP`. (The `proj/2` relation was defined previously in figure 4.5.) Note that `specFinal` is a language parameter that will be defined to hold only for those languages for which specifiers come after heads and complements. If `specFinal` does not hold, then the order of the constituents will be reversed.[11] The next two rules state what the possible specifiers are:

[10]Note that some non-terminals such `complements(X)` and `specifiers(X)` are *pseudo*-non-terminals in the sense that they do not represent linguistic categories. For example, no constituents with a category label such as `Complements(c)` will be constructed. Although such non-terminals appear in the rule schema, they are eliminated when the rule schema is converted into the underlying LR-style tables used for parsing. (See section 4.5.3 for the description of this process.) More precisely, when the schema is grounded, all occurrences of pseudo-non-terminal symbols in the RHS of any rule will be replaced by the RHS of a rule that contains the relevant pseudo-non-terminal as its LHS.

[11]Actually, this can be replaced with just about any predicate. For example, we can write `specFinal(XP)` instead of just `specFinal` (and make the corresponding adjustments in the language parameters) to make the order sensitive to particular categories.

1. The first rule states that the specifier of `X1` will be `Y` provided `spec(X1,Y)` holds and Y is not `[]` (used to represent the null specifier).

2. The second rule states that `X1` may take no specifier (`[]`) if `noSpec(X1)` holds, i.e. when there is no specifier listed for `X1` or the null specifier itself is explicitly listed.

The second group in figure 4.6 defines the lower half of the prototype. As with `specFinal`, `headInitial` is a language parameter which will be defined to hold for languages like English, but not for Japanese (where heads generally follows complements). There are five choices for complements:

1. `[]`, i.e. no complement, provided `lexicalProperty(X,grid(_,[]))` holds. In general, `lexicalProperty(X,F)` is defined to hold only if there exists some element of category `X` with lexical property `F`. Now, we assume that nouns, verbs, adjectives and prepositions generally have θ-grids, that is a specification of the thematic roles associated with each particular element. For example, as we have described before, the verb *file* is associated with some 'filer' (represented by an 'agent' θ-role) and an object to be filed (represented by a 'theme' or 'patient' θ-role). We distinguish between *internal* and *external* roles which are realized as complements and specifiers, respectively. In the present system, this is indicated by having a verb like *file* have the lexical feature `grid([agent],[theme])`, the internal roles occupying the second argument of `grid`. Returning to the goal `lexicalProperty(X,grid(_,[]))`; this holds if there exists a lexical element of category `X` that takes no complements.

2. The second possibility is `[Y] st compl(X,Y)`, i.e. a single complement `Y` for category `X`. `compl/2` was defined previously in figure 4.5 for categories I and C.

3. The next possibility is `[Y]` if there is some element of category `X` that has lexical feature `subcat(Y$_,_)`. We will go into the details here except to mention that not all lexical subcategorization for complements will be done through θ-grids. Sometimes, it is more convenient to allow some elements, e.g. a verb, to have an object that it does not assign a θ-role to.

4. The fourth possibility is `[Y]` if there is some element of category `X` that selects one internal role, to be realized as an element of category `Y`. Here, we have adopted Chomsky's [12] Canonical Structural Realization (CSR) relation, represented by `csr/2`, that maps θ-roles to syntactic categories. For example, the role 'agent' will be realized as a `NP`.

5. The last case states that there may be two complements `[Y1,Y2]` that are also realized through the θ-grid.[12] For example, *persuade* selects a 'goal' and a 'proposition' as in *persuaded* [*NP John*] [*CP to leave*] *right away.* Note that the CSR relation may map a role into different syntactic categories. For example, a 'proposition' can be realized as an clause (`CP`) or a noun phrase (`NP`), as in *persuaded John of* [*NP the importance of leaving*].

(Note that the above rules will only accommodate lexical items with two or fewer objects (as defined through the θ-grid). If necessary, more rules can be added to handle three or more internal θ-roles.)

We will now turn to examples of other rules not included in the $\overline{\text{X}}$-prototype that are needed at D-structure:

- *Empty categories.*

```
rule np with Features -> [] st ecNPFeatures(Features).% empty NP
```

[I want [[*NP*-e] to win]] [*NP*-e]= *PRO (at S-structure)*
[[*NP*-e]was arrested [*NP* John]] *Passive.*
[you file which report without [[*NP*-e] reading [*NP*-e]]] *Parasitic gap.*

```
rule c with Features -> [] st nullFeatures(Features). % empty Comp
rule i with Features -> [] st nullFeatures(Features). % empty Infl
```

The first rule allow the introduction of empty NPs. Such NPs may occur as empty subjects (embedded or otherwise) by the Extended Projection Principle (EPP) and as objects as required by thematic properties of individual predicates as shown above. Empty categories are also required at D-structure for the two non-lexical categories C and I. CP may be headed by an overt complementizer, as in ... *happy* [*CP* [*C that*] [*John left*]] or *happy* [*CP* [*C for*] [*John to leave*]], or be headed by an empty complementizer [*C*], as in [*CP* [*C*] [*IP John left*]]. In the current theory, at D-structure inflection is always empty, consisting of two components, AGR and TNS, implemented as syntactic features. For empty categories at D-structure, unlike their overt counterparts which have lexical entries (and thus 'pick up' their syntactic features from those entries), they must have features instantiated separately from the lexicon. In the empty category rules shown above, this is accomplished by predicates `ecNPFeatures/1` and `nullFeatures/1`.

- *Adjunction.*

[12] We include this last case to show that the representation can also handle theories that use non-binary branching. Actually, the present implementation does not make use of ternary branching. Instead, double objects are represented using binary branching with an extra bar level. The definitions shown here come from an earlier version of the theory used by the system.

```
adjunction rule vp -> [vp,adv].
adjunction rule vp -> [vp,pp].
adjunction rule np -> [np,c2].
```

(Non-movement) adjunction is required for relative clauses. We assume that the modifying clause will be right-adjoined (in English) to the 'head' noun phrase, following van Riemsdijk & Williams [46]. For example, *the man who I saw* will be assigned the following D-structure: [*NP* [*NP the man*] [*CP who* [*IP I saw*]]]. Similarly, adjunction of PPs to VPs is required to handle examples such as *John* [*VP* [*VP was arrested*] [*PP after leading the demonstration*]] and *John* [*VP* [*VP was arrested* [*PP by the police*]]]. Moreover, the object of *reading* in *which report did you file without reading* must occur in a VP adjunct for the empty NP to be properly licensed as a parasitic gap. (See pg.72ff in Lasnik & Uriagereka for the details.) Finally, adverbs such *often* and *completely*, are assumed to be adjoined to VPs at D-structure; for example, *John* [*VP often* [*VP sees ghosts*]]. Following Lasnik & Uriagereka, *wh*-adverbs such as *why* are also initially adjoined to a VP at D-structure. However, they will be subsequently fronted to a clause-initial position either at LF or S-structure. For example, the D-structure representation for *why did John leave* is assumed to have the form [*John* [*VP* [*Adv why*] [*VP leave*]]].

Phrase Structure Rules at S-structure

In the previous section, we have described a grammar for phrase structure at D-structure. By considering the effect of Move-α on this grammar, we will obtain the corresponding grammar for S-structure. Unfortunately, because this process has not been automated in the current system, the burden of specifying the additional phrase structures rules will fall on the grammar writer.[13]

The general process of Move-α allows any constituent to move anywhere in a given structure; the idea being that all instances of improper movement will be filtered out by some general principle. However, as Lasnik & Uriagereka note (pg.113), this is largely a promissory note that has not been made to work in full generality. To avoid generating structures that will not be relevant to any of the principles that have been implemented, the scope of Move-α has been restricted to cases of head movement, adjunct fronting and general movement of NPs; these being the 'core' cases handled by the principles of Case and θ- theory, and the

[13] However, the consequences of Move-α that are described in this section could be automatically derived from a description of the categories that move, the possible landing sites and whether substitution or adjunction (or both) are allowed in each case — although, given the somewhat non-regular nature of the non-$\overline{\mathrm{X}}$ phrase structure rules described in this section, it is not clear that such a system would offer much in the way of abstraction over the current approach.

Empty Category Principle (ECP). One consequence of this restriction is that the S-structure grammar must be updated if other instances of Move-α such as movement of intermediate projections or VPs are to be covered. With this caveat, we now enumerate those core cases:

- *Movement of noun phrases.*

  Movement of NPs generally fall into four categories distinguished by characteristics of the *landing* and *launching* sites. Such sites may be classified either as being A-positions, i.e. positions to which thematic roles may be assigned, such as object and subject positions; or as $\overline{\text{A}}$-positions such as the specifier position of a complementizer. For example, passive or raising examples such as *John seems to be happy* are one instance of A-to-A-position movement. Simple *wh*-movement examples such as *what did you see* fall under the category of A-to-$\overline{\text{A}}$-movement. Movement of *wh*-adverbs as in the example *why did John leave*, as described in the previous section, is an instance of $\overline{\text{A}}$-to-$\overline{\text{A}}$-movement. It seems difficult to construct a grammatical example of $\overline{\text{A}}$-to-A-movement, the only remaining case, in the current theory.[14] In the current implementation, traces of movement are just instances of the generic empty NP ' [*NP*-e]'. If we assume that can be no adjunction to Comp at S-structure, e.g. as in Lasnik & Saito [34], then the cases we have described can be covered by the existing D-structure rules because all relevant A-positions (subject and object positions) and specifier of Comp already permit empty NPs.

- *Scrambling.*

  `VP -> NP VP` % *adjunction to VP*
  `IP -> NP IP` % *adjunction to IP*

  Although there is seems to be no need for adjunction in Comp at S-structure (as Lasnik & Saito indicate), adjunction of NPs at VP and IP is commonly assumed (see Hoji [29] for example) for cases of scrambling in languages such as Japanese. We repeat here the four examples augmented with traces for the Japanese equivalent of *John gave Mary a book* (shown previously as (8) in chapter 1):

[14] Of course, it would be wrong to rule out instances of $\overline{\text{A}}$-to-A-movement on this basis. The phrase structure recovery routine should allow examples of structures with $\overline{\text{A}}$-to-A-movement and not preempt the constraints imposed by independent principles. For example, Lasnik & Uriagereka (pgs. 98–100) discuss examples of 'Super-raising' such as **John seems that it is likely to leave* where *John* originates from the subject position of predicate *leave*. In a model of phrase structure that allows Comp to have two landing sites available, namely, specifier of Comp and Comp itself; then such a model can generate an S-structure with *John* moving from the specifier position of Comp (an $\overline{\text{A}}$-position) to the matrix subject position (an A-position).

(5) a. [*IP* John-ga [*VP* Mary-ni hon-o ageta]]
    b. [*IP* hon$_i$-o [*IP* John-ga [*VP* Mary-ni t_i ageta]]]
    c. [*IP* Mary$_i$-ni [*IP* John-ga [*VP* t_i hon-o ageta]]]
    d. [*IP* John-ga [*VP* hon$_i$-o [*VP* Mary-ni t_i ageta]]]

- *Head movement.*

```
adjunction rule i(IV) -> [v(V)] st raiseVtoI(V,_,IV).
adjunction rule v(VI) -> [v(V)] st lowerItoV(V,_,VI).

rule v -> trace.
rule i -> trace.

adjunction rule c(CIV) -> [v(V)] st raiseVtoC(V,_,CIV).
```

In the current implementation, a simple model of verbal inflection has been adopted. Depending on certain properties of the agreement component of inflection (Infl) and the verb in question, either the verb will raise to adjoin to Infl, or Infl will lower to the verb. (The details of the mechanism — encoded by predicates `raiseVto/3` and `lowerItoV/3` — are unimportant in the current context.) The distinction between the two cases may be seen in English from the order of VP-adverbs (assumed in the previous section to be left adjoined to VPs at D-structure) with auxiliary and 'proper' verbs, as in *John completely lost his mind* (Infl lowers to V) and *John has completely lost his mind* (V raises to Infl). These two cases are covered by the first four rules shown above. The last rule is provided to cover cases of 'subject-verb-inversion', as in (6a), and *Do*-support, as in (6b):

(6) a. [*CP* [*C* [*V* is]$_i$ [*C*]] [*IP* John [*VP* [*V*-t]$_i$ here]]]
    b. [*CP* what [[*C* [*C*] [*I* [*I*]$_j$ [*V* do]$_i$]] [*IP* you [*I*-t]$_j$ [*VP* [*V*-t]$_i$ [*VP* see]]]]]

- *Movement of Adverbs.*

```
rule adv -> trace
```

Finally, rules must also be added to handle cases in English of adjunct movement at S-structure such as the fronting of wh-adverbs such as *why* as in *why did John move.* The above rule allows the introduction of empty adverbs only as traces of movement. In the current model, these cases are accomodated by allowing adverbs to occupy the specifier position of Comp at S-structure (the head position being reserved for zero-level projections).

Note that the actual phrase structure grammar to be constructed for S-structure will be language-dependent. For example, the scrambling rules are not used in English, and the rules allowing movement of *wh*-noun phrases and *wh*-adverbs will not be present in Japanese (since all such movement can only take place at LF in such languages). To provide some idea of the number of rules involved, the following table summarizes the situation for the phrase structure grammars used for the two languages:

Language	$\overline{\mathrm{X}}$-prototype rules	Empty category rules	Adjunction rules	Others
English	12	5	8	6
Japanese	(same)	4	6	2

By any measure, the number of rules (around thirty or so) can be considered to be small and, therefore, quite manageable.

In the next section, we will turn our attention to the efficient recovery of phrase structure representations that conform to the rules defined above.

4.4.2 Canonical LR(1)-Based Phrase Structure Recovery

The rule schema given in the previous section expands into a context-free grammar (CFG).[15] Therefore, to recover structures at the level of S-structure, we can take advantage of any one of the rich variety of different parsing mechanisms that have been developed to parse languages generated by CFGs or specific subsets of the context-free languages. Among these are the general tabular methods such as the Cocke-Kasami-Younger (CKY), Earley's, and the Graham-Harrison-Ruzzo algorithms, or the restricted table-driven shift-reduce methods such as the LR(k) algorithms (to be described below). There are also many non-tabular methods, for example, another class of algorithms are based on the formal equivalence between CFGs and push-down automata. Yet another possibility is to simply interpret CF rules in a top-down (or recursive descent) fashion. Tomita [56] has claimed that the LR-style shift-reduce machines are more efficient than any other existing parsing algorithm *in practice*.[16] This is the method that we have adopted in the current

[15] Obviously, since each schema rule is of the form $A \rightarrow B_1, \ldots, B_n$ (n bounded), and this form is preserved under simple substitution of $\overline{\mathrm{X}}$-categories for variables.

[16] Tomita principally compares the efficiency of Earley's algorithm to his extended-LR algorithm. Conceptually, Earley's algorithm and the shift-reduce algorithms, e.g. the LR algorithm extended to handle ambiguity, are very similar. Essentially, both algorithms manipulate configurations (sets of dotted productions) in similar ways. Nevertheless, Tomita shows that, in practice, LR-style parsing is generally more efficient than Earley's algorithm despite inferior worst-case complexity results. The reason for this is that LR-style parsing can be viewed as a *compiled* counterpart to Earley's algorithm. The essential difference is that all possible configurations are pre-computed when the LR machine is built off-line, whereas Earley's algorithm has the additional burden of computing and storing configurations during parse time.

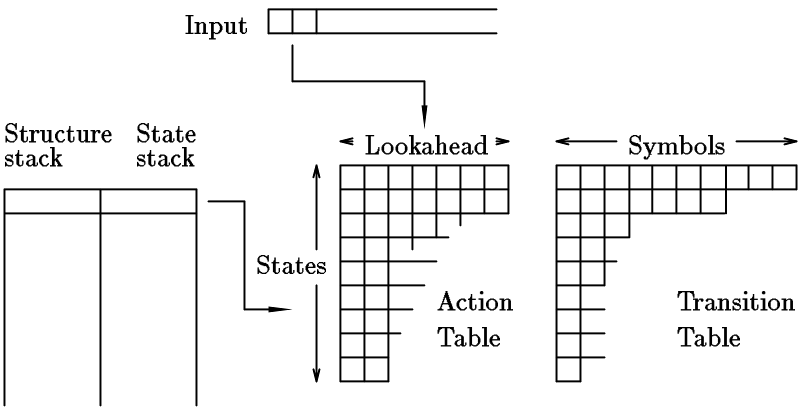


Figure 4.7 The components of a shift-reduce parser.

implementation.

Roadmap

This section is divided into two parts. In the first part, we will review the components of shift-reduce parsers in general, and the properties of LR-style parsing in particular. Then, we will discuss the problems of adapting LR-style parsing to the phrase structure grammar of S-structure described in the previous section.

Shift-Reduce Parsing

As is standard in the literature, a shift-reduce parser is a bottom-up parser that consists of a control and structure stack (sometimes combined); an input sequence, a prefix of which may form a limited lookahead buffer; and a table with an action and a transition component, as shown in figure 4.7. The structure stack holds terminal and non-terminal symbols that have already been recognized. The lookahead buffer contains a fixed number of input symbols (possibly zero depending on the parser) that have not yet been parsed. Based on the contents of the lookhead and the current state held on the top of the control stack, the action component of the table tells the parser what action to take next. There are four types of actions: (1) *Shift*: to recognize the first terminal symbol of the input by popping that symbol off the input onto the structure stack; or (2) *Reduce*: to recognize a non-terminal symbol A from a rule $A \rightarrow A_1 \ldots A_n$ by replacing the top n constituents of the structure stack (corresponding to $A_1 \ldots A_n$) with a (new) single non-terminal of category A and the n elements popped off the structure stack as immediate constituents. Additionally, the parser can: (3) *Accept*: signal that the sentential symbol has been recognized;

that is, a successful parse has been found; or (4) signal that an impossible, or 'error', state has been reached; that is, the sentential symbol cannot yield the given input. After performing the appropriate action, the transition component of the table tells the parser what state to go to next on the basis of the current state and the non-terminal symbol (after a reduce action) or terminal symbol (after a shift action) recognized. (For examples of shift-reduce parsing, the reader is referred to an introductory compiler text such as Aho & Ullman [7].)

The concept of LR(k) parsing, introduced by Knuth [33], is a class of *deterministic* shift-reduce parsers parameterized by k, the number of symbols of terminal lookahead. LR(k) parsers have many interesting properties; we briefly summarize a few of the most relevant ones below:

1. LR(k) parsers are of interest because they are the most powerful class of deterministic bottom-up parsers using up to k symbols of lookahead. This means that if a grammar G cam be parsed (deterministically) by *any* bottom-up parser using k symbols of lookahead; then an LR(k) parser can be constructed for G. However, not all CF grammar are LR(k) grammars.[17] The determinism requirement means that a LR(k) parser must be able to uniquely determine which action to take in any state given any lookahead. Equivalently, there may be no *conflicting* actions; instead, each action entry must contain a unique action.

2. Whatever the state of the parser, the contents of the structure stack will constitute a prefix of a *right sentential form.* (A *sentential form* for a grammar G with distinguished symbol S is any string β such that $S \Rightarrow^{\star} \beta$; that is, β is derived from S in zero or more steps, each step being the expansion of a non-terminal by some production in the grammar. If the rightmost non-terminal is always expanded, then β is a *right sentential form.*) Now, all LR-style parsers will admit only valid prefixes. An erroneous prefix, i.e. a prefix that cannot be part of a sentential form, will be detected as soon an attempt is made to shift an erroneous input symbol.

3. However, the size of LR(k) tables has proved to be a practical limitation on the size of grammars that can be handled and the number of symbols of lookahead that can be used. In fact, k is usually limited to one or zero. Even with this restriction, canonical LR(1) parsers are rarely used in the domain of (non-natural) programming languages. Instead, for practical LR-style parsing, parsers based on (sub-) classes such as the SLR(1) (*simple*-LR) and LALR(1) (*lookahead*-LR) parsers that trade power for reduced table size

[17] For example, no ambiguous CF grammar can be an LR(k) grammar for any k due to the deterministic action restriction.

are most commonly used.[18] However, one disadvantage with such a tradeoff is that these less-powerful parsers may do more work than strictly necessary. More precisely, in the case of parsing invalid prefixes, they may perform more reduce actions, i.e. build more constituents, than strictly necessary. The canonical LR(1) parser is *optimal* in the sense that it provides the earliest possible error detection, thereby reducing the amount of unnecessary work, given one symbol of lookahead.[19] (See the discussion of LR-parsing in Fischer & LeBlanc [21] for an explanation.)

In the interest of bringing the most powerful machinery to bear on the phrase structure recovery problem, the canonical LR(1) parser was selected as the target machine for the recovery of structure at the level of S-structure. Table size was a secondary consideration. However, as section 4.5.3 will describe, table size turned out not be a limiting factor for the grammar introduced in the previous section. In the next section, we will discuss what changes have to be made to the LR(1) algorithm to accommodate the grammar for phrase structure at S-structure.

Modifying the LR(1) Parser

The current implementation adopts the basic model of LR(1) shift-reduce parsing described in the previous section. However, some modifications to the basic model are necessary in order to handle the problems posed by the presence of empty categories in the S-structure grammar:

1. Although the S-structure rule schema expands into a CFG, the CFG is not an LR(1) grammar. The main reason for this is the presence of ambiguity (both lexical and structural), as discussed at the beginning of this chapter. At S-structure, the presence of movement traces will further exacerbate this problem:

   - *Shift-reduce conflicts.*

     For example, consider the choice that a LR(1) parser faces in a state where a verb may be expected and a verb occupies the first element of the input sequence.[20] In operational terms, the possibility of head movement (described in the previous section) creates a situation where the parser can either shift the verb at the head of the input, or reduce

[18] For example, YACC, perhaps the best-known LR-style parser-generator, is an LALR(1)-based parser-generator.

[19] Note that the only difference between these parsers is in the number of extraneous reductions before an error is detected. In particular, as stated above, all LR-style parsers never 'over-shift'.

[20] That is, a state where the sentential prefix may be augmented with a verb without creating an invalid prefix. More precisely, there will exist a dotted rule with a verb in its lookahead set in the configuration set corresponding to the current state.

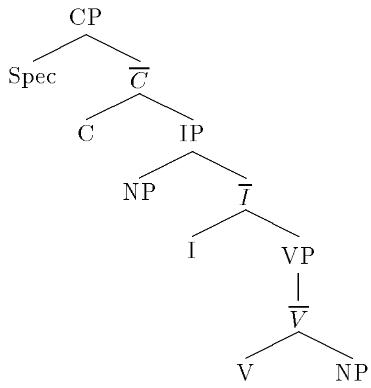


Figure 4.8 Basic clausal structure.

to create a verb trace (the expected verb may have moved elsewhere, as in *John* [*I* [*V has*] [*I*]] [*VP*• [*V-t*] [*V lost*] *his mind*], in which (•) marks the point reached by the parser and *lost* is the verb that heads the input). In standard terminology, this situation is known as a *shift-reduce conflict.*

- *Reduce-reduce conflicts.*

  Movement can also produce *reduce-reduce conflicts.* For example, suppose the parser has just built a verb phrase, say [*VP* [*V* [*I*] *lost*] [*NP his mind*]], as in *John lost his mind.* Moreover, for simplicity, assume that all input has been shifted. In this case, the parser can go on to complete a $\bar{I}$, i.e. [$\bar{I}$ [*I-t*] [*VP* [*V* [*I*] *lost*] [*NP his mind*]]], by performing a reduce action using a production of the form $\bar{I} \rightarrow I, VP$. But despite the lack of further input, this is not always the correct decision. For instance, it might be necessary to add further material to the VP, e.g. an adverbial trace, resulting in [*VP* [*VP* [*V* [*I*] *lost*] [*NP his mind*]] [*Adv-t*]] as in *why has John lost his mind.* In this case, the correct decision will be to reduce using a production of the form $Adv \rightarrow \lambda$.

Hence, strict determinism must be relaxed if the LR(1) algorithm is to handle such conflicts. This is handled in the current implementation, by permitting each action table entry to hold an arbitrary number of actions. As a result, the parser can handle arbitrary CFGs.

2. Parsing termination is a classic problem posed by the presence of empty categories at S-structure. Consider the typical skeletal form of S-structure as

shown in figure 4.8. Now, each 'leaf' of this tree may derive the empty string. As discussed previously, the rule schema contains a production of the form $NP \rightarrow \lambda$ to account for NP-traces and *PRO*. From the head movement rules, both V and I may be realized as traces. Furthermore, C may be occupied by an empty complementizer. Finally, in the current theory, the specifier position of C need not be filled at all. Therefore, we can deduce that CP $\Rightarrow^{\star} \lambda$, i.e. full clauses may be empty. Hence, the parser will be able to build full clauses without shifting a single symbol. Morever, clauses may contain other clauses because some heads (such as the verb *believe*) may select a proposition that may be syntactically realized as a CP. The recursion may be also be slightly less direct. For example, a noun phrase may contain a relative clause which will also be realized as a CP. Taken together, the ability to recursively predict a CP without needing to consume any input will cause the parser not to terminate.

There are many ways to block infinite recursion through CP. For example, one can impose a depth bound on the structure stack. The solution that was adopted allows an arbitrary number of CPs to be predicted, but blocks infinite recursion by requiring each CP to consume some input. Basically, the parser uses a simple condition on traces to ensure that each trace is 'licensed' in the sense that a trace cannot exist without an original overt constituent of the same category being present somewhere in phrase structure. (Note that these conditions are not imposed for empty noun phrases. As described earlier, different conditions involving the θ-grid and the requirement that subjects be present are used for noun phrases.) To ensure termination, it will suffice to apply the restriction to cases of head movement only. From the previous section, we have that verbs can adjoin to Infl, as in (7a), or vice-versa, as in (7b):

(7) a. John [[*I* [*I*] [*V* has]$_i$] [*VP* completely [*V*-t]$_i$ lost his mind]]
    b. John [[*I*-t]$_i$ [*VP* completely [*V* [*V* lost] [*I*]$_i$] his mind]]

Raising the resulting Infl-verb complex to Comp is a further optional step to handle the cases of subject-verb-inversion and *do*-support discussed previously. Now consider the head movement rules from pg. 137 (repeated below):

```
adjunction rule i(IV) -> [v(V)] st raiseVtoI(V,_,IV).
adjunction rule v(VI) -> [v(V)] st lowerItoV(V,_,VI).

rule v -> trace.
rule i -> trace.

adjunction rule c(CIV) -> [v(V)] st raiseVtoC(V,_,CIV).
```

If a verb is adjoined to Infl; then, obviously, there must be a corresponding verb trace present, assuming that no traces may be deleted at S-structure. Similarly, there must be an Infl trace if Infl is adjoined to the verb. Finally, if the verb is adjoined to Comp; there must be both a Infl and a verb trace present elsewhere in phrase structure. That is, each of the three adjunction rules will license Infl or verb traces (or both). Then, let a rule of the form `X -> trace` have the special interpretation: "Generate an empty category `X` only if $[X\text{-}t\,]$ is licensed." (Of course, this licensing restriction can only work if this is the only way an empty verb can be generated; that is, there are no other productions of the form `V ->[]`.) Consequently, all CPs must now contain an overt verbal element. Infinite recursion through CP is now impossible since CP $\not\Rightarrow^{\star} \lambda$.

To enforce this restriction, the LR(1) machine model is slightly modified as follows:

(a) Production rules may communicate by placing constituents on a new environment stack that functions as a 'scratchpad'. For example, a rule such as `i(IV) -> [v(V)] st raiseVtoI(V,_IV)` that licenses a verb trace will, in addition to building an Infl constituent (with a verb adjoined), have the side-effect of placing a verb (complete with lexical features) on the environment stack.

(b) A production that introduces traces such as `v -> trace` will build a verb trace (and also pick up its lexical features) only if a verb has been previously placed on the environment stack.

(c) So far, we have assumed that all traces will have been predicted ahead of time (in a left-to-right manner). Of course, when an occurrence of a trace precedes the relevant head-adjunction structure, as in the case of Infl lowering to V in English, then the production that introduces the traces, `i -> trace` in this case, must look ahead (possibly many terminal symbols ahead) to the input buffer for licensing. (For head-final languages like Japanese, this must happen for the case when a verb raises to Infl.) The current system contains an automatic mechanism that computes the relative order of the elements Infl, V and Comp (for a given language) and tells the trace licensing mechanism to look either to the environment stack, or to the input buffer, or even (non-deterministically) to either side for an appropriate licensing head-adjunction structure.

This mechanism is also used to eliminate another source of infinite recursion. For example, consider the sentence *how did John lose his mind*. Here, *why* is adjoined to VP at D-structure, using a production of the form $VP \rightarrow VP, Adv$, resulting in the structure $[VP\ [VP\ lose\ his\ mind\,]\ [Adv\ how\,]]$. At

S-structure, *how* will move to Comp leaving behind a trace. As described in section 4.4.1, the trace is generated by a rule of the form `adv -> trace`. Hence, by using the same licensing restriction as for head movement, we can prevent the generation of an infinite class of structures of the form [*VP* [*VP*... [*VP* [*VP lost his mind*] [*Adv-t*]]... [*Adv-t*]] [*Adv-t*]] containing an arbitrary number of adjunctions.[21]

Having described the basic components of the modified LR(1) parser, in the next section, we will turn our attention to the efficient implementation of the parser, including further modifications of the LR(1) model to improve its performance, and the automatic compilation of the S-structure rule schema down to the level of the underlying machine.

4.5 Implementation and Compilation

So far we have discussed the rationale for adopting a multi-stage approach to recovering the levels of phrase structure from the point of view of computational efficiency; and, in particular, the adoption of a modified LR(1) algorithm for recovering the first level at S-structure. In this section, we will briefy describe the implementation, noting, where relevant, those features of the underlying model that affect how the parser builds phrase structure. We will also note further modifications of the scheme described so far that have been incorporated to further improve its efficiency. Finally, we will briefly discuss the method by which the S-structure rule schema may be turned into the tabular form required by the underlying LR(1)-based model.

4.5.1 Basic Implementation

The modified LR(1) machine introduced in the previous section contains the usual LR(1) action and transition tables, albeit slightly modified to allow entries to hold multiple actions; plus the three internal stacks that are used to manipulate machine states, to hold partially constructed constituents, and to facilitate the communication of contextual information. As with the regular linguistic principles discussed in chapter 3, the machine is represented as PROLOG predicates to be interpreted by the standard top-down, left-to-right, depth-first search mechanism used by PROLOG. In the following discussion, we will briefly mention how each of these components are

[21] Currently, the system is set up to allow only one iteration of `adv -> trace` to apply in any one 'context', or single state of the CFSM (Characteristic Finite State Machine) associated with the LR(1) machine. In other words, the infinite recursion will be terminated after the first iteration. This parameter can be varied for individual categories by the grammar writer on a state-by-state or global basis.

realized; we will also point out those salient features that will prove to be useful when the interleaved application of principles is considered in chapter 6.

- *The representation of the transition table.*

  In the current implementation, the transition and action tables are represented separately. Only the action table need depart from the standard LR(1) model to accommodate non-determinism. The transition table δ that tells the parser what state S' to go to next following a reduction or shift operation, based on the current state S and the non-terminal or terminal symbol V, i.e. $\delta : S \times V \mapsto S'$, remains deterministic. δ is represented by a predicate `transition` of arity three such that each clause `transition(`S`,`V`,`S'`)` holds iff $\delta(S, V) = S'$.

- *The representation of the action table.*

  All actions will alter the machine configuration, i.e. the control stack (CS), the structure stack (SS) and the environment stack (ES) plus the input sequence (I). Hence, actions may be regarded as instances of a prototype $f : \mathrm{CS} \times \mathrm{SS} \times \mathrm{ES} \times \mathrm{I} \mapsto \mathrm{CS}' \times \mathrm{SS}' \times \mathrm{ES}' \times \mathrm{I}'$. (The control and structure stacks have been separated so they need not operate in 'lockstep' for reasons that will become clear later.) Hence, we can represent f as a predicate `action` of arity ten such that each clause `action(`S`,`V`,CS,SS,ES,I,CS',SS',ES',I')` represents a single LR action in state S given lookahead symbol V.[22] As an example, suppose both shift and reduce (using the production $\bar{A} \rightarrow A, NP$) are possible actions in a state S_1 with lookahead N. Then, the action table will contain the following two clauses:

  1. Shift:

     `action(`S_1`,n,CS,SS,ES,[I1|I],[`S_2`|CS],[I1|SS],ES,I).`[23]

     where `I1` represents the noun to be shifted from the input to the structure stack (SS), and $\delta(S_1, N) = S_2$ holds.

  2. Reduce:

     `action(`S_1`,n,[`S_1`,`S_2`,`S_3`|CS],[NP,A|SS],ES,I,`
     `[`S_4`,`S_3`|CS],   [A1|SS],  ES,I).`

     where `A1` represents a constituent of category $\overline{\mathrm{A}}$ with immediate constituents `A` and `NP`, and $\delta(S_3, \bar{A}) = S_4$ holds.

[22] `action` is of arity ten rather than eight for efficiency reasons. Strictly speaking, the first two arguments are redundant since the current state is the same as the top of the control stack, and the single lookahead symbol is the category of the head of the input. The extra two arguments were added to take advantage of clause indexing under multiple arguments, which facilitates almost constant time access to individual clauses in the version of PROLOG used.

[23] We have represented all stacks and the input sequence as PROLOG lists.

The representation of the LR tables has the following interesting features:

- *No explicit error entries.*

  Note that there are no restrictions on the number of actions per table entry. Each possible action is represented as a separate clause. Furthermore, only shift, reduce and accept actions are explicitly represented by clauses. The action table is sparse in the sense that error entries are deliberately omitted. If there are no matching clauses (actions) for a given state and lookahead; then the parser will assume an error entry by default. Both conflicts and implicit error entries are automatically handled by the backtracking search mechanism used by PROLOG. If there is more than one matching clause for a given state and lookahead, as is the case when conflicts exist, then the interpreter will pick the first clause it encounters and leave a choice point so that it can try the other possibilities when it backtracks later. If there are no matching clauses, as is the case with an implicit error entry, then the interpreter will immediately backtrack to a previous choice point (if one exists) to search for other solutions. By forcing the interpreter to explore all choice points, the S-structure parser can find all possible parses.

- *Pre-computing the transition function.*

  For both reduce and shift actions, updating the control stack involves computing the transition function δ. In some cases, it is possible to eliminate the access to the transition table by computing the next state off-line. For example, consider the clause shown above for the shift operation. The current state and the category of the symbol to be shifted is already known when the action clause is constructed. Since these are the input arguments to δ, and δ is a (deterministic) function, this information suffices to determine the next state. Of course, the next state cannot always be computed ahead of time. For example, consider the clause shown above for the reduction of $\bar{A} \rightarrow A, NP$. Here, the top of the control stack, i.e. the current state S_1, and the lookahead symbol `n` are both known (off-line). Note that δ operates on the third element from the top of the control stack, S_3.[24] However, the value of the S_3 is not known at compile time. Hence the transition table must be consulted at parse time. In the current implementation, a partial evaluation mechanism is used to pre-compute the next stage whenever possible.[25]

---

[24] There are two non-terminals on the RHS of the rule for $\bar{A}$. The completion of each non-terminal will result in the addition of a single state to the control stack. Hence the reduce action must first pop off these two elements to restore the state that corresponds to the completion of $\bar{A}$.

[25] Whilst this cannot result in a large improvement in the overall efficiency of the LR(1) machine — after all, it is an instance of 'simple unfolding' (in the sense of section 2.6) — it is a fairly safe procedure in the sense that it is unlikely that there will be many situations in which the parser

- *Adding extra pre-conditions.*

  Since actions are represented as PROLOG clauses, table entries need not be limited to unit clauses only. In general, additional constraints can be placed on actions by introducing goals on the RHS of unit clauses. For example, this is how the trace licensing mechanism described in the previous section is implemented. Consider the following reduce action generated for a production `v -> trace`:

  ```
  action(116,n,CS,SS,ES,I,[55|CS],[[V-t]|SS],ES',I) :-
  predicted(v,ES,ES').
  ```

  This clause states that in state 116 with a noun as the lookahead symbol, the parser should build a new constituent, the verb trace [*V-t*], on the structure stack and proceed to state 55, *provided* a verb has been predicted on the environment stack.

  The option of adding extra conditions to action clauses is also exploited for the principle interleaving mechanism to be described in chapter 6. Basically, principle interleaving works by applying the constraints specified by linguistic principles to portions of phrase structure as they are built up in a bottom-up fashion using the S-structure parser described here. The essential idea will be to insert PROLOG goals (that call various constraints) in those reduce action clauses that build constituents on which those constraints apply. For example, if some constraint realized as a predicate P operates only on VPs and NPs; then for all clauses that reduce VPs or NPs, i.e. every time a VP or NP constituent is built, we should insert a goal $P(tos(\texttt{SS}'))$, i.e. apply P to the new constituent on the top of the structure stack.

4.5.2 Further Modifications

So far the major modifications to the basic LR(1) algorithm have been motivated by the need to handle specific properties of the S-structure grammar. We now turn to other (non-essential) modifications that have been tried purely for efficiency reasons, as opposed for coverage. One of these, namely, the elimination of the need to consult the transition table in some cases, has already been discussed. We now briefly mention modifications that have been designed to reduce the effect of allowing conflicting actions; that is, to reduce the degree of non-determinism present in the action table:

---

will do more work than before. To provide a rough idea of the applicability of the procedure: calls to the transition table were resolved off-line in 61% and 40% of the 895 and 765 LR(1) action table entries for English and Japanese, respectively.

- *Unbounded lookahead.*

  The basic motivation for the trace licensing mechanism was to prevent the parser from hypothesizing empty clauses which would lead to non-termination problems. By controlling the generation of certain empty categories, in particular, verb traces; all attempts to generate empty clauses were bound to fail sooner or later. But, from examining the behaviour of the algorithm, it was found that a considerable amount of unnecessary work, i.e. in building partial clauses, was performed by the parser. For example, relative clauses are analyzed as clausal adjuncts introduced by a production of the form `np -> [np,c2]`. Therefore, each time a noun phrase was constructed, the parser would attempt to build a relative clause even if there was no following input. Sometimes, it would (incorrectly) consume overt input elements intended for the matrix clause. For example, consider the sentence *John visited Mary*. Once the NP corresponding to *John* had been built, the parser would attempt to build a (relative) clause in the following ways:[26]

  1. Using no further input, as in [*NP* [*NP John*] [*CP*]] *visited Mary*.
  2. Using *visited* as part of the relative clause, as in [*NP* [*NP John*] [*CP (who) visited* [*NP-e*]]] *Mary* ....
  3. Using *visited Mary* as part of the relative clause, as in [*NP* [*NP John*] [*CP (who) visited Mary*]]....

  All three attempts will eventually fail (the first one because the relative clause is empty, and the other two because the matrix clause will have no overt verbal element). However, the parser will have performed many shifts and reductions, all of which are wasted because none of the three possibilities will lead to a complete parse. Given an arbitrarily long lookahead, a parser can, of course, detect before a single shift or reduce is performed (unnecessarily) that generating a relative clause for *John* is futile. Unfortunately, the parser described so far has only one symbol of lookahead to go on. Furthermore, uniformly extending the LR table to handle two or more symbols of lookahead would be prohibitively expensive. (Roughly speaking, each additional symbol would increase the number of table entries by a factor of $|V_T|$, the number of terminal symbols.) Hence, a solution was adopted that kept the LR table size almost constant, but allowed actions to inspect the input.

  Basically, we would like to be able to say something of the form:

[26] Note that we cannot get around this problem simply by adding a requirement of the form: "All relative clauses must have a relative pronoun". This would be an incorrect generalization. Ellipsis of the relative pronoun is permitted in many circumstances. For example, both *the man who I saw* and *the man I saw* are well-formed relative clauses.

In schema $\overline{\overline{\mathbf{X}}} \rightarrow \{$ *Specifiers*($\mathbf{X}$), $\overline{\mathbf{X}}$ $\}$,
if $\mathbf{X}$=C, then first check the input for an available verbal element.

The essential idea will be to impose this condition by adding a PROLOG goal to the relevant action clauses. Note that it will be possible for the extra goal to access the input sequence since it is one of the ten arguments to `action`. In the expanded grammar, the above stipulation is equivalent to generating a production of the form:

$CP \rightarrow$ `checkInput(I)`, *Spec*, $\overline{C}$

where `checkInput` is not a non-terminal, but a pre-condition of the production. `checkInput(I)` will be designed to succeed only if a verbal element in `I` (the input) is available. If no such element is available; then the production may not apply. Unfortunately, as it stands, the above production cannot be directly used by the LR machine. Attaching the condition to either a shift or reduce clause does not provide the desired behaviour because we want to check the input *before* predicting a CP, i.e. before a single terminal for the CP is shifted. Attaching the condition to the reduce CP clause would result in the condition being applied only *after* the entire RHS has been recognized; that is, only after $\overline{\text{C}}$ has been reduced. The solution is to simply introduce a (new) dummy non-terminal at the point where the `checkInput` is to be invoked. Next, we augment the grammar by adding a dummy production that causes the LR machine to neither build structure nor read any input; but simply to have the side-effect of applying the desired condition. Hence, we generate the following two replacment productions:

$CP \rightarrow$ *Dummy*, *Spec*, $\overline{C}$
Dummy $\rightarrow \lambda$ if `checkInput(I)`

The latter production will result in a reduce action of the form:

```
action(S1,V,CS,SS,ES,I,[S2|CS],SS,ES,I) :-
        checkInput(I).
```

such that $\delta(S_1, \mathit{Dummy}) = S_2$ holds. [27] By modifying the LR machine to use this unbounded lookahead mechanism, three to six-fold improvements in PS recovery times were observed.

- *Subcategorization.*

  Dummy non-terminals and extra pre-conditions also allow phrase structure construction to be 'driven' by selectional properties of heads. As discussed before, in general, lexical heads select semantic roles such as patient, agent and proposition. Normally, these roles may be syntactically realized as NPs or CPs. Hence, the above schema may be expanded into 'ground' rules by consulting the lexical entries for each head category. For example, consider the following (incomplete) list of production rules that expand $\overline{\mathrm{V}}$:

$\overline{\mathrm{V}} \rightarrow \mathrm{V}$	e.g. *John* [$_{\bar{V}}$ *slept*]
$\overline{\mathrm{V}} \rightarrow \mathrm{V}$, NP	e.g. *John* [$_{\bar{V}}$ *saw* [$_{NP}$ *Mary*]]
$\overline{\mathrm{V}} \rightarrow \mathrm{V}$, CP	e.g. *John* [$_{\bar{V}}$ *believes* [$_{CP}$ *Mary left*]]
$\overline{\mathrm{V}} \rightarrow \mathrm{V}$, NP, CP	e.g. *John* [$_{\bar{V}}$ *asked* [$_{NP}$ *Mary*] [$_{CP}$ *to leave*]]

  Of course, it would be grossly inefficient to allow the LR machine to operate directly on these rules without subcategorization restrictions. For example, consider the sentence *John slept.* Assuming the LR table to contain the list of ground rules shown above, once *slept* has been shifted, the machine would go through each possibility, inventing empty constituents, e.g. [$_{NP}$-*e*], if necessary in an attempt to complete each RHS. Roughly speaking, by immediately consulting the lexical features of the verb as soon as it has been shifted (onto the structure stack), we can eliminate the unnecessary search to complete the last three productions. For instance, we might impose conditions of the following form:

  $\overline{\mathrm{V}} \rightarrow$ V, `takesNoObjects(`*tos*`(SS))`
  $\overline{\mathrm{V}} \rightarrow$ V, `takesNPObject(`*tos*`(SS))`, NP
  $\overline{\mathrm{V}} \rightarrow$ V, `takesCPObject(`*tos*`(SS))`, CP
  $\overline{\mathrm{V}} \rightarrow$ V, `takesNPCPObjects(`*tos*`(SS))`, NP, CP

[27] This is a standard trick used in compiler construction. Note that the dummy non-terminal must be made invisible to the reduce clause for the CP production. Normally, the reduce action for a production with RHS of length three would involve popping the top three elements of both the control and structure stacks. The elements popped off the control stack are discarded and those popped off the structure stack will become immediate constituents of the new phrase. However, the dummy production builds no structure. Hence, we must modify the default behaviour of the reduce action to pop only the top two elements off the structure stack. (Note however, we must still pop three elements off the control stack.) The need to accommodate such dummy non-terminals is one reason for separating the control and structure stacks so that they need not operate in lockstep.

> Here, we run into the same problem with semantic actions as in the unbounded lookahead case. The solution will also be the same: just invent new dummy non-terminals and empty productions containing just the semantic action for each case.

In the actual implementation, the separation between such productions is taken one step further. In the current system, the parser computes equivalence classes for each set of conflicts. Each class represents the dummy productions that can be (ambiguously) satisfied by some lexical element. For example, currently the verb *buy* is analyzed as selecting a 'theme', i.e. something to be bought, and (optionally) a 'location', i.e. the place of purchase, as complements. Hence, the two productions `vNP -> v(V) if takesNPObject(V)` and `vNPNP -> v(V) if takesNPNPObjects(V)` will overlap, and thus belong to the same class. However, if there is no verb in the lexicon that can either take no complements and a single NP as a complement (or none and a double NP), then `vZero -> v(V) if takesNoObject(V)` will not belong in the same class as the previous two dummy productions. Hence, if the current verb in the input can be reduced using the production for `vZero`, there will be no need to test if it takes one or two NPs as objects. The current system exploits this wherever possible to effectively reduce the branching factor in states with multiple reduce-reduce conflicts of this nature. (The effect on the conflict distribution of action table entries will be analyzed in section 4.6.)

Note that the introduction of such dummy non-terminals may cause only a linear increase in the number of productions of the expanded grammar. Hence, such modifications should not unduly impact grammar size. More importantly, the modifications that we have discussed should be transparent to the grammar writer. In the current implementation, the grammar writer merely specifies phrase structure at the abstract level of the rule schema; the details of the underlying machinery such as the LR stacks and action clauses being invisible (and inaccessible). The generation of the LR table and various efficiency modifications that we have described are all handled automatically when the rule schema is compiled. In the following section, we will briefly describe this compilation process.

4.5.3 Compilation

In order to maintain transparency, it is necessary to automate the process of translating the rule schemas that encode phrase structure representations down to the level of the target LR machine. This is carried out in two stages as shown in figure 4.9.

In the first stage, schema rules that contain variables ranging over $\overline{\mathrm{X}}$ categories will be expanded, or 'grounded', by instantiating over each category. Furthermore, the variety of complement structure for each lexical head category will be determined by consulting the lexicon. The expanded rules for $\overline{\mathrm{V}}$ shown in the previous

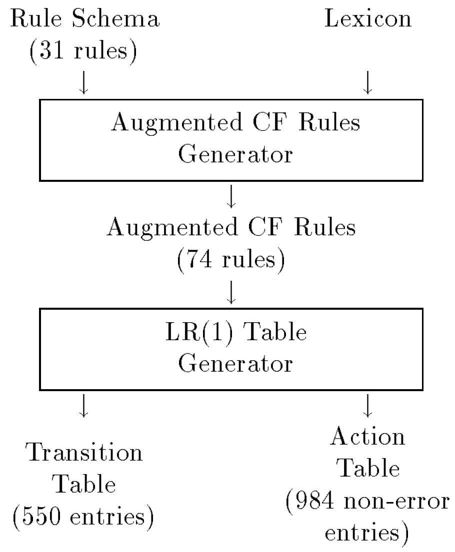


Figure 4.9 The two stage LR-table generator

section are an example of this process. The various optimizations that have been described are also carried out at this stage. The resulting output will therefore consist of ground $\overline{\mathrm{X}}$ rules augmented with extra conditions. This intermediate representation is then fed to the 'back-end' stage. In the current implementation, this stage contains a fairly standard procedure that constructs the PROLOG clauses that encode LR(1) tables. (For details of the method, see Algorithm 6.2 in Aho & Ullman [7].) This modular two stage approach facilitates the substitution of a different back-end, say, a more sophisticated LR(1) table generator, e.g. Pager [42], that generates a more compact machine; or, indeed, any of the other CFG parsing techniques mentioned earlier.

To provide a rough measure of the machine size for the phrase structure grammar of S-structure, the augmented CFG consists of about 74 productions derived from a schema of 31 rules. The resulting characteristic finite state automaton (CFSM) consists of 123 states with 550 transitions between the various states. The action table consists of a total of 984 individual (non-error) entries, counting conflicting actions as separate entries. This compares very favourably with typical (artificial) programming languages. For example, Aho & Ullman state that a typical programming language has about 100 productions resulting in a LR(1) CFSM of several thousand states. Even with a less powerful shift-reduce method such as the LALR(1) algorithm, the CFSM may still contain several hundred states. Also, the corresponding action table may easily have 20000 entries. Direct comparisons with natural language grammars are obviously difficult for reasons of coverage. However, Tomita [56] mentions two natural language CFGs that contain about 220 and 400 productions. He reports that these grammars have LR(0) CFSMs of the order of 400 and 600 states, respectively. Although, the corresponding numbers for a canonical LR(1) CFSM are not available, it is likely that such machines would contain many thousands of states. Bearing in mind the caveat just mentioned, these numbers suggests that canonical LR(1) parsing is not an available option for systems that encode most of the grammatical knowledge using CF rules only. The grammars cited by Tomita would seem to fall in this category. LR(1) parsing is still feasible for theories in the principles-and-parameters framework such as the one described in this thesis because the search for explanatory adequacy has led to the elimination of much of the work traditionally borne by the phrase structure component. (See section 3.3.2. in Chomsky [12] for discussion on this point.) Outside the principles-and-parameters framework, Graham *et al.* [27] discusses the use of a CF grammar for English of about 160 productions that was supplied by Pratt who conducted an experiment to "see how simple the grammar could be made at the expense of complicating other portions of his system."

Although canonical LR(1) parsing is certainly feasible, we have not yet addressed whether its major feature, the (full) one-symbol lookahead mechanism, actually contributes to making the parser more deterministic or not. In the next section, we will address the suitability of the LR(1)-based approach for the S-structure grammar.

4.6 Results and Conclusions

To summarize, we have described a multi-stage approach to recovering phrase structure. This process is driven off the first stage; that of recovering structure at the level of S-structure. By using a method based on the LR(1) algorithm, we have described how such structures may be efficiently recovered. From the level of S-structure, both LFs, the designated level of output of the system, and D-structure may be recovered. Constituents of the latter level, being a theory-internal level of phrase structure, are only computed on a demand basis. In this section, we will evaluate how this approach measures up to the design goals outlined in chapter 1.

- *Abstraction and transparency.*

  Although the current implementation makes considerable use of compilation techniques to automate the process towards producing an efficient phrase structure recovery procedure; still, a significant portion of the gap between the definitions found in linguistic theory and a working parser has to be bridged by the grammar writer. For example, the input to the compilation process is a phrase structure grammar for S-structure. This grammar is derived manually, as opposed to being automatically generated, from a definition of $\overline{\mathrm{X}}$-theory (which applies at D-structure) plus a specification of how much of Move-α to use. Similarly, the procedure that recovers D-structure constituents from S-structure is not automatically generated. Much more work must be done, in particular, involving the addition of automated theorem proving techniques to deduce the effects of Move-α on phrase structure at D-structure, in order to close the abstraction gap.[28]

The various extensions to the basic LR(1) model such as multiple actions, the addition of the environment stack, and arbitrary conditions on LR actions drastically increase the power of the resulting machine. In fact, the last extension allows the modified machine to have access to general computing power. However, note that each extension has been motivated either by the need to handle problematic features of the grammar, e.g. traces and empty categories in general, or by the goal of computational efficiency. Note also that the modifications for purely efficiency reasons, such as the unbounded lookahead mechanism or equivalence class computation, do not affect the language generated by the grammar, and, hence, are transparent to the grammar writer. Moreover, the additional capabilities over the CF model are essentially hidden; that is, there is no direct access to the features of the target machine. The compilation process here serves the dual purpose of insulating the grammar writer from the power of the underlying implementation. Of course, the insulation will not be complete unless the grammar represented by the rule schema can be debugged at a higher level of abstraction than LR actions. Currently, such 'source-level' debugging facilities remain to be added to the system.[29]

A potentially more serious problem is that correctness could be compromised by the limitation placed on recursion by the trace licensing mechanism. For example, as discussed earlier, repeated adjunction of adverbs to a VP is disallowed to ensure termination. Hence, if there are well-formed structures that involve cases of repeated adjunction of adverbs, then the current system will fail to recover them. Although the bound on the number of such adjunctions can be adjusted by the grammar writer, and that no such structures have been found to be necessary for the sentences that the parser has been tested on in

[28] Recently, there has been some work in this direction (Stabler [52]).

[29] Note that the process of generating an LR table can also provide useful tools for debugging phrase structure rules. For example, one by-product of computing lookahead symbols is the set of all non-terminals that derive the empty string. This information can be provided to the grammar writer as a tool for detecting parsing termination problems.

Lasnik & Uriagereka, a more sophisticated control mechanism to detect cases of vacuous adjunction would be a more satisfactory solution. That is, ideally the system should be able to detect and suppress all cases where further adjunction would not affect well-formedness (or ill-formedness) of the structure being built.[30]

- *Computational Efficiency.*

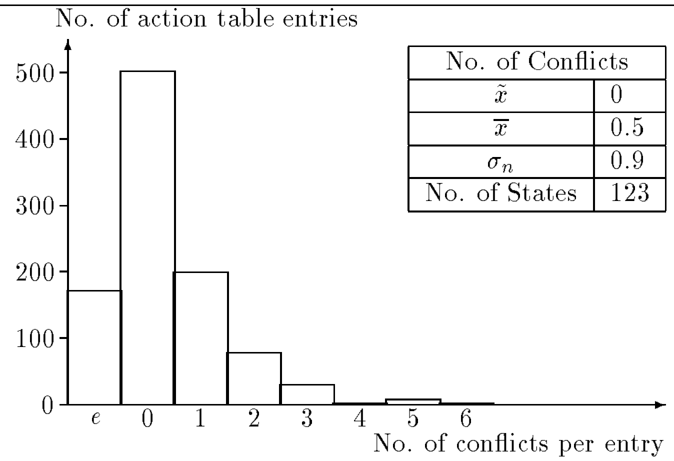
  As mentioned at the beginning of the chapter, one of the motivating factors for optimizing phrase structure recovery was that, in general, there will be many possible structures, within the space of possibilities specified by a rule schema, for a given sentence. By simply taking advantage of one of the extant efficient algorithms for parsing with CF rules, the ratio of the amount of work spent on phrase structure recovery as opposed to application of the principles described in chapter 3 changed drastically. (See note 1 for specific details.) Previously, grammar rules had simply been interpreted in an straightforward, on-line fashion. Although the original LR(1) model has been 'tweaked' considerably, there are still areas that could be improved. Perhaps the most significant among these is that there is no special mechanism for structure sharing between multiple parses — cf. Tomita's packed forest representation. For example, the VP built for [*CP* [*NP John*] [*IP* [*NP-e*] [*VP saw Mary*]]] will be discarded when the parser backtracks past the VP (in order to re-position *John*) to build the second parse [*CP* [*IP* [*NP John*] [*VP saw Mary*]]]. Preserving shared constituents in the backtracking framework would require the implementation of some kind of memoization scheme.[31]

We now conclude the chapter by considering how well the LR(1)-based algorithm 'matches' the phrase structure grammar for S-structure.

[30] See Stabler [52] (section 13.6) for discussion on this point.

[31] In the backtracking framework, the amount of structure sharing between parses depends on how far the parser has to backtrack to find the next valid choice point. When the parser backtracks to an earlier choice point, it also (automatically) restores the state of all the stacks, effectively discarding any structure built on the structure stack since that choice point. If some common structure had been built before the choice point, then it is still shared; otherwise, it must be rebuilt from scratch. To escape this restriction, the lifetimes of structures to be saved must be extended by storing those structures in an external database. Depending on the PROLOG implementation, such a memoization technique can be very expensive. In informal experiments, memoization turned out not to be a cost-effective optimization. The problem was that, for performance reasons, structures are usually allocated room on a special stack. Such structures must be copied into a less transient area of memory if they are to be preserved under backtracking. Unfortunately, this copying process turned out to be expensive enough to more than compensate for the shortcut of not having to re-build any shared structure. (Note that the costs involved may very well be different given some other implementation of PROLOG.)

Of course, the obvious alternative is to abandon the backtracking approach and construct all possible parses in parallel. This is essentially what happens in Tomita's extended LR parser.



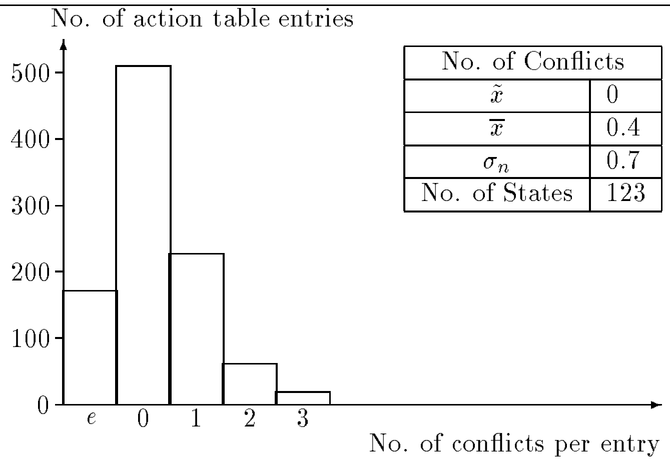
No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.5
σ_n	0.9
No. of States	123

Figure 4.10 English S-structure grammar conflict statistics.

(Here, 'e' represents an entry with no possible actions, i.e. the error entry.)

Tomita has suggested that LR-based algorithms can be much faster than Earley's algorithm in practice because 'natural language grammars' are considerably closer to LR than other general context-free grammars. The essential claim is that the extended-LR parser is able to handle non-LR grammars with little loss of LR efficiency, provided that the grammar is 'close' to LR. In the following discussion, we address the question: How close to LR(1) is the phrase structure grammar for S-structure? We also consider a related question: Does (one symbol) lookahead help? That is, does lookahead reduce the degree of non-determinism represented by multiple actions? What about the effect of the equivalence class shortcut procedure?

We can get an idea of the 'LR(1)-ness' of a grammar by considering the distribution of the number of conflicts per entry in the action table. The basic intuition is this: if a grammar is 'close' to LR(1), then its LR(1) action table should have relatively few entries with one or more conflicts. On the other hand, if a grammar is relatively non-LR(1)-like, then a large proportion of its table entries should have multiple conflicts. Consider the conflict set distribution for the phrase structure grammar of S-structure for English shown in figure 4.10. Note that the number of conflicts shown does not take into account the reduction in non-determinism due to the unbounded lookahead mechanism. With a more sophisticated LR(1) table generator such as Pager's (as mentioned earlier), certain states may be merged, thereby reducing the frequency of some of the conflict entries. With these caveats in mind, the median and average number of conflicts per state is zero and 0.5, respectively.



No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.4
σ_n	0.7
No. of States	123

Figure 4.11 Revised English S-structure grammar conflict statistics.

For interpretation purposes, an average of one would roughly correspond to binary branching in the computation search space every time the machine was forced to pick an action.[32]

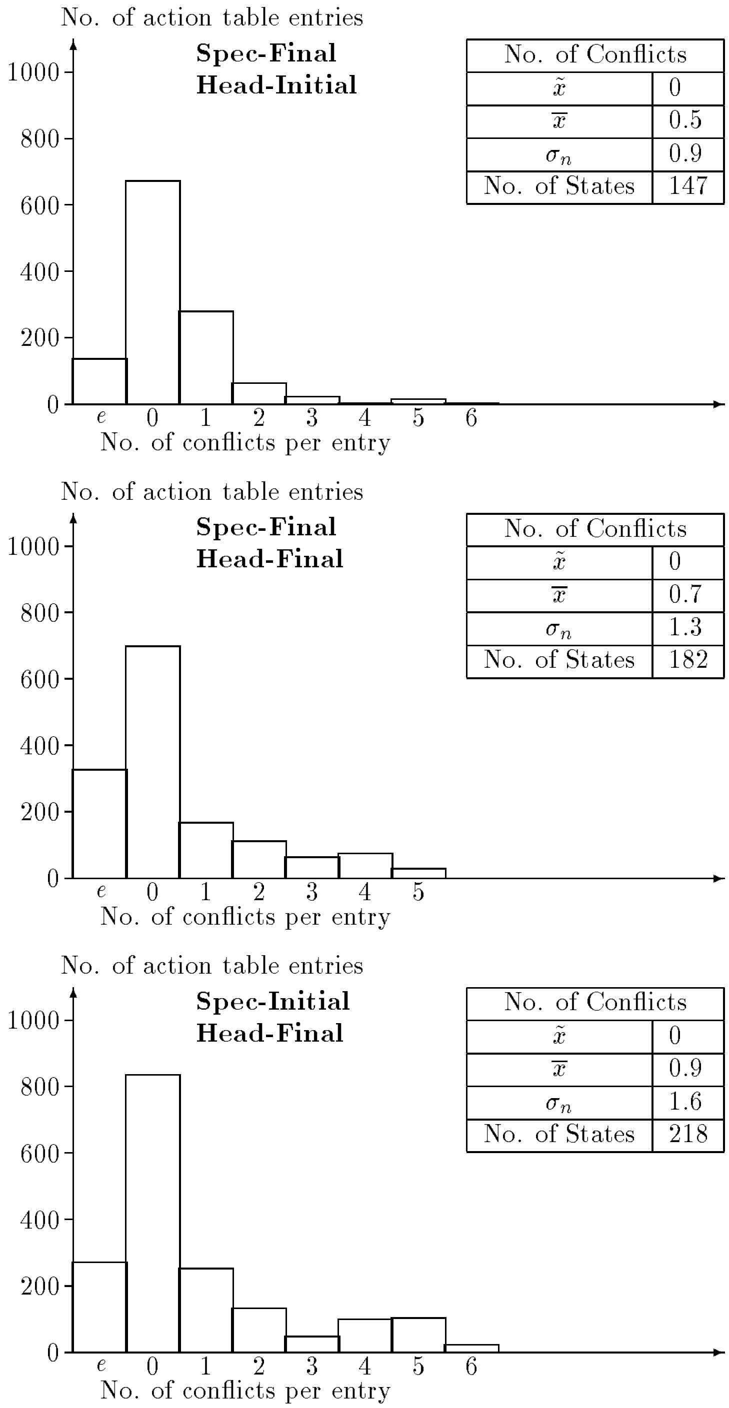
If the equivalence class shortcut described in the previous section is taken into account, there is a significant shift in the overall conflict distribution as shown in figure 4.11.[33] However, the conflict distribution essentially retains the same profile.

As for the question whether the one symbol lookahead mechanism is profitably utilized in the grammar of S-structure: out of the 123 states generated, 83 (or approximately 67%) benefitted in some degree; that is, there exists at least one table entry that had strictly fewer conflicts when lookahead is added. There were 252 entries (out of a total of 984) that had the number of conflicts reduced by (exactly) one. The degree of conflict was reduced by two in 26 other entries. Hence, we can conclude that lookahead is certainly a useful feature.

The rule schema used to encode the S-structure grammar also allows different values of the specifier-head and head-complement parameters to be specified. The conflict statistics shown so far are only valid when the specifier precedes its head, i.e. specifier-initial, and the head precedes its complements, i.e. head-initial, as in English. The question is: Will the distribution of conflicts change significantly for different values of the $\overline{\text{X}}$-parameters? As figure 4.12 shows, there seems to be rela-

[32] Of course, strictly speaking, it is a vast oversimplification to correlate the average number of conflicts (which is only a *static* measure) with the branching factor at parse time. For example, not all entries are equally likely to be visited at parse time — which, presumably, should be a function of sentence type.

[33] In this case, we have taken the number of conflicts to be the maximum number of different actions that can be taken before shortcutting occurs.



No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.5
σ_n	0.9
No. of States	147

No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.7
σ_n	1.3
No. of States	182

No. of Conflicts	
$\tilde{x}$	0
$\overline{x}$	0.9
σ_n	1.6
No. of States	218

Figure 4.12 Conflicts statistics for various parameter settings.

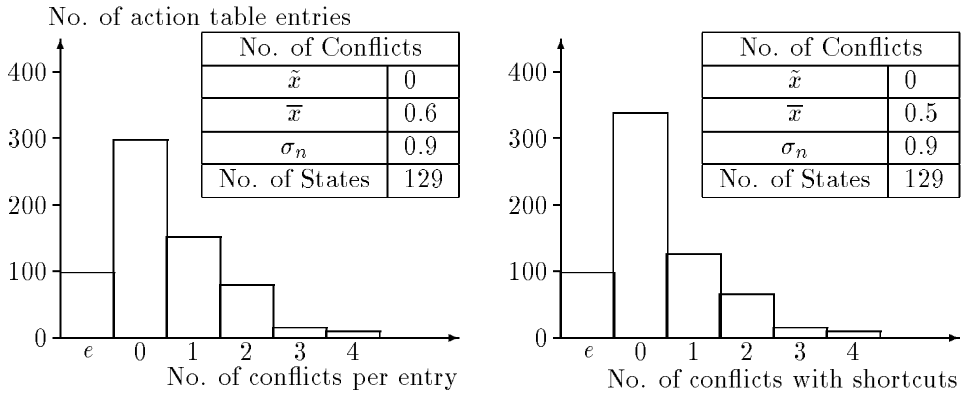


Figure 4.13 Japanese S-structure grammar conflict statistics.

tively little difference in the distribution of conflicts for all four parameter settings. Hence we can conclude that the results obtained for the S-structure grammar seems to be robust, at least with respect to parameter setting. However, informally speaking, there seems to be some correlation between the average number of conflicts per table entry and the relative position of the head with respect to its specifier and complements. The lowest and highest averages were obtained when the head preceded and followed both its specifier and complements, respectively. In the other two cases, i.e. when the head separates the specifier from the complements, the averages fell inbetween the two extremes. This result is perhaps not surprising for a left-to-right parsing algorithm since, typically, a head will carry lexical information that helps to prune the space of possible complements and specifiers.

The grammar for Japanese differs in coverage with respect to scrambling and syntactic *wh*-movement from that of English, and therefore, strictly speaking, should not be directly compared. However, for completeness, the corresponding conflict distributions (both with and without the conflict shortcut procedure) are shown in figure 4.13. We will just comment that the reduction in the number of conflicts due to the equivalence class shortcut is less marked than for the English grammar. This can be traced to the fact that heads precedes their complements in English, and hence subcategorization information of individual heads, e.g. the θ-grid, can be profitably used to prune the search space for parsing following complements in a left-to-right parser. (By contrast, in Japanese, heads will follow their complements, and hence, complements will generally be parsed before heads.) However, despite these differences, note that the basic distribution of the conflicts is similar to that for English.

Finally, let us compare the canonical LR(1)-based machines built for English with the corresponding LALR(1)-based machines that, as mentioned earlier, trade early error detection for reduced table size. It is a simple matter to derive the corresponding LALR(1) CFSM given a canonical LR(1) CFSM. (See Algorithm 6.4 in Aho & Ullman [7].) LALR(1) CFSMs were built for each of the four parameter settings. The pertinent results are summarized in the following table:

	Spec-Initial Head-Initial	*Spec-Final Head-Initial*	*Spec-Final Head-Final*	*Spec-Initial Head-Final*
No. of states LR(1)/LALR(1):	123/79	147/76	182/86	218/92
Percentage of error entries LR(1)/LALR(1):	17%/6%	11%/6%	22%/15%	16%/5%

In each case, the LALR(1)-based machine contains approximately half the number of states of the corresponding canonical LR(1)-based machine. This difference is not significant since, in both cases, the CFSM can be considered to be relatively small. Also, the distribution of conflicts were found to be very similar (modulo the general reduction in the size of the LALR(1)-based tables), except in the ratio of error entries as a percentage of the total number of table entries. As the table shows, in all cases, a larger proportion of error entries were consistently found in the LR(1)-based machines. This difference is due to the fact that for some LR(1) error entries, the corresponding entry in the LALR(1) table will contain an (erroneous) reduce action that will be detected at a later state. In general, this result is expected because an LALR(1) machine may perform more reduce actions (but not more shifts) than its LR(1) counterpart.

Chapter 5

Principle Ordering

In this chapter we will examine if, and how, a parser can exploit the freedom to reorder principles to avoid doing unnecessary work. In particular, we will investigate the following questions:

1. What effect, if any, does principle ordering have on the amount of work needed to parse a given sentence?

2. If the effect of principle ordering is significant, then are some orderings much better than others?

3. If so, is it possible to predict (and explain) which ones these are?

By characterizing principles in terms of the purely computational notions of *filters* and *generators*, we will show how principle ordering can be exploited to minimize the amount of work performed in the course of parsing. Basically, some principles such as Move-α (a principle relating 'gaps' and 'fillers') and free indexation (a principle relating referential items) are "generators" in the sense that they may provide more structures as output than they receive as inputs. For example, in the case of Move-α, there are likely to be many possible assignments of movement chains to a given structure. Other principles, like the θ-criterion which places restrictions on the assignment of thematic relations, or the Case Filter which requires certain noun phrases to be marked with abstract Case, act as filters and weed-out ill-formed structures.

The effects of principle ordering can be investigated in the current system by specifying (separately from principle definitions of the kind discussed in the previous chapter) the order in which principles are to be processed. Note that no matter which ordering is chosen, the system makes use of the same set of principles, and thus will make the same grammaticality judgements and return the same LF structures as output. The only visible difference will be that different orderings will result in different parsing times.

Roadmap

We will begin by reviewing the underlying processing model that will be used for principle ordering. Then, we will investigate how significant a factor re-ordering principles can be for parsing in that model. We will also explain under what conditions certain orderings are preferable to others. Using the observation that the efficiency of a particular order depends on local conditions, i.e. at the level of individual structures, we will go on to describe a dynamic ordering mechanism that attempts to re-schedule the order of processing based on structural cues in conjunction with simple heuristics using the filters-and-generators model. Finally, we will describe a small revision to the computational notion of a filter that will allow the use of partial dependency-directed control mechanism.

5.1 The Model of Processing

We will adopt a very simple model of processing for exploring the effects of principle ordering. In this model, parser operations that correspond (roughly speaking) to linguistic principles will be the atomic elements that are subject to re-ordering. For example, consider the following:

$$\mathrm{SF} \rightarrow \boxed{PS} \rightarrow \underbrace{\boxed{P_2} \rightarrow \boxed{P_1} \cdots \rightarrow \boxed{P_{n-1}} \rightarrow \boxed{P_n}}_{\pi(P_1, P_2, \ldots, P_n)} \rightarrow \mathrm{LF}$$

Here, each box represents a single principle that is connected to both the previous and next principle to be processed in a "production line" sequence. Given an input Surface Form (SF), the initial PS parser operation will produce the corresponding phrase structures that will be passed from one parser operation to the next in a "stream-like" fashion. Because the current system employs a backtracking depth-first search strategy, the set of permissible structures at each point will be produced one at a time. Note that the structures passed between each operation will be phrase structures for the *complete* input. (The case where individual operations will act on partial structures that represent portions of the input SF will be dealt with in the next chapter.) The principle ordering problem thus is to find the permutation

$\pi(P_1, P_2, \ldots, P_n)$ that minimizes the cost of producing all possible output Logical Forms (LFs) corresponding to a given input SF. In this and the following chapter, we will adopt the number of parser operations performed during parsing as a simple, machine-independent measure of the cost of parsing.[1]

5.2 The Principle Ordering Problem

Let us begin by examining the question of whether there exists a significant performance difference between various orderings? Equivalently, is the "principle ordering problem" really a problem at all? An informal experiment was conducted using the system to provide some indication of the magnitude of the problem. As mentioned earlier, the current system consists of about two dozen principles. Clearly, it would be impractical to examine all the possible permutations.[2] However, it turns out that order-of-magnitude variations in parsing times can be achieved merely by permuting the order of a few principles. For instance, consider the graph shown in figure 5.1.

All of the parser configurations shown here have been derived from a single "default" ordering (`Default`). That is, the order of principles will be the same except that a few selected principles will have been fronted as far forward as possible. For example, `Case` represents a parser with the same ordering as `Default` except that the operations of Case theory have been 'favoured' over all other modules. In the current system, this means that the operations *inherent Case assignment*, *structural Case assignment*, *Case filter* and *Case condition on traces* will be processed as early as possible.

Each configuration was tested on a small set of fifteen examples (including both well-formed and ill-formed sentences) from Lasnik & Uriagereka. This selection represents a cross-section of different linguistic phenomena including parasitic gaps, *that*-trace (ECP) effects, strong crossover, superiority, subjacency, exceptional Case marking (ECM), passivization, and Binding effects. These examples are shown in figure 5.2.

---

[1] For simplicity, we assume that all parser operations will have comparable cost. The actual ratios will, of course, depend on the precise definition of each principle and the efficiency of the compilation process. However, this measure correlates quite well with actual parsing times. One reason why the approximation seems to hold is that one of the major tasks involved in applying a principle is the work required in simply "walking" through each sub-phrase for a given structure.

[2] Actually, only a fraction of the possible $n!$ permutations are *valid* orderings due to logical dependencies between various parser operations — see the dependency graph (figure 1.6) and accompanying discussion in section 1.5.2. The relevant notion of validity here is that correctness with respect to grammaticality judgements should be preserved. Estimates of the number of valid permutations have been made for an earlier version of the system which contained one dozen principles. Out of the almost 500 million possible permutations, only about half a million are actually valid. Even so, this is far too many to test exhaustively.

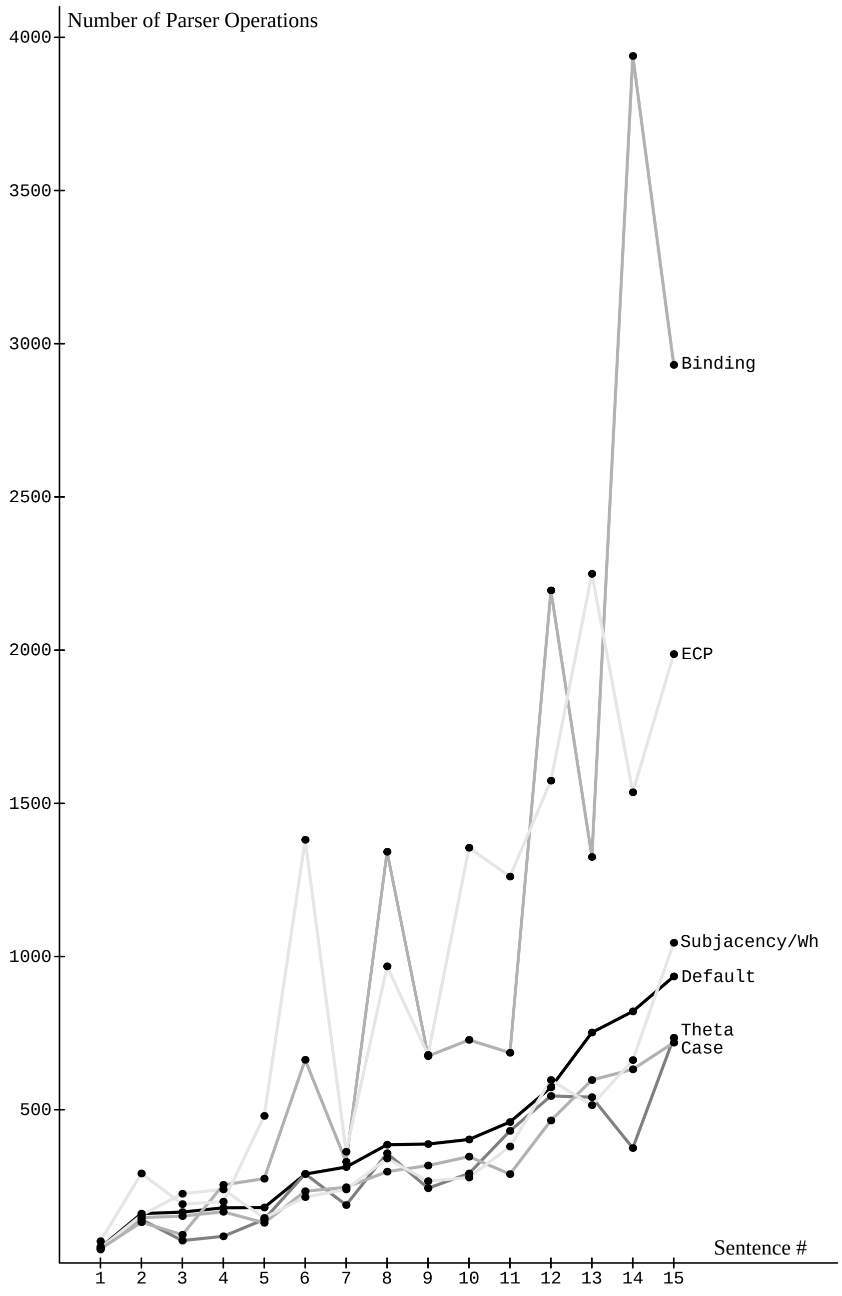


Figure 5.1 The effect of principle ordering. [3]

(1)	*What will who read (4:21c)	
(2)	Who will read what (4:21b)	*Superiority*
(3)	*John's mother likes himself (2:8)	*Binding condition A violation*
(4)	John's mother likes him (2:16b)	*Pronoun Binding ambiguity*
(5)	*Which book did you file the report without reading (3:18)	*cf. (11)*
(6)	*Who does Mary wonder why John hit (4:57)	*Subjacency*
(7)	*Who do you think that saw Bill (4:8a)	that-*trace effect*
(8)	John was believed to be intelligent (1:49c)	*Exceptional passivization*
(9)	Who do you think saw Bill (4:8b)	*cf. (7)*
(10)	Who that John knows does he like (1:15a)	Wh-*movement/pronoun ambiguity*
(11)	Which report did you file without reading (3:17)	*Parasitic gap*
(12)	*It was believed John to be intelligent (1:49b)	*cf. (8)*
(13)	Who does he think Mary likes (2:45a)	*Strong crossover*
(14)	I want to visit you (2:66a)	*Control of PRO*
(15)	*I am eager John to be here (1:22)	*Case violation*

Figure 5.2 Test examples from Lasnik & Uriagereka.

Each point on the graph represents the total number of parser operations invoked for a given sentence. (The examples are not given in any particular order.) As the graph clearly illustrates, the order in which principles are processed can make a large difference in the amount of work required to obtain the same grammatical judgements. Other experiments on the system have shown that this order of magnitude in variation is possible even for four word sentences such as *I want to win*.

5.2.1 Explaining the Variation in Principle Ordering

The variation for various principle orderings that we observed in the previous section can be explained by assuming that overgeneration is the main problem, or bottleneck, for systems such as the current one. That is, in the course of parsing a single sentence, the parser will hypothesize many different structures. These structures are produced by operations that function as generators in the sense discussed earlier. Most of these structures, the ill-formed ones in particular, will be accounted for by one or more linguistic filters. Computationally speaking, filters are operations that produce N or fewer structures given N structures as input.[4] A sentence

[3]Note that the discrete data points for each parser configuration have been connected for presentation purposes only.

[4]However, one complicating factor is that some parser operations may behave either as filters or generators depending on the degree of instantiation of features in the phrase structures that are presented as input to such operations. For instance, the *theta criterion* operation can function in either mode depending on processing order. For example, (following Chomsky [9]) the last element

will be deemed acceptable if there exists one or more structures that satisfy every applicable filter. Note that even when parsing grammatical sentences, overgeneration will produce ill-formed structures that need to be ruled out. Given that our goal is to minimize the amount of work performed during the parsing process, we would expect a parse using an ordering that requires the parser to perform extra work compared with another ordering to be slower.

Overgeneration implies that we should order the linguistic filters to eliminate ill-formed structures as quickly as possible. For these structures, applying any parser operation other than one that rules it out may be considered as doing extra, or unnecessary, work (modulo any logical dependencies between principles).[5] However, in the case of a single well-formed structure, principle ordering cannot improve parser performance. By definition, a well-formed structure is one that passes all relevant parser operations. Unlike the case of an ill-formed structure, applying one operation cannot possibly preclude having to apply another.

5.2.2 Optimal Orderings

Since some orderings perform better than others, a natural question to ask is: Does there exist a "globally" optimal ordering? The existence of such an ordering would have important implications for the design of the control structure of any principle-based parser. For example, the current system also implements a novel "dynamic" control structure in the sense that it tries to determine an ordering-efficient strategy for every structure generated. If such a globally optimal ordering could be found, then we can do away with the run-time overhead and parser machinery associated with determining individual orderings. That is, we can build an ordering-efficient parser simply by "hardwiring" the optimal ordering into its control structure. Unfortunately, no such ordering can exist.

The impossibility of the globally optimal ordering follows directly from the "eliminate unnecessary work" ethic. Computationally speaking, an optimal ordering is one that rules out ill-formed structures at the earliest possible opportunity. A *globally* optimal ordering would be one that always ruled out every possible ill-formed structure without doing any unnecessary work. Now, consider the following three

of a chain must be assigned a θ-role if and only if it is headed by an argument. Consider the case when the head of the chain is an empty NP without any features (all empty NPs are initially inserted with null features.) In the current implementation, a simple generate-and-test strategy has been adopted; that is, *theta criterion* will function as a generator. If the empty head has to be an argument, it will try each possibility: *PRO*, *pro* and variable (but not an anaphor). On the other hand, if the empty head has been previously determined as one of these, then it will function as a filter.

[5] In the current system for example, the Case Filter operation which requires that all overt noun phrases have abstract Case assigned, is dependent on both the inherent and structural Case assignment operations. That is, in any valid ordering the filter must be preceded by both operations.

structures (taken from Lasnik & Uriagereka):

(1) a. *John$_1$ is crucial $[_{CP}[_{IP}\ t_1$ to see this $]]$
b. *$[_{NP}$John$_1$'s mother $][_{VP}$ likes himself$_1]$
c. *John$_1$ seems that he$_1$ likes t_1

Example (1a) violates the Empty Category Principle (ECP). Hence the optimal ordering must invoke the *ECP* operation before any other operation that it is not dependent on. On the other hand, example (1b) violates a Binding theory principle, namely Condition A. Hence, the optimal ordering must also invoke *Condition A* as early as possible. In particular, given that the two operations are (locally) independent, the optimal ordering must order *Condition A* before the *ECP* and vice-versa. Similarly, example (1c) demands that the *Case Condition on Traces* operation must precede the other two operations. To conclude, because optimality is a local property rather than a global one, there can be no single optimal ordering.

5.3 Dynamic Ordering

The principle ordering problem can be viewed as a limited instance of the well-known conjunct ordering problem (Smith & Genesereth [50]). Given a set of conjuncts, we are interested in finding all solutions that satisfy all the conjuncts simultaneously. The parsing problem is then to find well-formed structures (i.e. solutions) that satisfy all the parser operations simultaneously. Moreover, we are particularly interested in minimizing the cost of finding these structures by re-ordering the set of parser operations (i.e. conjuncts).

This section outlines the heuristics used by the parser to automatically determine the minimum cost ordering for a given structure. The parser contains a dynamic ordering mechanism that attempts to compute a separate minimum cost ordering for each phrase structure generated during the parsing process.[6] The mechanism can be subdivided into two distinct phases as shown in figure 5.3. First, we will describe how the dynamic ordering mechanism decides which principle is the most likely candidate for eliminating a given structure. Then, we will explain how it makes use of this information to re-order parser operation sequences to minimize the total work performed by the parser.

[6] In their paper, Smith & Genesereth drew a distinction between "static" and "dynamic" ordering strategies. In static strategies, the conjuncts are first ordered, and then solved in the order presented. By contrast, in dynamic strategies the chosen ordering may be revised between solving individual conjuncts. We have employed a dynamic strategy in the current implementation.

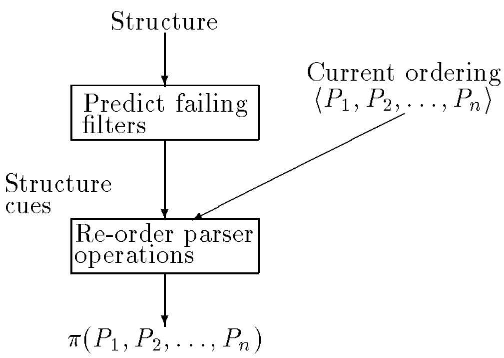


Figure 5.3 The dynamic ordering mechanism.

Predicting Failing Filters

Given any structure, the dynamic ordering mechanism attempts to satisfy the "eliminate unnececessary work" ethic by predicting a "failing" filter for that structure. More precisely, it will try to predict the principle that a given structure violates on the basis of simple structure cues. Since the ordering mechanism cannot know whether a structure is well-formed or not, it assumes *a priori* that all structures are ill-formed and attempts to predict a failing filter for every structure. In order to minimize the amount of work involved, the types of cues that the dynamic ordering mechanism can test for are deliberately limited. Only inexpensive tests such as whether a category contains certain features (e.g. [±anaphoric], [±infinitival], or whether it is a trace or a non-argument) are used. Any cues that may require significant computation, such as searching for an antecedent, are considered to be too expensive. Each structure cue is then associated with a list of possible failing filters. Some instances of this mapping is shown in figure 5.4. The system then picks one of the possible failing filters indicated by the structure cues.[7]

Note that the correspondence between each cue and the set of candidate filters may be systematically derived using simple causality from the definitions of the relevant principles. For example, Principle A of the Binding theory deals with the conditions under which antecedents for anaphoric items, such as *each other* and *himself*, must appear. Hence, Principle A can only be a candidate failing

[7] Obviously, there are many ways to implement such a selection procedure. Currently, the system uses a voting scheme based on the frequency of cues. The (unproven) underlying assumption is that the probability of a filter being a failing filter increases with the number of occurrences of its associated cues in a given structure. For example, the more traces there are in a structure, the more likely it is that one of them will violate some filter applicable to traces, such as the Empty Category Principle (ECP).

Structure Cue	*Possible Failing Filters*
`ecm(_)` *Exceptional Case Marker*	Case filter, Case condition on traces, $\overline{\text{S}}$-deletion
`inf(_)` *Infinitival clause*	Case filter, Case condition on traces,
`wh(+)`	*Wh*-in-syntax
`trace`	ECP, Case condition on traces, Subjacency
`noCaseMark` (*e.g. passive or intransitive*)	Case filter, Theta criterion D-structure Theta condition
`nonarg`	Theta criterion, D-structure Theta condition
`a(+)` *anaphor*	Binding theory Principle A
`p(+)` *pronoun*	Binding theory Principle B
`op(+)` *operator*	License Operator/Variables
`adj(nonhead)` *non-head adjunction*	License adjuncts

Figure 5.4 Mapping structure cues to failing filters.

filter for structures that contain an item with the [+anaphoric] feature. Other correspondences may be somewhat less direct: for example, the Case Filter merely states that all overt noun phrase must have abstract Case. Now, the conditions under which a noun phrase may receive abstract Case are defined by two separate operations, namely, *inherent Case assignment* and *structural Case assignment*. It turns out that an instance where *structural Case assignment* will not assign Case is when passive morphology occurs with a verb that normally assigns Case. Hence, the presence of a passive verb in a given structure may cause an overt noun phrase to fail to receive Case, which in turn may cause the Case Filter to fail.[8]

[8] It is possible to automate the process of finding structure cues simply by inspecting the closure of the definitions of each filter and all dependent operations. One idea is to collect the negation of all conditions involving category features. For example, if an operation contains the condition "`\+ Item has_feature noCaseMark`", then we can take the presence of an intransitive item as a possible reason for failure of that operation. However, this approach has the potential problem

The failing filter mechanism can been seen as an approximation to the *Cheapest-First* heuristic in conjunct ordering problems. It turns out that if the cheapest conjunct at any given point will reduce the search space rather than expand it, then it can be shown that the optimal ordering must contain that conjunct at that point. Obviously, a failing filter is a "cheapest" operation in the sense that it immediately eliminates one structure from the set of possible structures under consideration.

There are two other notable features of the failure filter mechanism:

- First, the failing filter approach does not take into account the behaviour of generators. For example, the operations corresponding to $\overline{\mathrm{X}}$ phrase structure rules, Move-α, free indexation and LF movement are the main generators in the current system. (Similarly, the operations that we have previously referred to as "filters" may be characterized as parser operations that, when given N structures as input, pass N and possibly fewer than N structures.) Due to logical dependencies, it may be necessary in some situations to invoke a generator operation before a failure filter can be applied. For example, Principle A of the Binding theory is logically dependent on the generator *free indexation* to generate the possible antecedents for the anaphors in a structure.[9] Consider the possible binders for the anaphor *himself* in *John thought that Bill saw himself* as shown below:

  (2) a. *John$_i$ thought that Bill$_j$ saw himself$_i$
  b. John$_i$ thought that Bill$_j$ saw himself$_j$
  c. *John$_i$ thought that Bill$_j$ saw himself$_k$

  Only in example (2b), is the antecedent close enough to satisfy the locality restrictions imposed by Principle A. Note that Principle A had to be applied a total of three times in the above example in order to show that there is only one possible antecedent for *himself*. This situation arises because of the general tendency of generators to overgenerate. This characteristic can greatly magnify the extra work that the parser performs when the dynamic ordering mechanism picks the wrong failing filter. Consider the ill-formed structure **John$_i$ seems that he likes* t$_i$ (a violation of the principle that traces of noun

of generating too many cues. Although, it may be relatively inexpensive to test each individual cue, a large number of cues will significantly increase the overhead of the ordering mechanism. Furthermore, it turns out that not all cues are equally useful in predicting failure filters. One solution may be to use "weights" to rank the predictive utility of each cue with respect to each filter. Then an adaptive algorithm could be used to "learn" the weighting values, in a manner reminiscent of Samuels [49]. The failure filter prediction process could then automatically eliminate testing for relatively unimportant cues. This approach will be the subject of future work.

[9] See figure 1.6 on page 30.

phrase cannot receive Case.) If however, Principle B of the Binding theory is predicted to be the failure filter (on the basis of the structure cue *he*), then Principle B will be applied repeatedly to the indexings generated by *free indexation*. On the other hand, if the *Case condition on traces* operation was correctly predicted to be the failing filter, then *free indexation* need not be applied at all. The current system takes a conservative approach by also adopting the heuristic of delaying generators as far as possible to avoid unnecessary "fan-out". All potential filters are prioritized by the number of generators that they are dependent on.

- The mechanism is dynamic because it will re-evaluate (and perhaps revise) its choice of failing filter after a given phrase structure becomes more fully instantiated. For example, after chains are formed by *trace theory*, it re-computes the failure filter because some empty categories may be instantiated as traces. Similarly, after *functional determination* has applied, all empty noun phrases in A-positions will have their [±anaphoric,±pronominal] features instantiated.

In informal experiments, the implemented mechanism performs quite well in many of the test cases drawn from the reference text. Figure 5.5 compares the dynamic ordering mechanism (represented by the dashed line) to the fixed orderings shown previously in figure 5.1.[10] All results shown have been normalized with respect to the number of parser operations performed by `Dynamic`. (The results for `Binding` and `ECP` have been omitted from the graph for presentation purposes.) As the ratios indicate, `Dynamic` performs better than all the fixed orderings except in a few cases. This result is highlighted in the table of average ranks shown in figure 5.6. We should add that no "tweaking" (e.g. by adjusting the weights assigned to each structure cue) has been done for the comparison. Figure 5.7 should provide some idea of the range of accuracy of the prediction mechanism. Although the dynamic ordering mechanism performs well in comparison to the fixed orderings (even in some cases where the "hit" rate is low), it is by no means foolproof. There are many cases where the prediction mechanism will trigger an unprofitable re-ordering of the default order of operations. (We will return to discuss how this might be partially remedied in the next section.)

A few comments about further improvements in performance are in order here:

1. Given the poor hit rates for some of the examples (illustrated in figure 5.7), it should be possible to further improve the overall performance of dynamic

[10] Like the other parsers, `Dynamic` is also based on `Default`. As a last resort, in cases where the prediction mechanism cannot decide on an appropriate filter, `Dynamic` will adopt the underlying order used by `Default`.

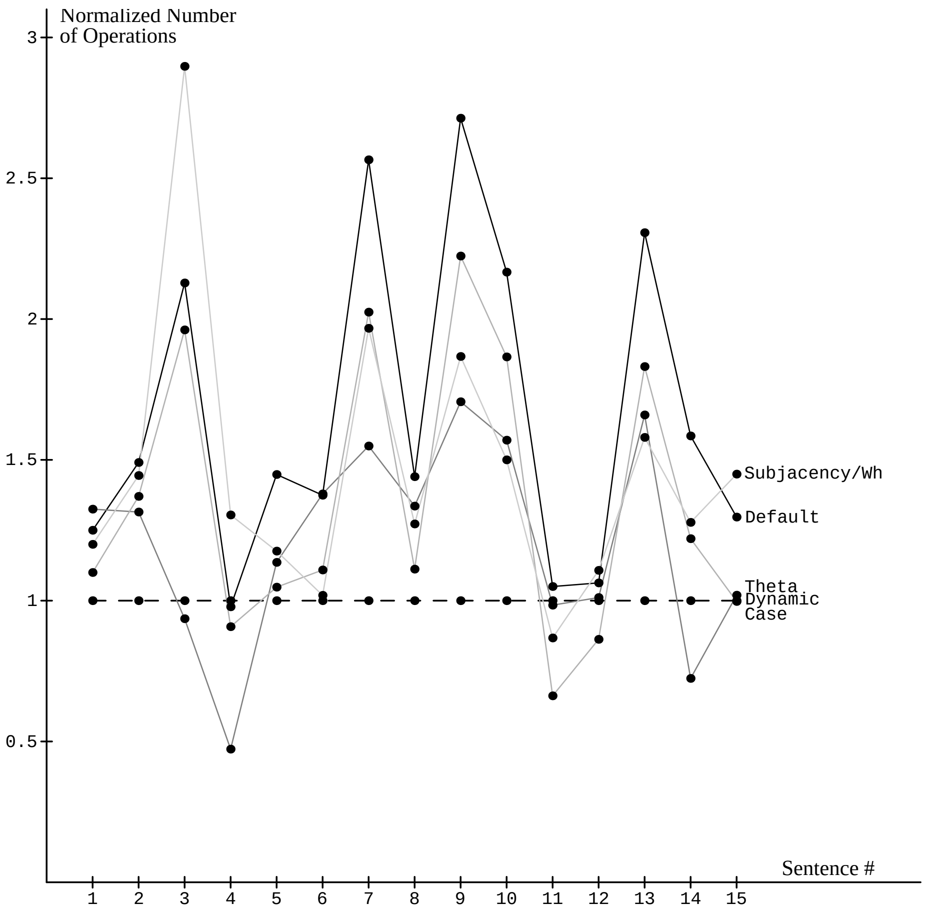


Figure 5.5 Comparison of dynamic ordering with fixed orders using normalized counts.

Parser Configuration	Average Rank
`Dynamic`	1.7/7
`Case`	2.7/7
`Theta`	2.9/7
`Subjacency/`*Wh*`-in-syntax`	3.8/7
`Default`	4.7/7
`Binding`	5.7/7
`ECP`	6.5/7

Figure 5.6 Comparison of dynamic ordering with fixed orders using ranks.

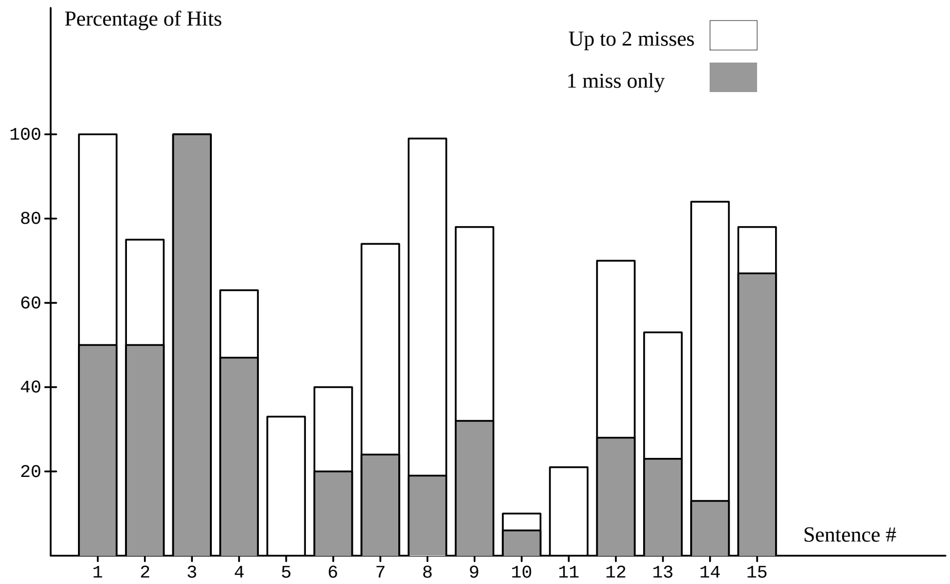


Figure 5.7 Percentage of hits achieved by dynamic ordering.

ordering, by tweaking the relative weights of structural cues, or possibly by "training" the mechanism over a suitable sample of sentences.

2. A more sophisticated prediction scheme, perhaps one based on more complex cues, could also increase the accuracy of the ordering mechanism. However, it is probably not cost-effective to do so. In general, there seems to be no *simple* way to determine whether a given structure will violate a certain principle.[11] That is, as far as one can tell, it is difficult to produce a cheap, but effective approximation to a filter (relative to the cost of the real operation). For example, in Binding theory, it is difficult to determine if an anaphor and its antecedent satisfies the complex locality restrictions imposed by Principle A without actually doing some search for a binder. Simplifying the locality restrictions is one way of reducing the cost of approximation, but the very absence of search is the main reason why the overhead of the present mechanism is not overwhelming.[12]

Logical Dependencies and Re-ordering

The second phase of the dynamic ordering mechanism is to re-schedule the parser operations so that the failing filter is performed as early as possible. Simply "fronting" the predicted failing filter to the front of the operations queue may violate logical dependencies between various parser operations. For example, suppose the Case filter was chosen to be the failing filter. To create the conditions under which the Case filter can apply, both Case assignment operations, namely, *inherent Case assignment* and *structural Case assignment*, must be applied first. Hence, fronting the Case Filter will also be accompanied by the subsequent fronting of both assignment operations — unless, of course, they have already been applied to the structure in question. Finally, in the case where the predicted failing filter fails to rule some structure, the re-ordering mechanism will then front the second choice and (if necessary) a third, and so on.

[11] If such a scheme can be found, then it can effectively replace the definition of the principle itself.

[12] The machine-independent results given above do not include the work spent on the dynamic ordering mechanism. Informal profiling suggests that the overhead of the mechanism varies from about 10% to 30%. This is already significant enough to outweigh the advantage of doing dynamic ordering on some cases. (Of course, the actual percentage would depend on the particular PROLOG system used.) For example, under symbolics PROLOG, the actual time taken to parse sentence (1) in figure 5.2 is virtually identical to the second best fixed ordering (`Theta`).

5.4 Linguistic Filters and Determinism

In this section we describe how the filters and generators model may be exploited to further improve the performance of the system for some operations. More precisely, we make use of an additional (computational) property of linguistic filters to improve the backtracking behaviour of the parser. The behaviour of this optimization will turn out to complement that of the ordering selection procedure quite nicely. That is, the optimization is most effective in exactly those cases where the selection procedure is least effective.

We hypothesize that linguistic filters, such as the Case filter, Binding conditions, ECP, and so on, may be characterized as follows:

> **Hypothesis**: Linguistic filters are side-effect free conditions on configurations

In terms of parser operations, this means that filters should never cause structure to be built or attempt to fill in feature slots.[13] Moreover, computationally speaking, the parser operations corresponding to linguistic filters should be deterministic. That is, any given structure should either fail a filter or just pass. A filter operation should never need to succeed more than once, simply because it is side-effect free.[14] By contrast, operations that we have characterized as generators, such as *trace theory* and *free indexation*, are not deterministic in this sense.

Given the above hypothesis, we can cut down on the amount of work done by the parser by modifying its behaviour for filter operations. Currently, the parser employs a backtracking model of computation. If a particular parser operation fails, then the default behaviour is to attempt to re-satisfy the operation that was called immediately before the failing operation. In this situation, the parser will only attempt to re-satisfy the preceding operation if it happens to be a generator. When

[13] So far, we have not encountered any linguistic filters that require either structure building or feature assignment. Operations such as θ-role and Case assignment are not considered filters in the sense of the definition given in the previous section. In the current implementation, these operations will never fail. However, definitions that involve some element of 'modality' are potentially problematic. For example, Chomsky's definition of an *accessible SUBJECT*, a definition relevant to the principles of Binding theory, contains the following phrase "*... assignment to α of the index of β would not violate the (i-within-i) filter* $*[_{\gamma_i} ...\delta_i...]$". A transparent implementation of such a definition would seem to require some manipulation of indices. However, Lasnik (p.58) points out that there exists an empirically indistinguishable version of accessible SUBJECT without the element of modality present in Chomsky's version.

[14] It turns out that there are situations where a filter operation (although side-effect free) could succeed more than once. For example, the linguistic filter known as the "Empty Category Principle" (ECP) implies that all traces must be "properly governed". A trace may satisfy proper government by being either "lexically governed" or "antecedent governed". Now consider the structure $[_{CP}$ *What*$_1$ *did you* $[_{VP}$ *read* $t_1]]$. The trace t_1 is both lexically governed (by the verb *read*) and antecedent governed (by its antecedent *what*). In the current implementation, the ECP operation can succeed twice for cases such as t_1 above.

the preceding operation is a filter, then the parser will skip the filter and, instead, attempt to resatisfy the next most recent operation and so on.[15] For example, consider the following calling sequence:

$$\boxed{G_1} \cdots \longrightarrow \boxed{G_2} \longrightarrow \boxed{F_1} \longrightarrow \boxed{F_2} \longrightarrow \boxed{F_3}$$

Suppose that a structure generated by generator G_2 passes filters F_1 and F_2, but fails on filter F_3. None of the three filters could have been the cause of the failure by the side-effect free hypothesis. Hence, we can skip trying to resatisfy any of them and backtrack straight to G_2.

Note that this optimization is just a limited form of *dependency-directed backtracking.* Failures are traced directly to the last generator invoked, thereby skipping over any intervening filters as possible causes of failure. However, the backtracking behaviour is limited in the sense that the most recent generator may not be the cause of a failure. Consider the above example again. The failure of F_3 need not have been caused by G_2. Instead, it could have been caused by structure-building in another generator further back in the calling sequence, say G_1. But the parser will still try out all the other possibilities in G_2 first.

Consider a situation in which the principle selection procedure performs poorly. That is, for a particular ill-formed structure, the selection procedure will fail to immediately identify a filter that will rule out the structure. The advantages of the modified mechanism over the default backtrack scheme will be more pronounced in such situations — especially if the parser has to try several filters before finding a "failing" filter. By contrast, the behaviour of the modified mechanism will resemble that of the strict chronological scheme in situations where the selection procedure performs relatively well (i.e. when a true failing filter is fronted). In such cases, the advantages, if significant, will be small. (In an informal comparison between the two schemes using about eighty sentences from the reference text, only about half the test cases exhibited a noticeable decrease in parsing time.)

5.5 Conclusions

We have seen that principle ordering is an important consideration in the design of efficient principle-based parsers. The order-of-magnitude (or greater) differences

[15] This behaviour can be most easily simulated using a 'throw' exception mechanism or the 'cut' predicate in PROLOG. We can route all calls to filter operations through such devices.

in the amount of work required for parsing suggests that it is worthwhile to try to optimize parser control. We have seen how these differences can be explained by characterizing parser operations as being filters or generators. The fact that no globally optimal ordering exists has motivated the dynamic ordering mechanism described in the last section. An important question that remains is whether the same performance results hold for different languages. Preliminary experiments using the Japanese *wh*-questions (from Lasnik & Saito [34]) have been inconclusive.[16] Obviously, more tests with a wider range of examples will be necessary to reveal any significant differences. Also, it remains to be seen whether dynamic ordering will be profitable if the principles are revised or extended. However, the experience with early prototypes of the current system suggests that as the number of principles increases the effect of principle ordering also becomes more pronounced.

[16] Due to language parameterization, there are differences in the number of choice points of some parser operations. For example, functional determination is deterministic for English. However, the possibility of assigning *pro* in Japanese (*pro* not being available in English) turns functional determination into a generator. Due to the small number of test cases, it is not clear whether this particular difference will be significant enough to cause different orderings to be preferred for Japanese. (The situation is also complicated by the fact that the phrase structure grammars for S-structures are not strictly comparable.)

Chapter 6

Principle Interleaving

In the previous chapter, we explored a simple model of parsing that consists of two distinct phases: the first being the phase in which phrase structures are built, and the second being the phase in which ill-formed structures are "weeded out" by applying each well-formedness condition, or principle, in turn. Note that in such a scheme, there is a strict separation between phrase structure construction and principle application. That is, phrase structure for a complete sentence will be built before it is tested for well-formedness. An alternative strategy may be based on the observation that we need not wait until phrase structure construction is complete before applying certain principles. In many cases, principles exhibit *locality of reference* in the sense that may apply only to certain sub-structures within a given phrase structure. (We discuss such an example immediately below.) We can take advantage of this by co-routining, or *interleaving*, principle application with phrase structure construction as individual phrases are built. The motivating principle here is that if some sub-structure is found to be ill-formed, then any complete structure built from that sub-structure will also be ill-formed. For example, the sentence *John saw he* is ill-formed — cf. the well-formed sentence *John saw him*. In fact, the verb phrase corresponding to *saw he* is ill-formed no matter what elements precede or follow it. For example, *Mary believes John saw he*, *John saw he win*, and *I thought that Mary believed John saw he* are all ill-formed sentences. The problem here is that Case assignment, a principle of Case theory, states that verbs such as *see* assign accusative Case to noun phrase elements in object position such as *he*. Therefore, the Case conflict can be discovered by applying the principle of Case assignment as soon as the object of *see* has been identified. Hence, it would seem to make sense computationally speaking (as suggested by Crocker [17]), to favour the interleaved over the non-interleaved approach of the previous chapter. Such a strategy of principle interleaving may provide an answer to the problem of controlling the combinatorial explosion due to overgeneration. In this chapter, we examine three basic questions for this alternative model:

- Is it worthwhile, in general, to interleave or not?
- Are there some principles that can be more profitably interleaved than others? If so, which ones are these?
- What is the cumulative effect of interleaving many principles?

To investigate these issues, a simple model of off-line interleaving was used. An interleaving mechanism merges definitions of linguistic principles with the LR(1)-based machine used to build phrase structure (described earlier in chapter 4). The modified LR machine will then apply principles on an incremental basis to phrase structure constituents in accordance with the order in which the LR algorithm builds structure. The interleaver makes use of a type inference mechanism to decide how best to merge principles with LR actions. The notion of the type of a principle will be introduced as an example of a computational property that can be used to help make design decisions. In particular, we will argue that the type cardinality of a principle is an important factor in determining which principles lend themselves to interleaving. We will also argue that the utility of interleaving is parasitic on the efficiency of the algorithm used to build phrase structure.

Roadmap

The remainder of the chapter is divided into four major sections:

- Section 6.1 develops the case for an off-line model of interleaving and discusses the problem of maintaining a transparent representation.
- Next, section 6.2 introduces the relevant notion of the "type of a principle" and presents a case study of how type construction proceeds on definitions of parser operations that implement conditions of θ-role and inherent Case assignment.
- Next, section 6.3 describes the principle interleaving mechanism that makes use of the type inference procedure to merge designated principles into the LR tables used by the structure building procedure.
- Finally, section 6.4 discusses how the type inference procedure performs in the current system. Here, we will also present and explain the differences in efficiency resulting from interleaving various combinations of principles.

6.1 A Model of Off-Line Interleaving

We will begin by outlining why the obvious on-line polling model is an impractical one. We will then proceed to sketch the requirements for an alternative off-line model based on type inference.

6.1.1 The Naïve Polling Model

Perhaps the obvious (and general) method for interleaved execution is to simply try to apply each principle every time some structure is built. We will call this the *naïve polling model*. Unfortunately, this is an impractical model because of the relatively high overhead of 'polling' constituents. The reason why polling is expensive comes from the general observation that many linguistic principles are fairly restrictive about the range of configurations that they apply to. In other words, many principles have the following structure: "for all constituents X such that X satisfies some property P, then some property Q *must* hold or else some addition (or modification) R *must* be made to constituent structure." For example, consider Case assignment. Roughly speaking, the principle states that: "for all configurations X, X being a configuration of Case assignment, assign Case to an appropriate noun phrase in X." However, the basic problem here is that it may take a considerable amount of computation to identify and retrieve the relevant components of a Case configuration. For example, we will need to find an element to assign Case, we will also need to find a corresponding element to receive Case, and finally, we will need to check that these two elements bear the appropriate structural relation to each other. But since these requirements are fairly restrictive, most constituents will fail at one of three points mentioned. However, this repeated evaluation of the pre-conditions will result in a high overhead above the amount of computation required to process those configurations that turn out to be Case configurations. (A more concrete example of a principle along these lines may be found in the definition of θ-role assignment to be discussed in section 6.2.)

Note however that, unless we exhaustively test each structure, we run the risk of either failing to rule out an ill-formed structure or causing a well-formed structure to be ruled out (by failing to assign Case.) In other words, non-exhaustive testing would constitute an *unsound* implementation of the (relativized) universal quantifier. Of course, if certain classes of structures that never satisfy the pre-conditions of a given principle can be identified and eliminated from consideration (on an *off-line* basis), this can lead to a substantial reduction in the number of (unsuccessful) tests necessary at *parse-time*. In the next section, we will outline an improved model that will only need to selectively poll partially-constructed phrase structures.

6.1.2 The Revised Model

Basically, the idea will be to cut the overhead by using information about the range of possible phrases that the pre-conditions of a principle may apply to. For instance, if we can determine that a certain pre-condition only applies to, say, noun, verb and adjectival phrases; then we can safely eliminate polling, say, of all adverbial phrases. Of course, the problem is how to extract this categorial information simply by inspecting the logic definition of a principle. Roughly speaking, a *type* will be a set that represents the range of category labels associated with a configuration. A type inference mechanism that operates over logic definitions will used to compute type values. Before proceeding further, let us briefly review some of the properties that such a mechanism must have to satisfy the design goals (discussed previously in chapter 2).

In the current system, an attempt has been made to abstract principle definitions away from control issues such as the question of non-interleaved versus interleaved processing. The basic idea here being that it should be possible for the grammar writer to specify principles in a manner that is both neutral and transparent with respect to such issues. In the current system, the fact that types are computed when principles are to be interleaved is completely transparent to the grammar writer. (However, the grammar writer is free to explore different control options independently of the representation.) To achieve this, a type inference mechanism must have the following properties:

1. The type inference mechanism must be *automatic* in the sense that it should compute types solely from a principle definition without having to solicit extra information from, or be volunteered by, the user, e.g. in the form of type specifications as program annotations. This also implies that the mechanism must be capable of *degrading gracefully* in cases where there is insufficient information to determine a particular type. In the current system, no annotations are allowed.

2. The *off-line* interleaving process must preserve *soundness*. For example, as discussed earlier, type computation must never cause the parser to (erroneously) fail to impose a well-formedness condition on an ill-formed structure.

3. Finally, of course, the inference mechanism must always halt. As will be discussed in a subsequent section, this is a potential problem with the type relations to be defined, given the "multiple-pass" nature of the implementation.

Although, type inference must be sound, it need not be *complete* (in a sense to be made clear below) in order to retain transparency. Suppose T is the smallest possible sound type for a configuration X, i.e. for every category c in T, there exists a constituent of category c that satisfies the conditions of X. Now, suppose a type T' computed for X by an inference algorithm is *strictly larger* than T in the sense that $T \subset T'$. Then, we say that the algorithm is incomplete (but still sound) since it has failed to determine the (minimal) type of X. To make the point clearer, we can say that the original naïve polling algorithm is equivalent to the revised system with a type inference algorithm that always returns the set of all possible category labels of the (linguistic) theory; that is, the algorithm is effectively failing to compute any meaningful type. (We will denote the set of all possible category labels, i.e. the *universal type*, by U.) The algorithm that we will describe in this paper will automatically substitute U for any part of the principle definition that a type inference fails to apply. As the section on inference rules will describe, this is usually not "fatal". In fact, this default substitution allows the inference algorithm to degrade gracefully by computing a strictly larger type but remaining sound.

6.2 Case Study: Types and θ-Role Assignment

In this section we will illustrate the use of type inference on the conditions of θ-role assignment — one of the parser operations that the θ-criterion is logically dependent on. First, we will show how θ-role assignment may be formalized in a logic program using the notation defined in chapter 3. Next, we will introduce the notion of a type of a principle, and define type inference rules that operate over primitive linguistic relations and logic programs in general . We will then apply these rules to the case of θ-role assignment, and further extend the example to include inherent Case assignment, to illustrate how the type computation is actually carried out. Finally, we will outline some of the limitations and tradeoffs in the implemented procedure.

6.2.1 The Conditions of θ-Role Assignment

θ-theory is concerned with the proper distribution of *thematic* roles. For example, consider the sentence:

(1) the police arrested John

Here, *the police* and *John* are considered to bear the thematic roles of 'agent' and 'patient', or 'theme', of the predicate *arrest*, respectively. In syntax, these relations are established by having a lexical head such as the verb *arrest* (which has an 'agent' and a 'theme' θ-role), assign its θ-roles to, i.e. *θ-mark*, the subject and object positions, respectively. Now, the fundamental principle of θ-theory is the θ-criterion, a two-part condition which ensures that θ-roles are properly discharged to *arguments*; roughly speaking, syntactic elements that have a referential function. Here is a typical definition of the θ-criterion (from Lasnik & Uriagereka [35]):

(2) θ-Criterion:

a. Every argument must be assigned a θ-role.

b. Every θ-role must be assigned to an argument.

For example, the θ-criterion rules out the following ill-formed examples:

(3) a. *the police arrested John Bill

b. *the police arrested

c. *there arrested John

d. *there arrested

In (3a), (2a) is violated since *Bill* is an argument without a θ-role. Here, the theme θ-role is assigned to *John*, and the agent role to *the police*. In (3b), no argument is available to bear the theme θ-role; and in (3c), the agent role is assigned to a non-argument, the pleonastic element *there*. In both cases, (2b) is violated. Finally, (3d) violates both sides of the θ-criterion.

The (Extended) Projection Principle requires that the θ-criterion must hold at all syntactic levels, i.e. at D-structure, S-structure and LF. For example, one consequence is that movement from a $\bar{\theta}$-position (a position to which no θ-role is assigned) to a θ-position is prohibited, as in **$John_i$ mentioned that* t_i *rains* — cf. *John mentioned that it rains* where no movement of a noun phrase has taken place.

To apply the θ-criterion, we have to first assign θ-roles. In general, *internal* θ-roles will be assigned to complement positions, e.g. the 'theme' θ-role of *arrest*, and *external* θ-roles will be assigned to subject positions, e.g. the agent θ-role of *arrest*. Following Chomsky [12], the possible cases of θ-marking are:

1. *Lexical heads θ-mark their complements.*

   We have already seen the case when the head is a verb. Nouns also θ-mark their complements. For example, the nominal form *destruction* assigns the same (internal) theme θ-role to *the city* in the configuration [[$\overline{N}$ *destruction*] [*NP of the city*]] that *destroy* assigns in [[$\overline{V}$ *destroyed*] [*NP the city*]]. Similarly, the adjective *proud* θ-marks the pronoun *he* in [$\overline{A}$ *proud* [*NP of him*]], and the preposition *to* assigns a destination θ-role in [$\overline{P}$ *to* [*NP the city*]]. Note that some heads such as the verbs *persuade* and *ask*, as in [$\overline{V}$ *persuaded* [*NP John*] [*CP that he should leave*]] and [$\overline{V}$ *asked* [*NP John*] [*CP to leave*]], may assign two or more internal roles. Others such as *sleep* take no objects and therefore have no internal roles to assign.

2. *Verbs that possess an* external *θ-role will θ-mark their subject.*

   For example, the verb *hit* will assign an 'agent' θ-role to *John* in [*IP* [*NP John*] [*VP hit Bill*]]. Note that in some cases, a verb may combine with an internal predicate to θ-mark the subject. For example, in *John is crazy*, the argument *John* is indirectly θ-marked by the adjective *crazy*.

Let X and Y be constituents, and let F_X be the feature set associated with X.
Let F be an instantiated feature.

Primitive Operations
`cat(`X`,`c`)` holds if c is the category label of X.
X `complement_of` Y holds if X occupies a complement position in phrase Y.
X `has_constituent` Y holds if Y is an immediate sub-constituent of X.
X `has_constituent_head` Y holds if Y is an immediate sub-constituent of and heads X.
X `has_constituents` L holds if L represents all the immediate sub-constituents of X.
X `has_feature` F holds if $F \in F_X$.
X `specifier_of` Y holds if X occupies the specifier position in phrase Y.

Figure 6.1 Primitive operations.

3. *Nouns that possess an external θ-role (optionally) θ-mark the subject position of the noun phrase that they head.*

   Consider the two noun phrases shown in (4) (taken from Chomsky [12]):

   (4) a. Bill's fear of John
   b. the fear of John

   In (4a), *Bill* receives same θ-role 'experiencer' that it receives in *Bill fears John*, but in (4b) the corresponding θ-role is not assigned.

(In the following discussion, we will make use of the linguistically-motivated primitive operations shown in figure 6.1.)

The configurations of θ-marking are summarized by the predicate `thetaConfig/3` shown in figure 6.2. `CF` is a θ-configuration with two components: (1) a list of the θ-roles to be assigned, and (2) a list of the constituents to receive the θ-roles. Note that the thematic representation of a head, the *θ-grid*, is encoded using a lexical feature `grid(L1,L2)`, where the lists `L1` and `L2` hold the external and internal θ-roles of the head, respectively. For example, *arrest* would have the feature `grid([agent],[theme])`.

The translation from each θ-configuration case described above to the formal definition in terms of the phrase structure primitives provided is relatively straightforward. Let us briefly review the first clause of `thetaConfig/3`. This clause is designed to extract the two components of a θ-configuration if `CF` is a configuration where a lexical head would θ-mark its complements. In general, `CF` need only be the

```
thRoleAssign in_all_configurations CF where
     thetaConfig(CF,Roles,Els) then assignRoles(Roles,Els).

thetaConfig(CF,[Role|Rs],Complements) :-% Case 1: [Xi X Complements]
     CF has_feature grid(_,[Role|Rs]),
     CF has_constituents L,
     CF has_constituent_head Head,
     pick(Head,L,Complements).          % List op: pick out Head from L
                                        % leaving Complements.
thetaConfig(IP,[Role],[NPSpec]) :-      % Case 2: [IP Spec [I1 I VP]]
     cat(IP,ip),
     VP complement_of IP,
     extRoleToAssign(VP,Role),
     NPSpec specifier_of IP.

thetaConfig(NP,Roles,Args) :-           % Case 3: [NP Spec [N1 N ..]]
     cat(NP,np),
     NP has_constituent NPSpec,
     cat(NPSpec,np),
     NP has_feature grid([Role],_),
     optionalThRole(Role,NPSpec,Roles,Args).

optionalThRole(Role,Arg,[Role],[Arg]).  % Commit to assigning the role.
optionalThRole(Role,Arg,[],[]).         % Omit assigning the role.
```

Figure 6.2 The definition of θ-role assignment.

smallest constituent containing both components.[1] This occurs when the head and its complements are both immediate constituents of `CF`. The identification of this condition is performed by the last three conjuncts of the clause. The first conjunct is there to make sure that the head has at least one θ-role to assign. For example, this would rule out [*VP* [*V sleeps*]] as a potential case of head–complement θ-marking. (Note that here we make use of the inheritance of features to examine the θ-grid associated with the head.) The definition of the other cases proceeds in

[1] It is not strictly necessary to pick the smallest configuration; in fact, the definition could be written to accept any arbitrary element containing both components. But, the reason for preferring the smallest lies in the bottom-up nature of the algorithm that builds structure. Recall that, in the interleaved model, we would like to be able to apply principles as early as possible. By defining the θ-configuration to be the smallest possible constituent, we can apply θ-role assignment as soon as the relevant components have been found.

```
% NB: assignRoles/2 will be defined in terms of assignRole/2
assignRole(Role,Argument) :- Argument has_feature theta(Role).
```

Figure 6.3 Assigning a θ-role to an argument.

a similar fashion. Finally, the operation of θ-role-assignment itself is formalized by universally quantifying over the configurations of phrase structure. The definition shown is meant to be read as follows:

> "In all configurations named by `CF` such that `CF` is a θ-configuration with components `Roles` and `Els`; then assign the θ-roles `Roles` to elements `Els`."

The actual assignment of θ-roles to constituents is carried out by `assignRoles`. Constituents will contain θ-slots that will be represented as a complex feature `theta(`S`)`, where S will hold the actual θ-role. Hence, assignment simply boils down to unifying a θ-slot with a given θ-role, as shown in figure 6.3. (Here, to keep the definition short, we only illustrate the case for a single θ-role and constituent.[2])

The issue of the high overhead involved in repeated evaluation of the preconditions of a principle can be seen clearly in this definition. The cost of determining the θ-configurations is by far the most expensive part of applying the principle. As we have seen, the conditions under which θ-marking takes place are quite restricted. Hence, a large majority of constituents that are constructed should fail to satisfy the predicate `thetaConfig`. But, given that there are three clauses to test and that several conjuncts of the clauses may succeed in the course of testing, each failure can be quite expensive. By comparison, the amount of additional work required to actually assign the θ-roles, once the appropriate components of the configuration have been determined, is small.

[2] The version of `assignRoles/2` discussed here is a little more complicated for two reasons: (1) θ-grid roles are assumed to be unordered. (More recent versions of the system adopt a revised theory in which some canonical order of complements exists.) For example, verbs such as *give* have two internal θ-roles that may permute freely in syntax, as in *gave John a book* and *gave a book to John*. (2) Also, θ-roles may have several syntactic realizations. For example, following Chomsky [12], *persuade* select a goal and a proposition as internal θ-roles. A proposition may be realized as either as a noun phrase or a clause, as in *persuaded* [*NP John*] [*CP that he should leave*] or *persuaded* [*NP John*] [*NP of the importance of leaving*]; but a goal may only be realized as a noun phrase. Note however, that although θ-roles are unordered, principles may apply to exclude ungrammatical permutations. For example, **persuaded* [*CP that he should leave*] [*NP John*] is ruled out by the Case adjacency requirement which is present in English. That is, a noun phrase must be adjacent to its Case assigner in order to receive structural Case.

6.2.2 Type Definitions and Inference Rules

In this section, we will formally define the notion of a type together with type equations for linguistic primitives and type inference rules over logic programs. In the following section, we will illustrate the process of type computation on the definition of θ-role assignment introduced in the previous section.

We begin by defining the possible category labels:

Definition 1 (Category labels)
Let L be the set of $\overline{\text{X}}$-theory head category labels, i.e. $\{N, V, A, P, I, C\}$. Let L' be the set of non-$\overline{\text{X}}$ category labels, e.g. $\{Adv, Det\}$. Now, let proj denote the $\overline{\text{X}}$ "projects to" relation. That is, let c proj $\overline{\text{c}}$ and $\overline{\text{c}}$ proj $\overline{\overline{\text{c}}}$ hold, for $c \in L$. Then, let proj* be the reflexive transitive closure of proj. Finally, let V denote the (finite) set of all possible category labels, i.e. $\{c' \mid c \text{ proj}^* c', \text{for } c \in L\} \cup L'$.

We can now define the notion of a type in terms of category labels:

Definition 2 (Type)
A *type* is simply a member of 2^V. In particular, let U, the *universal type*, be the largest member of this set, i.e. V.

In a logic program, types will be associated with logical variables that represent constituents. We will use the expression $X : T$ to represent a constituent variable X with associated type T. We will now define some of the type equations associated with linguistically-motivated primitives such as those in figure 6.1.

Definition 3 (Category labelling)
Given a constituent X and a category label c:
`cat(`X`:`T`,`c`)` holds $\Rightarrow$
$T = \{c\}$ if c is a category label, or
$T = \text{typeValue}(c)$ if c is uninstantiated.

For example, consider [Defn. 3] which defines the type of its first argument as follows:[3]

1. When c is instantiated to some label, say `NP`, as in "`cat(`X`,NP)`," we can immediately determine that X can only range over constituents with that label — for our example, we say that the type of X is $\{$`NP`$\}$.

2. On the other hand, c may be left uninstantiated: for example, we might write "`cat(`X`,`C`), head(`C`)`" to state that X must be a head. Here, the type of X

[3] Note that these type definition rules are transparent to the user. However, if the grammar writer wishes to incorporate a new primitive relation involving constituents, then a corresponding type definition should be written — although one is not strictly necessary, that is, the inference algorithm will still function, albeit, less effectively.

cannot be determined solely from "cat(X,C)." In general, if 'typeValue(C)' is returned, the inference algorithm will interpret it to mean that should make a second pass through the predicate in which cat(X,C) occurs to find other literals that involve C and compute all possible values of C where feasible.[4] In this case, head is a simple primitive of $\overline{\text{X}}$-theory that holds only for categories N, V, A, P, I, and C. Hence, the type of X will become $\{\texttt{N}, \texttt{V}, \texttt{A}, \texttt{P}, \texttt{I}, \texttt{C}\}$.

The lexicon can also play a part in type determination. Consider [Defn. 4] which applies to the feature membership primitive:

Definition 4 (Feature membership)
Given a constituent X and an instantiated feature F. Let E and F_E be a lexical entry and its feature set:
$X : T$ **has_feature** F holds $\Rightarrow$
$T = \{c \mid \exists\ E$ of category c' st. $F \in F_E, c'$ proj$^* c\}$
when F is a lexical feature, else
$T =$ typeValue(F) if F has a given type, otherwise
$T = U$.

For example, if only verbs and adjectives may be exceptional case markers (ECM), and "X has_feature ecm" holds, then we can infer (assuming feature inheritance) that X must have type (or be a sub-subset of) $\{\texttt{V}, \overline{\texttt{V}}, \texttt{VP}, \texttt{A}, \overline{\texttt{A}}, \texttt{AP}\}$. This is captured by the first type equation. The second equation applies to non-lexical features that are assigned independently of the lexicon, but have type definitions specified. Finally, the last equation introduces the universal type U. This applies as a 'catch-all' for all features not handled by the earlier type equations. (In fact, unless otherwise stated, type U will be the default type assigned whenever no type equation applies.)

Next, to conclude our examples of primitive relations, [Defn. 5] illustrates the notion of a type relation:

Definition 5 (Head relation)
Given two constituents X and Y:
$Y : T_Y$ **head_of** $X : T_X$ holds $\Rightarrow$
$T_X = \{c \mid c'$ proj$^* c$, for each $c' \in T_Y\}$,
$T_Y = \{c \mid$ head(c), c proj$^* c'$ for each $c' \in T_X\}$

head_of holds if the first constituent is the head of the second. In this case, two type equations, one for each argument, are defined in terms of the type of the other.

[4] The conditions under which typeValue(C) causes partial evaluation of a literal are too detailed to cover here. In this case, the predicate head is defined by the clauses "head(a) head(n) head(v) ...head(c)." The algorithm detects that head is defined using ground unit clauses, from which it can collect values for C without going into an infinite loop.

We now move on to examples of type composition rules for logic programs. In the following definitions, a *simple formula* F may be an atom. If F_1 and F_2 are simple formulas than so are: (`\+`F_1), (F_1`,`F_2), (F_1`;`F_2). For example, [Defn. 6] defines how types may be composed for conjunction '`,`':

Definition 6 (Conjunction) Given simple formulas F_1 and F_2:

$$\frac{X : T_1(F_1), \qquad X : T_2(F_2)}{X : T_1 \cap T_2(F_1\texttt{,}F_2)}$$

This inference rule has the following interpretation: "if X has type T_1 in F_1, and X also has type T_2 in F_2, then X has type T_1 intersect T_2 in the conjunction of F_1 and F_2." Although, strictly speaking, Horn clauses do not have disjunctions in the body of the clause, (see Lloyd [36]); it is both simple and convenient to allow the corresponding rule for disjuncts:

Definition 7 (Disjunction) Given formulas F_1 and F_2:

$$\frac{X : T_1(F_1), \qquad X : T_2(F_2)}{X : T_1 \cup T_2(F_1\texttt{;}F_2)}$$

The type rule for predicate definition is quite similar to that for disjunction, except, of course, for the variable substitutions:

Definition 8 (Predicate Definition)
Let predicate P be defined by n clauses of the form $A_i\texttt{:-}F_i, \quad 1 \leq i \leq n$, where A_i and F_i are atoms and simple formulas, respectively. Let A be an atom and let σ_i be a most general unifying substitution for A_i wrt. A. Then:

$$\frac{\forall i \text{ st. } \exists \sigma_i \text{ st. } A = A_i\sigma_i \quad X : T_i(A_i\texttt{:-}F_i)\sigma_i}{X : \bigcup_i T_i(A)}$$

Finally, we conclude the series of type definitions with a brief discussion of negation `\+` (as failure):

Definition 9 (Negation)

$$\frac{X : T(F)}{X : U(\backslash{+}F)}$$

Note that X has type U after negation, irrespective of its original type. This is a conservative rule which essentially provides no type information. Note however, that exceptions are defined for certain goals. For example,the primitive `has_feature` [Defn. 4] is an example where type information is not deleted across negation:

Definition 10 (Negation and feature membership)
Given a constituent X and an instantiated feature F. Let E and F_E be a lexical entry and its feature set:
\+ $X : T$ **has_feature** F holds $\Rightarrow$
$T = \{c|\ \exists E$ of category c' st. $F \notin F_E,\ c'$ proj$^* c\}$ if F is a lexical feature, else
$T = U - T'$ where $T' =$ typeValue(F) if F has given type, otherwise
$T = U$.

(Note: a complete list of the type rules used are given in appendix B.)

6.2.3 Type Computation

In this section, we will present an example of type computation using θ-role assignment, as defined in section 6.2.1. Let us consider again the clauses for `thetaConfig` shown previously in figure 6.2.

1. The first clause encodes the case of head–complement θ-marking. For this to occur, the head must have a θ-grid with one or more internal θ-roles. This is encoded by the first conjunct "`CF has_feature grid(_,[Role|Rs])`." The type inference algorithm will consult the type rule [Defn. 4]. According to this rule the algorithm must consult the lexicon to find all categories c' that have one (or more) lexical entries with a θ-grid containing one or more internal arguments. In the current lexicon, such entries will only exist for categories `N`, `V`, `A` and `P`. (For examples, see section 6.2.1.) Then, the type of `CF` is determined as the set of labels c such that c is zero or more (levels of) projections from c'. Hence, `CF` has type constrained by $\{\mathtt{N},\overline{\mathtt{N}},\mathtt{NP},\mathtt{V},\overline{\mathtt{V}},\mathtt{VP},\mathtt{A},\overline{\mathtt{A}},\mathtt{AP},\mathtt{P},\overline{\mathtt{P}},\mathtt{PP}\}$.

   Not all the primitives have associated type equations. For example, there is no equation for the second conjunct "`CF has_constituents L`." The list operation used in the fourth conjunct `pick` is a generic predicate that has no special linguistic significance. Hence, `CF` is given type U in both cases.

   However, the third conjunct, namely "`CF has_constituent_head Head`," specifies that `CF` immediate dominates its head; in other words, `CF` must be a single-bar category. Hence, `CF` has type upper-bounded by $\{\overline{\mathtt{N}}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}, \overline{\mathtt{I}}, \overline{\mathtt{C}}\}$.

   Using the type rule for conjuncts, we intersect each type instance to obtain $\{\overline{\mathtt{N}}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}\}$.

2. The second and third clauses encode two cases of external θ-role assignment: a noun assigning its external θ-role to its NP specifier, and the subject of IP being assigned an external θ-role, either directly or indirectly, via the VP predicate. In both cases, by applying the type rule for category labelling [Defn. 3],

```
inCaseAssign in_all_configurations CF
   where inCaseConfig(CF,Case,Items)
        then assignInherentCase(Case,Items).

inCaseConfig(CF,Case,Items) :-
      thetaConfig(CF,_,Items),
      Head head_of CF,
      cat(Head,C),
      inherentCaseAssigner(C,Case).

inherentCaseAssigner(n,gen).
inherentCaseAssigner(a,gen).
inherentCaseAssigner(p,obq).
```

Figure 6.4 A definition of inherent Case assignment.

> `CF` will be immediately restricted to a singleton set, {`NP`} in the former and {`IP`} in the latter case. Note that the algorithm will still try to explore the other conjuncts present in both clauses.[5]

Finally, using the type rule for predicate definition [Defn. 8] together with the precondition in `assignThRoles`, we can infer that the type of `thConfig` is the union of the types computed for each of the three clauses: that is, {$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, `IP`}. Note that this type is obviously much smaller than the universal type. Although not all structures that have its category labelled as one of these type components, e.g. [$\overline{V}$ [*V sleep*]] is not a θ-configuration as mentioned before, we have safely and automatically eliminated much of the unnecessary overhead incurred by the naïve polling model.

Let us extend this example one step further to illustrate the capability of the inference procedure to trace a number of different type variables. In general, evaluating type relations may require the algorithm to make multiple-passes through principle definitions. We have already seen one example of this, namely "`cat(`X`,`C`)`, `head(`C`)`", in which the computation of the type of X will require the computation of the possible values for the type value C. In the current implementation, this is done by making a separate pass for each such unknown. A more sophisticated example can be found in the fragment shown in figure 6.4. This example is taken from

[5] Incidentally, if the empty type for a definition is ever returned, then that definition can never succeed. Although, we have argued for the transparency of the type inference procedure, this is perhaps one case in which debugging information can and should be conveyed to the grammar writer.

the definition of inherent Case assignment, an assignment operation that operates on a proper subset of θ-configurations. The additional restriction here being that the head must be an inherent Case assigner.

By using the type rule for the `head_of` primitive [Defn. 5], the algorithm can infer that the type of `CF` is related to the type of `Head`. Therefore, by switching to computing the type of `Head`, the algorithm may be able impose further constraints on the type of `CF`. By using rule [Defn. 3] on the atom "`cat(Head,C)`," we can infer that the type of `Head` is given by the set of possible values for `C`. Switching once more to compute the values of `C`, we can follow the definition of `inherentCaseAssigner` to obtain $\mathtt{C} = \{\mathtt{N}, \mathtt{A}, \mathtt{P}\}$. By plugging these values back into the type equation for `head_of`, `CF` is constrained to be $\{\mathtt{N}, \overline{\mathtt{N}}, \mathtt{NP}, \mathtt{A}, \overline{\mathtt{A}}, \mathtt{AP}, \mathtt{P}, \overline{\mathtt{P}}, \mathtt{PP}\}$. Finally, we know that `CF` has type $\{\overline{\mathtt{N}}, \mathtt{NP}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}, \mathtt{IP}\}$ in `thetaConfig`, so by type intersection, we have that inherent Case assignment has type $\{\overline{\mathtt{N}}, \mathtt{NP}, \overline{\mathtt{A}}, \overline{\mathtt{P}}\}$.

Although, type relations allow the algorithm to infer the type of one variable in terms of another, as illustrated by in the preceding example; unfortunately, this also allows *circularity* in a chain of type inferences to occur. Perhaps, the simplest example occurs in the definition of `head_of` where the type of the first argument is defined in terms of the second, and vice-versa. Currently, the adopted solution is to keep track of the names of the variables being traced, and to terminate the search as soon as a previously encountered variable has been detected. Then, type computation will be resumed with the universal type substituted for all outstanding type variables. This system can be extended to cope with looping caused by recursive predicates by treating a recursive call as if the system switched to computing a new type variable. For example, this will prevent the algorithm from looping on `p(...,X,...) :- p(...,X,...)`, where X is the constituent variable being traced. This rudimentary loop detection scheme seems to be sufficient to guarantee termination, but at the cost of generality.[6]

This completes our description of the basic type inference procedure. We will return to discuss further extensions in a later section. The procedure has been successfully tested on all of the (relevant) principles found in the current system. (The pertinent results are summarized in section 6.4.) Next, we will turn our attention to the description of the mechanism (built on top of type inference) that transparently compiles principle definitions into the appropriate forms for interleaved execution.

6.3 The Interleaver

[6]It is easy to construct an example to show the limitations of the loop detection scheme. For example, consider the following (artificial) definition of a principle p: `p(0,X) :- cat(X,np).` `p(1,X) :- p(0,X).` The loop detection algorithm will prematurely abort the type computation process in the case of the goal `p(1,`X`)`.

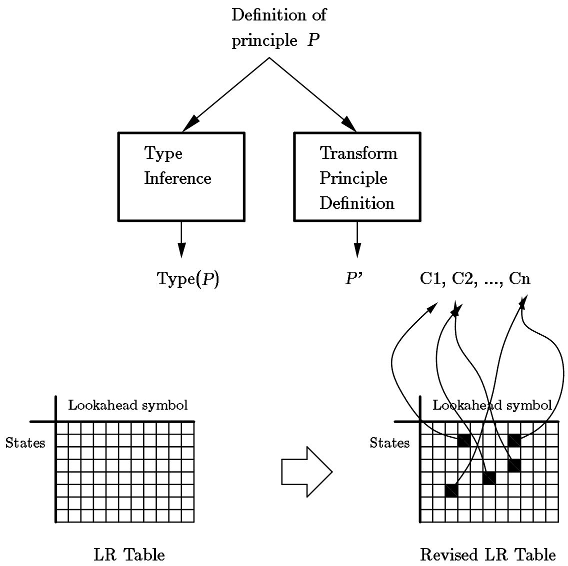


Figure 6.5 The interleaving process.

The job of merging principle definitions with the LR tables used by phrase structure recovery procedure is carried out by the off-line interleaver. (See chapter 4 for an overview of the LR parsing model.) Given the definitions of one or more principles, it uses the type inference mechanism to deduce the appropriate points in the LR tables where the parser operations corresponding to the (interleaved) principles should be called. Figure 6.5 illustrates this process schematically. For example, as we saw in the last section, θ-role assignment has type $\{\overline{\mathtt{N}}, \mathtt{NP}, \overline{\mathtt{V}}, \overline{\mathtt{A}}, \overline{\mathtt{P}}, \mathtt{IP}\}$. This means that θ-role assignment need only be consulted when the phrase structure recovery procedure completes a constituent with a category label belonging to the type. In this section, we will describe how principle definitions are transformed into a form that may be directly called by LR table actions. We will also discuss the problem of respecting logical dependencies in the case where more than one principle is interleaved.

6.3.1 Compiling Principle Definitions

Adding external calls to LR action entries is a standard technique used to implement side effects or "semantic actions" in compiler construction. In our case, this is implemented by having all reduce actions call a "hook" predicate, and passing along the structure it has just built. There is a separate hook predicate for each non-terminal symbol. For example, all reduce actions for rules matching $\mathtt{NP} \rightarrow \alpha$ will call a predicate `reduceNP/1` (passing along an NP constituent as input). In general, if a principle P has type T, then for each category label c belonging to T, there will be a goal P_c`(X)` in the body of the clause `reduce`c`(X) :-` β, where $P_c/1$ will be a specialized version of the definition for P (with respect to c). In this model, the task of the off-line interleaver is to recast principle definitions (written initially using one of the two forms, `in_all_configurations` or `compositional_cases_on`, as described in chapter 3), into a series of predicates specialized for each category given by the type of the principle. Here, we will describe the procedure in detail for the `in_all_configurations` form only.

Compiling the `in_all_configurations` form

Principles defined using `in_all_configurations` have the following general form:

```
<Name> in_all_configurations <CF>
        where <Pre-Cond-1> then <Post-Cond-1>
        else <Pre-Cond-2> then <Post-Cond-1>
        ...
        else <Pre-Cond-n> then <Post-Cond-n>
```

The type of the principle will be the type of the logical variable `<CF>`. This will be the union of the type of `<CF>` in each pre-condition. For each category c, we can determine if c belongs to the type of each pre-condition and produce a version of the general form specialized with respect to c. For example, if c belonged to the type of `<CF>` for `<Pre-Cond-1>` and `<Pre-Cond-n>` only, then we can construct the following single clause predicate:

```
<Namec>(<CF>) :-
       <Pre-Cond-1>
       -> <Post-Cond-1>
       ;  <Pre-Cond-n>
       -> <Post-Cond-n>
       ;  true
```

(Note: we have used standard PROLOG notation for "if–then–else" constructions. Also, assume that the goal `true` always succeeds.)

Here, type inference also eliminates unnecessary run-time testing (with certain failure) for `<Pre-Cond-2>` through `<Pre-Cond-(n-1)>`. In general, there may be n different specialized predicates generated, where n is the cardinality of the type of the principle. This compilation procedure is summarized in figure 6.6. There are two aspects of the procedure worth noting:

1. A preliminary expansion stage (Step 1) is used to further eliminate unnecessary testing in some cases. For example, consider again the definition of θ-role

Given a principle definition of the form:

```
:- <Name> in_all_configurations <Variable>
        where <Pre-conditions-1> then <Conditions-1>
        else <Pre-conditions-2> then <Conditions-2>...
```

with `<Variable>` having type T. Then:

1. **Preliminary Unfold Step**: If the definitions of any pre-condition has multiple clauses, then expand that pre-condition by adding an extra condition for every clause.
2. For each category label c in T do the following:
3. Let sequence S_c be the empty sequence $\langle\rangle$.
4. **Collect Compatible Conditions**: For each condition "P `then` C" in the definition:
   (a) Let T_P be the type of `<Variable>` wrt. P.
   (b) If $c \in T_P$ then c is compatible with "P `then` C". If so, append $\langle(P, C)\rangle$ to S_c.
5. **Construct Predicate**: Combine the conditions in S_c into a nested form F according to the following rules:

   $$\langle\rangle \longmapsto \texttt{true}.$$
   $$\langle(P_1, C_1), S_1\rangle \longmapsto (P_1 \texttt{ -> } C_1 \; ; F_1)$$
   $$\text{if sub-sequence } S_1 \longmapsto \text{form } F_1.$$

   If F has been constructed before, do nothing. Otherwise, generate a new predicate symbol p and emit p(`<Variable>`) `:-` F.

Figure 6.6 Compiling the `in_all_configurations` form.

assignment from figure 6.2. Note that `thetaConfig/3` is defined using three clauses. As we saw in section 6.2.3, the type of `CF` ({$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, `IP`}) was determined to be the union of {$\overline{\texttt{N}}$, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$}, {`NP`} and {`IP`} — the corresponding types for each clause. By unfolding the definition of `thetaConfig/3` first, we can apply the same optimization trick. Because the clauses are independent (i.e. the types do not overlap), the result will be the following three specialized predicates:

```
thRoleAssign1(CF) :-
        thetaConfig1(CF,Roles,Els)
        -> assignRoles(Roles,Els)
        ;  true.
thRoleAssign2(CF) :-
        thetaConfig2(CF,Roles,Els)
        -> assignRoles(Roles,Els)
        ;  true.
thRoleAssign3(CF) :-
        thetaConfig3(CF,Roles,Els)
        -> assignRoles(Roles,Els)
        ;  true.
```

where each `thetaConfig`*i*`/3` is a predicate with just one clause; namely, the *i*th clause of the original `thetaConfig/3` predicate.

2. Specialized predicates are shared between type members that generate the same sequence of pre-conditions (Step 5).

Compiling the `compositional_cases_on` form

The corresponding compilation procedure for compositionally defined principles has the same structure as the `in_all_configurations` form. In this situation, the conditions to be considered are simply all the base cases plus the single compositional case. (Because of the similarity to the previous form, we omit the detailed description of the procedure.) However, one important difference is that compositional definitions compute intermediate values for each phrase. For example, condition A of the Binding Theory (as described in chapter 3) maintains a set of unbound anaphors for each (sub-) phrase. Since the value for an aggregate phrase is computed as a combination of the values of its sub-constituents, the value computed for each constituent must be preserved for subsequent use. Hence, the expanded forms will contain extra goals for memoization.[7]

[7] The internally maintained values for each constituent are stored as features within the constituent itself.

6.3.2 Respecting Logical Dependencies

So far we have assumed a simple model of processing where interleaved principles are applied to phrases as soon as they are recovered. This "eager" model breaks down in the case where an interleaved principle depends on information that may not be available at the time it is first called. For example, consider the inherent Case assignment operation. Now, the system recovers phrase structure at the level of S-structure, but inherent Case is assigned at D-structure. A D-structure constituent recovery operation (discussed previously in section 4.3.2) is used to determine the appropriate constituent that occupies any given position at D-structure. This operation depends, of course, on the prior assignment of movement chains (by the Trace theory operation). Hence, the inherent Case assignment operation must wait on chain assignment.

As another example, consider the Case filter operation. Inherent and structural Case assignment have type {$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{A}}$,$\overline{\texttt{P}}$} and {`NP`, `VP`, `AP`, `PP`, `IP`, `CP`}, respectively. But, the Case Filter (which is logically dependent on both operations) has type {`NP`}. The interleaver cannot directly link the Case Filter to reduce `NP` $\rightarrow \alpha$ actions because Case may not have been assigned before the filter is invoked. For example, [$_{\bar{V}}$ [$_{V}$ saw] [$_{NP}$ Mary]] is a configuration of structural Case assignment, but the Case filter must be delayed for *Mary* until the $\overline{\mathrm{V}}$ constituent has been identified (and Case assigned).

The interleaver automatically reformulates principle definitions with such dependencies to use a simple constraint mechanism.[8] Constraints are associated with individual constituents — generally, those with category labels from the type of the principle. An interleaved principle may cause a constraint (i.e. an unevaluated goal) to be "posted" on any completed constituent. Constraints are posted with respect to a arbitrary signal. When another principle posts a signal S to a constituent with constraint C on signal S, it will trigger the execution of C. Of course, constraints and signal may also be posted in reverse order; that is, if constraint C on S is posted on a constituent that has had signal S raised, C will not be delayed, but immediately executed.

Now, returning to the previous two examples:

- The inherent Case assignment operation will post a constraint (i.e. delay the execution of) `assignInherentCase(`$Case$`,`X`)` for signal `chain` on the S-structure NP X at the position where inherent Case is to be assigned (at D-structure). When a chain involving X is completed (by the Trace theory operation), the signal `chain` will go out to all elements of the chain. Because computing the chain for X allows the D-structure constituent recovery proce-

[8] However, as mentioned in the previous chapter, the dependency relations are not yet automatically generated.

dure to determine the corresponding constituent at D-structure, the delayed goal `assignInherentCase(`$Case$`,`X`)` may now be executed. (Also, inherent Case assignment will, in turn, raise the signal `case` (for "Case assigned") on successful completion of the delayed goal.)

- In the second example, because the Case filter has type {`NP`}, the constraint `assignedCase(`X`)` on signal `case` will be attached to each NP X for which `lexicalNP(`X`)` holds. When X is assigned (either inherent or structural) Case, the delayed goal which completes the Case filter operation for X will be triggered.

To summarize, a given constituent may have a number of posted constraints on various signals.[9] A simple signal applied to a constituent may cause the evaluation of a single constraint or trigger a "cascade" of evaluations. Any untriggered constraints left over will be automatically triggered after the complete structure has been built. For example, *John* (which must satisfy `assignedCase(`*John*`)`) cannot be assigned Case by any element in *I am eager John to win.* Operations (e.g. structural Case assignment) which have no logical dependencies (other than the phrase structure recovery procedure) do not make use of the constraint mechanism, but are interleaved directly.[10] Finally, to complete our description, we will briefly mention two noteworthy features of this mechanism:

1. *Selective polling is maintained.*

   In contrast to schemes that satisfy constraints by propagating constraints posted at a particular node up to dominating nodes (and attempt to merge compatible constraints from sub-nodes whenever possible at each interior node), the present scheme only attempts to test a posted constraint when the necessary information (e.g. chains or case) has been computed.

2. *Constraint mechanism used only when required.*

   Since no general meta-interpretation is needed to implement this scheme, unnecessary overhead will be avoided for principles that can be directly interleaved.

[9] Operations that have multiple immediate dependencies are handled using the same scheme. For example, the Trace Case condition operation (which ensures that *wh*-traces, but not NP-traces, get Case) depends directly on both Case assignment (signal `case`) and chain formation (signal `chain`). The operation will post *two* mutually dependent constraints of the form `(signalled(`S_1`,`X`) ->` $applyCondition(X)$ `; true)` on signal S_2, where S_1 and S_2 are instantiated to `chain` and `case` (in turn). The `signalled(`S_1`,`X`)` goal ensures that the condition is not applied unless S_1 has also been signalled.

[10] However, directly interleaved operations may signal other operations. For example, the structural Case assignment operations signals `case` on each NP that it assigns Case to.

1. *EC Reduce for $V \rightarrow \lambda$.*

   Builds an empty category with label V.

2. *Adjunction Reduce for $V_i \rightarrow \alpha V_i \beta$.*

   Builds a larger category V_i with identical category label and features to V_i on the right side of the rule.

3. *Normal Reduce for $V^{j+1} \rightarrow \alpha V^j \beta$.*

   Projects V^j up to the next bar-level.

Figure 6.7 The reduce actions of the LR machine.

```
thetaConfig(NP,RoleList,Arguments) :- % [NP Spec [N1 N ..]]
     cat(NP,np),
     NP has_constituent SpecOfNP,
     cat(SpecOfNP,np),
     NP has_feature grid([Role],_),
     optionalThetaRole(Role,SpecOfNP,RoleList,Arguments).
```

Figure 6.8 A case of θ-configuration.

6.3.3 Restrictions: An Extension to Type Inference

We have seen how type information allows the interleaver to *safely* restrict the categories of constituents that need to be polled for a given interleaved principle. As described earlier in section 6.3.1, each element V of a type maps onto the reduce actions in the extended-LR table for rules of the form $V \rightarrow \alpha$. By further distinguishing the various kinds of reduce actions, it is possible to implement a more "finely-grained" partitioning of the underlying LR table, and thus further improve the selectivity of the interleaving mechanism. In this section, we will introduce the notion of a *restriction set* that may contain elements of the form `ec` (i.e. restricted to empty categories (ECs) only), `adjS` (i.e. restricted to adjunction structures), `head` (i.e. must be at the level of a head projection), and $\neg R$ (i.e. must not be R) — where R is one of the primitive restrictions. In general, a principle will have an associated restriction set in the same way that it has a given "type".

In the current system, the extended-LR machine distinguishes three different kinds of reduce actions, as shown in figure 6.7.[11] That is, each rule in the S-structure grammar describes either an empty category, an adjunction structure, or simply one level of $\overline{\mathrm{X}}$-projection. In an analogous manner to that for type inference, if the system can deduce from the definition of a principle that the principle never applies to, say, empty categories (i.e. have restriction set $\{\neg\texttt{ec}\}$), then the interleaver can eliminate useless polling of all empty categories (irrespective of category label). Consider again the definition of θ-role-assignment — in particular, the second case (repeated in figure 6.8). This clause covers the case of a noun (optionally) assigning its external θ-role to its NP specifier. Since the type of this configuration is $\{\texttt{NP}\}$, only reduce NP actions need call this configuration. However, note that all phrases forwarded by the EC Reduce action for NP can never satisfy this configuration. For instance, in order for the second goal (`NP has_constituent SpecOfNP`) to hold, the constituent represented by the variable `NP` cannot be empty. In the current system, the restriction to non-empty categories is encoded by associating a set $\{\neg\texttt{ec}\}$ with the first argument of the primitive operation `has_constituent`. This restriction suffice to prevent this instance of θ-role-assignment from being called every time the parser hypothesizes an empty NP.

In general, restriction sets will be defined for the arguments of linguistic primitives — just as for regular types. Similarly, propagation rules can be defined for the usual logical connectives. (We will omit the description of the corresponding restriction rules in this chapter. The interested reader is referred to the merged (type and restriction) definitions given in appendix B.) The only interesting difference occurs for conjunction (and disjunction) where restrictions are combined using set union (resp. intersection), rather than using intersection (resp. union) as in the case of types. Because of the similarity between types and restrictions, the same inference procedure is used for both.

6.4 Results and Conclusions

In this section, we will discuss the performance of the type inference mechanism on the principles that apply at S-structure. We will also correlate type cardinality with parsing times. We will explain the correlation by reference to the interaction between the degree of error detection in the phrase structure recovery procedure and principle interleaving. More precisely, we will conclude that the effectiveness of interleaving is dependent on the amount of "garden-pathing" done by the LR(1)-based phrase structure recovery mechanism.

[11] Actually, as described in chapter 4, the system distinguishes other types of reduce actions for dummy and vacuous categories which do not build structure, and thus will not be relevant here.

Parser operation P	*Type(P)*	$\lvert$*Type(P)*$\rvert$	*Restrictions*
Theta-role assignment	{$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, `IP`}	6	{¬`ec`}
Theta criterion	{`NP`, `CP`}	2	{}
D-structure theta condition	{$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{V}}$, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$, `IP`}	6	{¬`ec`}
Inherent Case assignment	{$\overline{\texttt{N}}$, `NP`, $\overline{\texttt{A}}$, $\overline{\texttt{P}}$}	4	{¬`ec`}
Structural Case assignment	{*NP*, `VP`, *PP*, *AP*, `IP`, `CP`}	6	{¬`ec`}
$\overline{\text{S}}$-*Deletion*	{`VP`, `AP`}	2	{¬`AdjS`, ¬`ec`}
Case filter	{`NP`}	1	{¬`ec`}
Trace Case condition	{`NP`}	1	{`ec`}
Wh-*in-syntax*	{`CP`}	1	{}
Coindex subject and Infl	{`IP`}	1	{¬`ec`}
License syntactic adjuncts	{`VP`, `NP`}	2	{`adjS`}
Trace theory	{`NP`, `Adv`} ∪ U	20	{}
Subjacency	{`NP`} ∪ U	20	{}
Free indexation	{`NP`} ∪ U	20	{}
Functional determination	{`NP`} ∪ U	20	{`ec`}
Binding condition A	{`NP`} ∪ U	20	{}
Binding condition B	{`NP`} ∪ U	20	{}
Binding condition C	{`NP`} ∪ U	20	{}
Control	{`NP`} ∪ U	20	{`ec`}
ECP	U	20	{`ec`}

Figure 6.9 Computed types for parser operations at S-structure.

6.4.1 Type Computation Results

Figure 6.9 illustrates the type, the associated type cardinality and restriction set computed by the inference procedure for each principle. The procedure is powerful enough to come up with the minimal type for almost all of the twenty or so parser operations that operate at S-structure in the current system.[12] Compositionally defined principles will incrementally compute values for each constituent of phrase structure, and therefore, will have the universal type U.

We conclude that automatic type inference is practical in the context of the current system because the principle definitions make considerable use of the linguistic primitives for which type definitions exist. In general, we can probably rely on the grammar writer to make use of the supplied primitives. Hence, the type compu-

[12]Due to an (artificial) limitation of the multi-pass algorithm, the procedure fails to take into account the set of possible structural Case assigners, thereby failing to compute the minimal type for structural Case assignment (NP, PP and AP being extraneous values). Nevertheless, soundness is preserved. Note also that (unlike D-structure operations) no LF operations can be interleaved in the current system.

tation process usually has access to enough type information so that the universal type may be avoided, despite the conservative handling of negation and recursive definitions. Moreover, since the language used to represent the grammar is a fairly simple one, we have avoided many complexities for type inference present in general programming languages.[13] These two factors conspire to make the type computation problem simple enough to be practical for use in principle-based systems.

6.4.2 Principle Interleaving and Parsing Efficiency

Given the interleaving model described in the previous section, how well does principle interleaving perform in practice? Figure 6.10 illustrates the typical effect of interleaving on parsing performance for a series of six parsers. Each parser (numbered $i + 1$) builds on the preceding parser (numbered i) by adding one or two interleaved operations. (The actual parser specifications are shown separately in figure 6.11.) The fifteen test sentences are the same examples taken from Lasnik & Uriagereka that were used to test different principle orderings in chapter 5 (see figure 5.2). As the graph clearly shows, the general trend is that parsing times will increase dramatically as more principles are interleaved. Observe also that:

- There is a large jump between the second and third configurations (`DirectOnly` and `Chains`).[14] The crucial difference between the two is that the operation responsible for chain formation (Trace theory), one of the major generators in the system, has been interleaved in the third configuration.
- Adding filters such as the Case filter and the Trace Case condition (e.g. configuration `CaseTheory`) does *not* result in a reduction in parsing time.
- Although it is difficult to tell from the scale of the graph, the second configuration (`DirectOnly`) almost always performs better than the original, non-interleaved configuration (`Default`) — on the average about 30% faster.

Explaining the results

Why should there be such a difference between interleaving and non-interleaving? After all, in both cases, the parser will make use of the exact same definitions (i.e. execute the same code) for each principle and return the same linguistic judgments. The only possible difference occurs in the underlying mechanism (transparent to the user) used to "hook up" the well-formedness conditions to phrase structure.

[13] For example, there is no data type polymorphism or coercion. (See Milner [39].)

[14] Note also that the results for some sentences (i.e. (1)–(4)) are largely unaffected by the amount of interleaving. These examples are relatively simple sentences (i.e. containing no embedded clauses) with correspondingly simple phrase structure at S-structure.

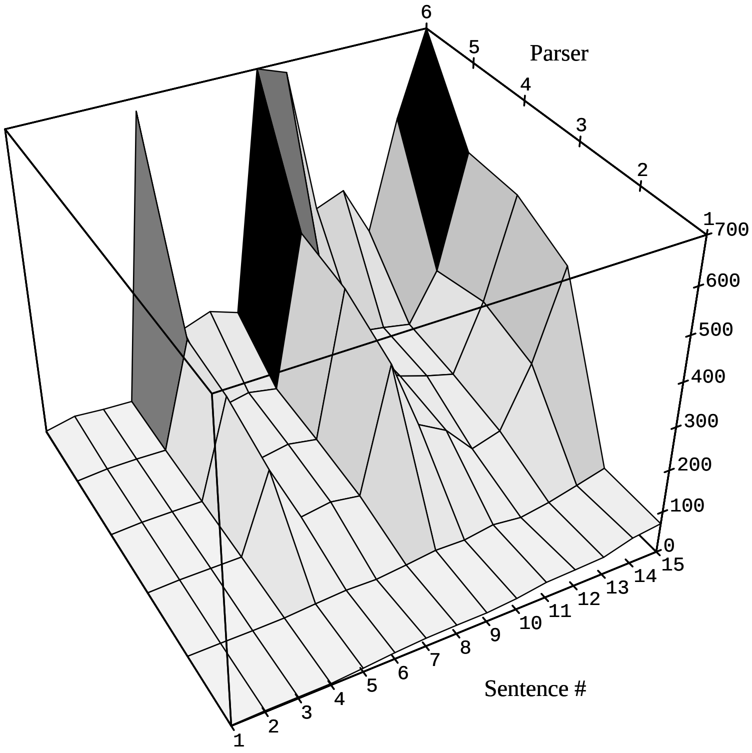


Figure 6.10 The effect of interleaving.

Parser Configuration	*Operations Interleaved*
1. `Default`	None
2. `DirectOnly`	*License syntactic adjuncts* + $\overline{\mathrm{S}}$-*Deletion* + *Coindex subject & Infl*
3. `Chains`	= `DirectOnly` + *Trace theory* + *Inherent Case assignment*
4. `CaseAssignment`	= `Chains` + *Structural Case assignment*
5. `CaseFilter`	= `CaseAssignment` + *Case filter*
6. `CaseTheory`	= `CaseFilter` + *Trace Case condition*

Figure 6.11 Test parser configurations.

Let us briefly review both mechanisms in the context of a typical condition "`q(X) if p(X)`":

1. *Non-interleaved processing*: A "tree-walker" is used to explore every node `X` in a structure, and apply `q(X)` to each node satisfying `p(X)`.

2. *Interleaved processing*: No tree-walker is used. Instead, `p(X)` is called directly for each node with category label belonging to the type of `X` (with respect to `p`). For each node `X` satisfying `p`, `q(X)` may be evaluated immediately or delayed pending some signal.

Note that at first glance, it seems that the second mechanism should have less overhead.[15] A number of efforts have been made to keep the overhead of interleaving as low as possible. The use of type inference cuts down on the number of times `p` has to be called (compared with tree-walking). Also, the constraint mechanism is relatively cheap since no polling is used.

The source of the large overhead lies in the *total* number of times `p(X)` (and `q(X)`) are called during parsing. Let us now review how the operation that builds phrase structure works. Consider the search space diagram shown in figure 6.12. Basically, the parsing mechanism will proceed from LR state to LR state, reducing the current sentential string until it either succeeds in replacing it with a single

[15] One caveat: other details (e.g. `call(G)` *vs.* `G`) of the actual PROLOG implementation will also have some effect. But these overheads are relatively minor compared to the cost of `G` itself.

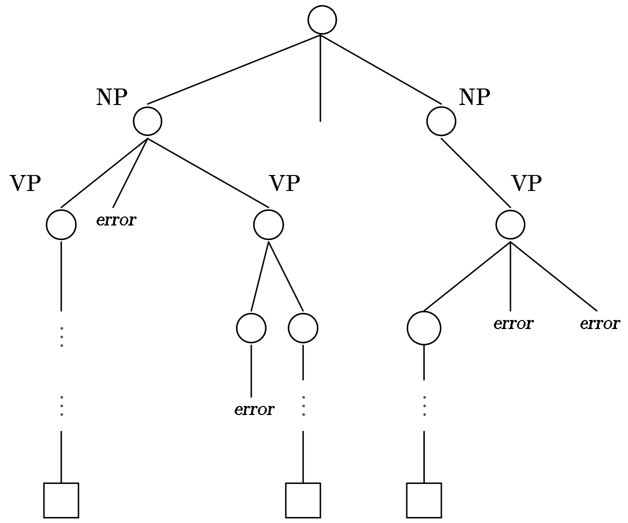


Figure 6.12 Search and error detection.

sentential symbol, *or* it deduces (either with or without the one symbol lookahead) that there can be no derivation from the sentential symbol to the current sentential string. At each step, the parser can shift an input symbol or perform a reduce action, i.e. complete a non-terminal (e.g. an NP or VP). Because the S-structure rules do not form an LR(1) grammar, the parser cannot always determine the appropriate action to take at all times. Hence, it will end up trying several different actions for certain state/lookahead combinations — some of which will not lead to successful derivations. (See chapter 4 for a discussion of LR(1) conflicts for various S-structure phrase structure grammars.) Only at a later stage in the derivation, when the parser has read enough of the input to determine that the current path is incorrect (i.e. the parser has *garden-pathed*), will it automatically retrace its steps to explore alternative paths. (In the current system, the default behaviour is to find all possible parses — so the parser will explore the entire search space.) The crucial difference with respect to interleaving and non-interleaving is that, in the former case, any extra conditions (representing interleaved principles) that are attached to reduce actions will have been tried. If this occurs in a state on a garden path, then extra (wasted) work has been performed in comparison to a non-interleaved configuration. In other words, interleaving poses a large cost penalty for visiting states that do not lead to successful derivations (for the current input). In general, the more principles are interleaved, the larger the associated penalty. Although the addition of off-line type inference keeps the number of actions with attached conditions low, the experiments show that even in this case the penalty overwhelms the advantage of using principle violations to prune the search space, especially as the number of principles interleaved increases. Let us now re-examine the interleaving results:

1. As reported earlier, the second parser (`DirectOnly`) is about 30% faster on average than non-interleaving. In this case, only three operations of low type cardinality have been interleaved. (See figure 6.9 for type information.) Apparently, the penalty here is low enough to make interleaving cost effective.

2. The third parser (`Chains`) adds Trace theory which dramatically slows down the parser for two reasons: (1) it is compositionally defined (and therefore has a large type), thus adding a penalty for every phrase built, and more significantly: (2) it is a major generator, thus multiplying the branching factor in the search space.

3. Interleaving structural Case assignment (`CaseAssignment`), since it is not a filter (except in Case clash situations), results in a slightly slower parser. Next, adding the Case filter does not result in a slow down; in fact, it speeds up the parser slightly in a few cases. This suggests that the cost/penalty ratio for the Case filter is roughly even. Finally, adding a Case condition for *wh*- and NP traces results in another jump in parsing times. Note that this operation has the same type cardinality as the Case filter. The reason for the discrepancy is that the parser predicts many more empty NPs than overt NPs. In general, NPs can appear in a large number of different positions in phrase structure. Hence, a large number of empty NPs will be hypothesized during parsing. However, the need to first shift a noun in the input prevents the parser from doing the same with overt NPs. Note also that the use of restrictions means that the two filters are attached to different reduce actions.[16]

Prospects for principle interleaving

In this chapter, we have discussed and explained why it is generally not cost effective to interleave principles, even in this system where an attempt has been made to minimize overhead costs. However, experience with the current system indicates that it *is* cost effective to selectively interleave those principles that do not depend on, and thus do not require the prior interleaving of, generators. To make interleaving a considerably more attractive model, it seems that a large reduction in the amount of garden-pathing during phrase structure construction is necessary. The

[16] Unlike the condition on traces, the Case filter is not attached to *EC Reduce* (see figure 6.7). In fact, since the trace condition is only attached to *EC Reduce*, the two filters share no common attachment.

fact that the current system has employed a fairly efficient canonical LR(1)-based algorithm extended to include unbounded lookahead to further improve error detection suggests that the prospects for such an improvement is not good. As a final note, we should add that interleaving may be more effective in a parser that minimizes search by only looking for a single parse.

Chapter 7

Conclusions

This chapter discusses whether the current system meets its original goals and outlines possible directions for future work.

7.1 Remaining Problems and Limitations

In the preceding chapters, we have described how the parsing system represents linguistic principles and compiles those descriptions into forms that can be more efficiently processed. We have also described the (sometimes surprising) effects and tradeoffs from varying simple parameters of control. At this point, it is appropriate to review how the system measures up to the original goals and point out limitations and difficulties that were encountered.

- *Linguistic coverage.*

  The current set of principles will correctly account for almost all of the examples from the first four chapters in Lasnik & Uriagereka— plus a small set of additional examples taken from *Knowledge of Language* (Chomsky [12]) and the Japanese *wh*-questions from Lasnik & Saito [34]. This can probably be considered to represent broad coverage with respect to existing principle-based systems. It should be pointed out here that the problem of assembling a mutually compatible set of twenty-five or so principles turned out to be the most expensive portion of the project, both with respect to time and computational resources. The degree of interaction between the various principles and the need to import principles from sources other than Lasnik & Uriagereka were the major contributing reasons. In particular, the almost *chaotic* nature of the system (in the sense of being sensitive to perturbations in the definitions of principles) made it very difficult to maintain a monotonically increasing set of correctly analyzed examples.[1] Hence, substantial additions or revisions to the current theory will require much careful planning. Actually, the small size of the lexicon is probably the most significant barrier to having increased coverage. Currently, the lexicon contains exactly those entries required to handle the test cases and no more.

- *Transparency of representation.*

  One of the major goals for the representation was to try to minimize the cost of formalizing linguistic principles by pitching the level of definitions as closely as possible to the notation used by linguists whilst still being able to parse. Independence from control parameters and keeping as much of the underlying machinery as possible (e.g. the LR machine and type inference mechanism) hidden from the grammar writer were also important goals. Whilst it is not claimed that someone completely unfamiliar with the underlying system could successfully encode and debug principles, it is claimed that the majority of the definitions should be readily accessible to linguists (i.e. the semantic import of the definitions should be immediately obvious without knowing how the definitions are executed). Probably the most obvious defects in this area are the following:

  1. The grammar writer is insufficiently insulated from the default search strategy used by PROLOG. In general, one will have to worry about the order in which goals are processed, particularly when negation (as failure) is involved.

  2. The process of turning principle definitions into a parser has not been fully automated. In particular, the grammar writer will have to manually identify any logical dependencies with respect to the existing base of principles. In general, this is necessary for both the simple principle ordering and interleaving models. Moreover, structural cues for dynamic ordering are not automatically identified either. However, in either case there seems to be no foreseeable reason why the process cannot be made automatic.

  3. Finally, as discussed in chapter 3, problems remain with principles that are not stated in a form that can be directly used for parsing. Free indexation (*"assign indices freely"*) is a prime example of this. Currently, the system has no knowledge of how to go about computing the possible indices in a sensible manner.

[1] Note that this same property can be considered to be a virtue in linguistic theory.

Nevertheless, it should be mentioned that the relative ease of formalizing principles and the independence of the resulting definitions from the underlying machinery played a large role in keeping the cost of debugging the system low.

- *Parsing efficiency.*

  The modified LR machine, the conjunct (i.e. principle) ordering mechanism and the type inference procedure are all examples of well-known methods that together make it possible to obtain results in a reasonable amount of time. Even with these optimizations, the system can still take anything from around a hundredth of a second to about five minutes to search for all possible parses on a single example from Lasnik & Uriagereka.[2] Moreover, it seems that the amount of work required in some cases can be relatively unpredictable (i.e. governed by theory-internal artifacts rather than, say, sentence length or similarity of input). These figures are reasonable for use as a research tool, but not for applications such as text retrieval or understanding. Clearly, much more work is required before the current system can be deemed truly practical.

7.2 Future Work

Future extensions to the system will be in two main directions: (1) towards a tool to aid in testing linguistic theories, and (2) towards a practical system by improving computational efficiency. Also, expanding the coverage and sophistication of the basic system is another obvious place to start. The extension to handle the Japanese *wh*-questions from Lasnik & Saito with only a small modification in existing principles is a recent addition to the system. Further developments along these lines, e.g. to handle phenomena such as Japanese passive constructions at the same time as moving to VP-internal subjects, are also planned. Note that the current system implements just one of many possible theories in the principles-and-parameters framework. Supporting other theories might help to raise the level of abstraction or point out inadequacies in the current representation language. One obvious starting point would be to replace specific sub-theories of grammar. For instance, chapter 5 of Lasnik & Uriagereka discusses alternatives to the implemented Binding theory that seem largely compatible with other existing sub-theories; namely, Aoun's Generalized Binding theory (which also replaces the ECP) and Higginbotham's Linking theory.

[2]Times mentioned are for a Quintus PROLOG/sparcstation platform. The median time taken for configuration `Default` (introduced in chapter 5) with respect to all the examples in the first four chapters of Lasnik & Uriagereka is 4.38 (s).

A test tool

Ideally, the system should be usable as a tool to help verify or develop linguistic theories, i.e. as the linguist's equivalent of a "hand calculator" that saves the linguist, say, from having to manually verify that some particular structure does not violate some unforeseen principle, or that there is no unlikely structure or derivation sequence that will not be blocked. The availability of a relatively fast and exhaustive (constructive) "proof checker" may also encourage the linguist to explore the ramifications of subtle perturbations in the theory at a relatively low cost. However, the current system requires further development to support debugging at a level that is linguistically-relevant. For example, the current system supports tracing of control and candidate structures down to the level of individual principles, but not within. To find out what is happening within a principle (e.g. if a particular trace is being lexically or antecedent governed in the ECP), one has to drop down into the underlying PROLOG system. Similarly, if there is a problem with the phrase structure recovery procedure (e.g. parser attempts to construct infinite structures), debugging must be performed at the level of LR(1) state sets, rather than $\overline{\mathrm{X}}$-rules. Clearly, it is unreasonable and a violation of transparency to expect the grammar writer to perform anything other than "source level" debugging.

Automatic control configuration for efficiency

The present system will automatically construct a parser from a given control specification. It should be possible to take this one step further. That is, given a set of principles, the user should have the option of allowing the system to choose an appropriate (i.e. efficient) control configuration for that particular set. Automatically tailoring the control structure to fit principles would be another step in raising the level of abstraction of the system. It is a relatively simple matter to construct such a decision procedure. In the current model, we have that type cardinality is a good indicator of whether interleaving is appropriate for a given principle. For principles that have been determined to be unsuitable candidates for interleaving, a determinacy detector can be used to distinguish filters from possible generators, and thus assign an appropriate ordering.[3] Static measures of fan-out could also be used to prioritize generators.

[3] For most classes of logic programs, there only exist sound but incomplete determinacy detectors. That is, such procedures correctly detect many cases of determinacy, but will miss others. It would be interesting to see if determinacy detection (like type inference) will work well in the restricted domain of linguistic principles.

Towards the generation of efficient principle-based parsers

It is probably safe to say that the current system is much less efficient than an equivalent system that uses "hand-compilation". The problem lies in the weakness of the compilation procedures used. For example, in chapter 2 we discussed how Correa used information from a variety of different principles to produce more deterministic (and efficient) versions of chain formation and functional determination operations. Except for the possibility of interleaving, the current system (unnecessarily) preserves a strict separation between parser operations. For example, even if free indexation is interleaved with all of the Binding conditions, all indexing possibilities will still be produced (although the ill-formed choices will be eliminated much earlier than in the corresponding non-interleaved case). Partial evaluation on the conjunction of the principle definitions may lead to the elimination of inappropriate indexing choice points.[4]

[4] One problem that must be solved is the control of the partial evaluation process (e.g. selection of the goal to evaluate and when to stop useless evaluation). In many cases, such information is communicated to the partial evaluator using program annotations — which would be incompatible with the goal of maintaining transparency.

Bibliography

[1] Abney, S. & J. Cole. "A Government and Binding Parser." *Proceedings of NELS 15.* 1985.

[2] Abney, S. "Licensing and Parsing." *Proceedings of NELS 17. GLSA.* UMass/Amherst. 1987.

[3] M. Abramowitz & I.A. Stegun, *Handbook of Mathematical Functions.* 1965. Dover.

[4] Barton, G.E., Jr. *"Towards a Principle-Based Parser."* MIT A.I. Memo 788. July 1984.

[5] Berge, C., *Principles of Combinatorics.* 1971. Academic Press.

[6] Chomsky, N.A., *Lectures on Government and Binding: The Pisa Lectures.* 1981. Foris Publications.

[7] Aho, A.V. & J.D. Ullman. *"Principles of Compiler Design."* Addison-Wesley. 1977.

[8] Baltin, M.R., and A.S. Kroch eds. *"Alternative Conceptions of Phrase Structure."* The University of Chicago Press. 1989.

[9] Chomsky, N.A. *"Lectures on Government and Binding."* Foris Publications. 1981.

[10] Chomsky, N.A. *"Some Concepts and Consequences of the Theory of Government and Binding."* MIT Press. 1982.

[11] Chomsky, N.A. *"Barriers."* M.I.T. Press. 1986.

[12] Chomsky, N.A. *"Knowledge of Language: Its Nature, Origin, and Use."* Prager. 1986.

[13] Chomsky, N.A. "Linguistics and Adjacent Fields: the State of the Art. A personal view." Israel *(m.s.)* 1988.

[14] Chomsky, N.A. "Some Notes on Economy of Derivation and Representation." *MIT Working Papers in Linguistics.* Vol. 10. 1989.

[15] Correa, N. *"Syntactic Analysis of English with respect to Government-binding Grammar."* Ph.D. thesis. Syracuse University. 1988.

[16] Correa, N. "Empty Categories, Chain Binding, and Parsing." *The MIT Parsing Volume 1987–88.* Center for Cognitive Science. MIT.

[17] Crocker, M.W. *"A Principle-Based System for Syntactic Analysis," (m.s.)* 1989.

[18] Dennis J.B. "Modularity." *Software Engineering.* Springer-Verlag. 1973.

[19] de Marcken, C.G. "Parsing the LOB Corpus." *Proceedings of the 28th Meeting of the ACL.* 1990.

[20] Dorr, B.J. "UNITRAN: A Principle-Based Approach to Machine Translation." S.M. thesis. MIT Department of Electrical Engineering and Computer Science.

[21] Fischer, C. & R.J. LeBlanc, Jr. *"Crafting a Compiler."* Benjamin/Cummings. 1988.

[22] Fodor, J.A. *"The Modularity of Mind."* MIT Press. 1983.

[23] Fong, S. & R.C. Berwick "The Computational Implementation of Principle-Based Parsers," *International Workshop on Parsing Technologies.* Carnegie Mellon University. 1989.

[24] Frank, R. "Licensing and Tree Adjoining Grammar in Government Binding Parsing." *Proceedings of the 28th Meeting of the ACL.* 1990.

[25] Fujisaki, T. *et al.* "A Probabilistic Parsing Method for Sentence Disambiguation." *Internation Workshop on Parsing Technologies.* Carnegie Mellon University. 1989.

[26] Fujita, H. "An Algorithm for Partial Evaluation with Constraints." ICOT TM-367.

[27] Graham S.L., *et. al.* "An Improved Context-Free Recognizer." *ACM Transactions on Programming Languages and Systems.* Vol. 2, No. 3, July 1980.

[28] Hentenryck, P. van. *Constraint Satisfaction in Logic Programming.* MIT Press. 1989.

[29] Hoji, H. "Logical Form Constraints and Configurational Structures in Japanese." Ph.D thesis. Department of Linguistics. University of Washington. 1985.

[30] Hornstein, N. *"Logic as Grammar: An Approach to Meaning in Natural Language."* MIT Press. 1984.

[31] Johnson, M. "Use of the Knowledge of Language," *Journal of Psycholinguistic Research.* 18(1). 1989.

[32] M.B. Kashket, *"A Government-Binding Based Parser for Warlpiri, a Free-Word Order Language."* TR-993, MIT A.I. Laboratory.

[33] Knuth, D.E., *The Art of Computer Programming: Volume 1 / Fundamental Algorithms.* 2nd Edition. 1973. Addison-Wesley.

[34] Lasnik, H. & M. Saito "On the Nature of Proper Government." *Linguistic Inquiry.* Vol. 15, Number 2. 1984.

[35] Lasnik, H. & J. Uriagereka *A Course in GB Syntax: Lectures on Binding and Empty Categories.* 1988. M.I.T. Press.

[36] Lloyd, J.W. *"Foundations of Logic Programming,"* 2nd Ed., Springer Verlag. 1987.

[37] Macías, B. *"Government-Binding Theory and Parsing as Deduction." (m.s.)* 1989.

[38] Makino, S. & M. Tsutsui. "A Dictionary of Basic Japanese Grammar." The Japan Times. 1986.

[39] Milner, R. "A Theory of Type Polymorphism in Programming," *Journal of Computer and System Sciences.* 17, 348–375. 1978.

[40] Naish, L. *Negation and Control in Prolog.* Springer-Verlag. 1985.

[41] Page, E.S., and L.B. Williams. *"An Introduction to Computational Combinatorics."* Cambridge University Press. 1979.

[42] Pager, D. "A Practical General Method for Constructing *LR(k)* Parsers." *Acta Informatica.* 7. 249–268. 1977.

[43] Petrick, S.R. *"A Recognition Procedure for Transformational Grammars."* Ph.D. thesis. Department of Modern Languages. 1965.

[44] Purdom, P.W., Jr. & C.A. Brown, *The Analysis of Algorithms.* 1985. CBS Publishing.

[45] Reingold, E.M., *et al.* *"Combinatorial Algorithms: Theory and Practice."* Prentice-Hall. 1977.

[46] Riemsdijk, H.C. van, and E. Williams. *"Introduction to the Theory of Grammar."* MIT Press. 1986.

[47] Riordan, J. *"An Introduction to Combinatorial Analysis."* John Wiley & Sons. 1958.

[48] Sharp, R.M. "A Model of Grammar Based on Principles of Government and Binding Theory." M.S. thesis. Department of Computer Science. University of British Columbia. 1985.

[49] Samuels, A.L., "Some Studies in Machine Learning Using the Game of Checkers. II — Recent Progress," *IBM Journal.* November 1967.

[50] Smith, D.E., & M.R. Genesereth, "Ordering Conjunctive Queries," *Artificial Intelligence* **26** (1985) 171–215.

[51] Sterling, L. & E. Shapiro, *"The Art of Prolog."* MIT Press. 1986.

[52] Stabler, E.P., Jr. *"The Logical Approach to Syntax: Foundations, Specifications and Implementations of Theories of Government and Binding." (m.s.)* M.I.T. Press. 1989.

[53] Stabler, E.P., Jr. *"Avoid the Pedestrian's Paradox." (m.s.)* 1990.

[54] Takeuchi, A., and H. Fujita. "Competitive Partial Evaluation — Some Remaining Problems of Partial Evaluation." ICOT TR-361.

[55] Tamaki, H., and T. Sato. "Unfold/fold Transformations of Logic Programs." *Proceedings of the 2nd International Logic Programming Conference.* Uppsala University, Sweden. 1984.

[56] Tomita, M. *"Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems."* Kluwer. 1986.

[57] Webelhuth, G. "Syntactic Saturation Phenomena and the Modern Germanic Languages." Ph.D thesis. Dept. of Linguistics. University of Massachusetts. 1989.

[58] Wehrli, E. *"A Government-Binding Parser for French."* Working Paper No. 48. University of Geneva.

Appendix A

The Linguistic Theory

The following definitions constitute a complete description of the linguistic theory implemented in the current system. Most of the theory described is taken directly from Lasnik & Uriagereka [35]. We have attempted to supply references to the appropriate sources in cases where modifications and additions to the basic theory have been made.

A.1 Basic Definitions

In this section, we define basic notions such as arguments, c-command, chains, and government, that will be used in the definitions of the principles in the next section. Assume that the order of constituents is unimportant unless otherwise stated. (The definitions are given in alphabetical order.)

Definition 1 (Anaphor)

NPs are marked with the feature $\pm\mathsf{A}$ (Anaphoric), implemented as `a(+/-)`. The value of this feature is lexically determined for overt NPs. For empty NPs, this feature is instantiated by functional determination [Defn. A.2.6].

Definition 2 (A-position)

A-positions are potential θ-positions, that is, positions where θ-roles may be assigned. $\overline{\text{A}}$-positions are simply those positions that are not A-positions. The following positions are defined to be A-positions:

a. Complement positions of lexical heads corresponding to internal θ-grid roles.

b. Specifier of IP.

c. Specifier of NP, but only when occupied by a NP (not when occupied by a determiner).

The feature `apos` will be associated with all A-positions, i.e. if a constituent X occupies an A-position, then X `has_feature apos` will hold.

Definition 3 (Argument) [pg101,Chomsky [9]]

Arguments are the elements that bear a θ-role. The status of elements as arguments will be determined by lexical and positional features. Note that some elements considered to be arguments at D-structure may have non-argument status at other levels of representation, e.g. at S-structure and LF. The following overt elements are arguments:

a. Quantifiers in A-positions only.

b. *wh*-NPs in A-positions only (see [pg115,Chomsky [9]]).

c. All other overt NPs unless explicitly marked as non-arguments (lexical feature `nonarg`).

d. All clauses in A-positions.

The following empty elements are also arguments:

e. *pro* [Defn. 15].

f. *PRO* [Defn. 16].

g. Variables [Defn. 21].

h. Empty NPs in A-positions (at D-structure) from which empty operators (*Op* [Defn. 14]) are formed by $\overline{\text{A}}$-movement.

Definition 4 (Binding)

An element A *binds* an element B (equivalently, B is *bound* by A) if:

a. A c-commands B, [Defn. 5], and

b. A and B are coindexed [Defn. 8].

c. A and B are not coindexed for agreement purposes, e.g. subject and Infl ([Defn. 66]).

An element is said to be *free* if it is not bound. An element A *A-binds* an element B if A binds B and A is in an A-position. Similarly, A *$\overline{\text{A}}$-binds* B if A binds B and A is in an $\overline{\text{A}}$-position.

Definition 5 (C-Command)

"First branching node" definition: In $[\alpha\ X \ldots Y\ ]$, where X and Y are immediate constituents of α:

a. X c-commands all constituents *dominated by* Y ([Defn. 10]).

b. If X has form $[\beta\ Z\ ]$, Z being the only immediate constituent of β, then case (a) applies with Z in place of X. Also case (b) applies recursively with Z in place of X.

(Case (b) is actually implemented by having non-branching structures automatically collapsed during structure building.)

Definition 6 (C-Command$'$) [pg63f.,Lasnik & Uriagereka]

"Accessible SUBJECT" ([Defn. 25]) extension to c-command:

a. X c-commands$'$ Y if X c-commands Y.

b. I(AGR) c-commands$'$ NP in $[IP$ NP $[\bar{I}\ [I\,]\ \ldots\ ]]$, where I(AGR) denotes an inflection element with a non-vacuous AGR (agreement) component.

(AGR is implemented as a feature `agr(F)` where `F` encodes person, number and gender features (see section A.3.1). Hence I `has_feature agr(_)` must hold in the second case above.)

Definition 7 (Chains)

Chains are formed by movement [A.2.9]. A chain $\langle E_1, \ldots, E_n\rangle, \quad n \geq 1$ is a sequence of elements where E_1 is the *head* of the chain, and E_n, the last element of the chain. Each *link* $\langle E_i, E_{i+1}\rangle$ of a chain represents a single instance of movement of the head E_1 from the position occupied by the trace E_{i+1} to the position occupied by E_i. E_i is known as the *antecedent* of E_{i+1}. All elements of a chain will be assigned the same index.[1] When $n > 1$, E_n is known as a *base trace*. Furthermore, for $n > 2$, elements $E_2, \ldots, E_{n-1}$ are known as *intermediate traces*.
(Chains are represented by having each element of a non-trivial chain contain a syntactic feature `chain(`X`,`T`)`, where X is the antecedent of the element (or `[]` if none exists) and T is one of `head`, `base` and `medial`.)

Definition 8 (Coindexation)

Two elements A and B are said to be coindexed only if they have both been assigned indices, and the values of the indices are identical.

[1] Note, it need not be the case that all elements with same index will be part of the same chain.

(Indices are represented using the syntactic feature `index(I)`, `I` being a logic variable. This feature is assigned by either movement, free indexation ([Defn. 4]) or coindexation of subject and inflection ([Defn. 66]).)

Definition 9 (Comp)

The specifier of CP, if one exists, and the head of CP, C, are collectively referred to as Comp.

Definition 10 (Dominates)

Dominates is the primitive structural relation that captures the notion of hierarchy:

a. Any constituent dominates itself.

b. Let S be an element with immediate constituents $S_1, \ldots, S_n$, $n \geq 1$. S dominates a consituent P if for some S_i, $1 \leq i \leq n$, S_i dominates P.

An element S is said to *properly dominate* an element P if S dominates P, and S and P are distinct elements.

Definition 11 (Government)

Goverment is a structural relation between two elements:

a. *Core government.*

Let X be the head of a maximal projection XP. Then X governs any element P that XP dominates, provided:

1. X is a governor [Defn. 12], and
2. P does not dominate, nor is it dominated by X, and
3. There is no maximal projection YP, distinct from XP, that is dominated by XP yet dominates P.

b. *Exceptional government I.*

C governs the NP specifier of its complement (IP), provided C has lexical feature `ecm`, e.g. *for* in English.

c. *Exceptional government II.*

A lexical head X governs the IP specifier of its clausal complement, provided one exists, if:

1. X has lexical feature `ecm(oblig/opt)`, and
2. the clausal complement is a non-finite clause.

Definition 12 (Governor)

The following categories are considered to be governors:

a. Lexical heads, i.e. N, V, A and P.

b. Inflection I(AGR) containing a non-vacuous AGR element. ([Defn. 6] describes how AGR is represented.)

Definition 13 (NP-trace) [pc,Lasnik]

An NP-trace is the trace of movement of an NP from an A-position to another A-position.

Definition 14 (Operator)

The status of overt elements as operators is determined by lexical features. The following elements are considered to be operators:

a. Quantifier NPs marked with lexical feature `op(+)`, e.g. *everyone* and *someone* in English. Such NPs will form operator-variable constructions at LF.

b. [$+wh$] NPs (marked with lexical feature `wh`), e.g.*who* in English.

c. [$+wh$] adverbs, e.g. *why* and *how* in English.

d. An empty NP in an $\overline{\mathrm{A}}$-position marked as an operator *Op*.

(This is implemented by a feature `ec(op)`. It is assumed that empty operators are only formed through movement.)

Definition 15 (*pro*)

---

pro is an empty NP with Binding features $[-A, +P]$. *pro* is stipulated to not appear in complement positions. (Also, the *pro*-Drop parameter controls its availability for Japanese but not for English.)

Definition 16 (*PRO*)

PRO is an empty NP with Binding features $[+A, +P]$.

Definition 17 (Pronominal)

NPs are marked with the feature ±P (Pronominal), implemented as `p(+/-)`. As with anaphors [Defn. 1], the value of the feature will be determined lexically for overt NPs, and functionally, for empty NPs.

Definition 18 (R-expression)

An R-expression is any NP (overt or non-overt) marked with Binding features $[-A, -P]$.

Definition 19 (Subject)

The following positions are subject positions:

1. NP specifier of IP, and
2. NP specifier of NP.

Definition 20 (θ-grids)

Thematic grids encode the selectional properties of lexical heads. That is, they specify the range of external and internal θ-roles that a lexical head may assign. In the following sections, θ-grids will be denoted by a tuple $(\langle\theta_1 \ldots \theta_m\rangle, \langle\theta'_1 \ldots \theta'_n\rangle)$, $n, m \geq 0$. The first element contains the external θ-roles, and the second, the internal θ-roles.

Definition 21 (Variable)

A variable is an empty NP that is locally $\overline{\text{A}}$-bound by an operator. It has Binding features $[-A, -P]$. Note:

1. A variable may or may not be a trace — for example, a parasitic gap.
2. Not all empty NPs with features $[-A, -P]$ are variables (see [Defn. 47] for details).

Definition 22 (Wh-trace) [pc,Lasnik]

A wh-trace is the trace of the movement of a NP with feature $[+wh]$ from an A-position to an $\overline{\text{A}}$-position. (In particular, this definition will exclude intermediate traces in Comp.)

A.2 The Modules of Grammar

In this section we define the principles that make up the modules of grammar. The principles are organized as shown in figure A.1.

A.2.1 Binding Theory

Binding Conditions impose restrictions on the coreference possibilities for anaphors, pronominals and R-expressions. In the current theory, *PRO*, defined as having Binding features $[+A, +P]$, is treated specially. In particular, *PRO* will be allowed to 'escape' both Binding conditions A and B by virtue of not having a *Governing Category* (GC).

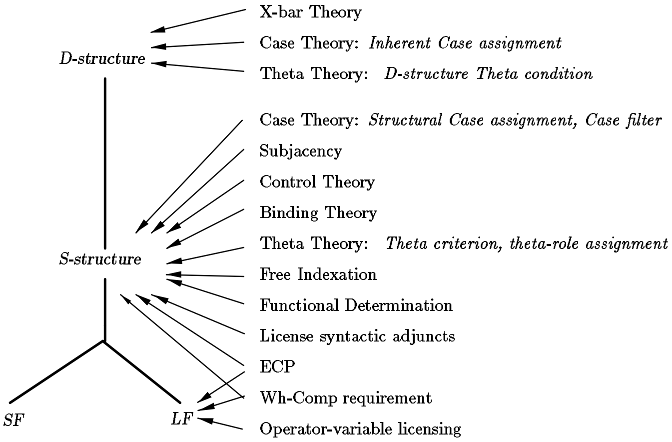


Figure A.1 The modules of grammar.

Definition 23 (Governing Category)

The governing category of an element A, GC(A), is the smallest *GC domain* containing:

a. A itself, and

b. an element G that *governs* A ([Defn. 11]), and

c. an element S such that S is an *accessible SUBJECT* to A ([Defn. 25]).

Definition 24 (GC Domain)

The GC domains are IP and NP.

Definition 25 (Accessible SUBJECT)

A constituent S is an accessible SUBJECT to A if:

a. S is a SUBJECT, and

b. S is accessible to A, that is:

1. S *c-commands'* A ([Defn. 6]), and
2. S is not coindexed with any category properly dominating A.

Definition 26 (SUBJECT)

An element X is a SUBJECT if:

1. X is a *subject* ([Defn. 19]), or
2. X is I(AGR) (see [Defn. 5]).

Definition 27 (Condition A)

All anaphors, i.e. NPs marked with feature `a(+)`, must be A-bound ([Defn. 4]) within its governing category ([Defn. 23]), if one exists.

Definition 28 (Condition B)

All pronominals, i.e. NPs marked with feature `p(+)`, must be A-free in its governing category, if one exists.

Definition 29 (Condition C) [pg98,Chomsky [12]]

Let X be an R-expression ([Defn. 18]). Then:

a. If X is part of a non-trivial chain, i.e. $X = E_i$ for some chain $\langle E_1, \ldots, E_n \rangle$, $1 \leq i \leq n$ and $n > 1$, then X must be *A-free* ([Defn. 4]) in the domain of the head of the chain E_1. Otherwise:

b. X must be A-free.

A.2.2 Control Theory

The following definitions deal with the cases in which *PRO* appears in a declarative clause complement or in a purposive adjunct clause, that is, in cases where the obligatoriness of local binding of *PRO* (wherever possible) holds. The definitions do not work for freer cases (e.g. interrogative clausal complements) where *PRO* is *optionally* controlled by the nearest appropriate binder.

Definition 30 (Control of *PRO*) [pg124,Chomsky [12]]

PRO must be controlled by the nearest *appropriate binder*, if one exists.

Definition 31 (Appropriate Binder) [pg124,Chomsky [12]]

To be an appropriate binder, an NP X must satisfy the following conditions:

a. X must *A-bind* *PRO* ([Defn. 4]).

b. X must not be lexically marked with feature `nonarg`, e.g. excludes the non-argument *it*.

c. X must not in a position associated with the feature `noControl`.

This definition deviates slightly from Chomsky's version. The last condition is designed to handle sentences like *John promised Mary PRO to leave* (cf. *John persuaded Mary PRO to leave*) where the nearest binder, *Mary*, cannot be the controller. Instead, the appropriate binder in this case is the next nearest binder *John*. This is accomplished by lexically marking the θ-role corresponding to *Mary* with the feature `noControl` (see section A.3.1).

A.2.3 Case Theory

The version of Case theory adopted is basically the "morphological" account of the Case Filter [pg9ff.,Lasnik & Uriagereka] — as opposed to an account based on the notion of 'visibility'. The only significant addition is for inherent Case assignment which occurs at D-structure (following Chomsky [12].[2])

Definition 32 (Structural Case assignment)

A head X assigns Case C at S-structure to an NP S under the following conditions:

a. X is a structural Case assigner that assigns Case C, and

b. X *governs* S ([Defn. 11]), and

c. X is adjacent to S if language parameter `CaseAdjacency` holds.

Note:

- Verbs only assign structural Case to direct objects. (See the description of inherent Case assignment below.)
- Elements marked as being optional Case assigners, i.e. having feature `ecm(opt)` (see section A.3.1), should have government operate consistently for structural Case assignment and Binding theory. For example, in *I want PRO to win*, *want* should not govern *PRO* — otherwise *PRO* will be Case-marked and also (incorrectly) have a governing category.

Definition 33 (Structural Case assigners)

The following elements are considered to be structural Case assigners:

a. I(AGR) assigns nominative Case.

b. V assigns accusative Case if:

1. V is not marked `noCasemark`, and

[2] Actually, Chomsky's approach to *of*-insertion is discussed very briefly in the context of illicit NP-movement in Lasnik & Uriagereka ([pg152]).

2. V is not marked as a dative Case assigner (in which case, it assigns dative Case to its direct object).

   For example, some verbs, e.g. in Japanese *nar* ('become') as in *Taro-wa naze kubi*-DAT *natta no* ((37b) in Lasnik & Saito [34]), seem to assign dative Case to their direct objects.

c. C with feature `ecm` assigns oblique Case, e.g. *for* in English.

Definition 34 (Inherent Case assignment) [pg192,KofL,Chomsky,86]

A head X assigns Inherent Case at D-structure to an NP S if:

a. X assigns a θ-role to S, and

b. X is an Inherent Case assigner.

The configurations of inherent Case assignment parallel those of θ-role assignment ([Defn. 51]).

Definition 35 (Inherent Case assigners) [pg193,KofL,Chomsky,86]

The following elements are the inherent Case assigners:

a. Prepositions assign oblique Case.

b. Nouns assign genitive Case.

c. Adjectives assign genitive Case.

d. Verbs assign genitive (or dative) Case for indirect objects only (roughly following [note 130, pg.219,Chomsky,86]).

Definition 36 (Genitive Case Realization) [pg194f,Chomsky [12]]

In English, genitive Case assigned to an NP S is realized according to the following rules:

a. If S is in a complement position, then genitive Case is realized as the semantically null marker *of*, otherwise:

b. If S is in a subject position, then genitive Case is realized as the possessive marker *'s*. Genitive Case may also be realized morphologically for pronouns, e.g. *me* + *'s* is equivalent to *my*.

Note, it is assumed that the markers *of* and *'s* do not form constituents at S-structure, but instead are inserted at Surface Form.

Definition 37 (Case Filter)

At S-structure, every lexical NP needs Case.

Definition 38 (Case Transmission)

The current implementation deviates from Lasnik & Uriagereka in requiring Case transmission to take place for A- to $\overline{\text{A}}$-movement for two reasons:

1. Because of the possibility of scrambling in Japanese, e.g. [*VP nani*-ACC$_i$ [*VP te*-DAT [*NP-t*]$_i$ *ireta*]] as in *Taro-ga nani-o te-ni ireta koto-o sonnani okotteru no* ((39a) in Lasnik & Saito [34]), it is assumed that accusative Case is transmitted from the trace to a VP-adjunct position (an $\overline{\text{A}}$-position according to [Defn. 2]) occupied by the scrambled direct object. Also, the current definition of government excludes the verb from governing VP-adjunct positions.

2. Without Case transmission, the Case filter will have to be modified to exclude operators in $\overline{\text{A}}$-positions, as in *who*$_j$ *did John see* t_j where it is assumed that the trace rather than the operator *who* requires Case.

However, the present definition is somewhat unsatisfactory since Case transmission overlaps the condition that *wh*-traces must have Case ([Defn. 39]).

Definition 39 (Trace Case Condition)

---

a. All Wh-traces must have Case.

b. No NP-trace can have Case.

A.2.4 Empty Category Principle (ECP)

The ECP holds both at S-structure and at LF.

Definition 40 (ECP)

All empty categories except for the empty complementizer C, empty operators and *PRO* must be properly governed. An empty category is properly governed if it is *lexically governed* or *antecedent governed.*

Definition 41 (Lexical Government)

An element S is lexically governed if there exists an element P such that:

a. P governs S [Defn. 11], and

b. P is a lexical governor.

Definition 42 (Lexical Governors)

The following elements are the lexical governors:

a. Lexical heads [Defn. 62].

b. I(AGR) if the *pro*-Drop parameter holds.

Definition 43 (Antecedent Government)

An element S is antecedent governed if there exists a local binder P such that there is no intervening *barrier*.

Definition 44 (Barriers for Antecedent Government)

The following categories are considered to be barriers to antecedent government:

a. All NPs.

b. All clauses (CP) unless:

   1. Element S is in specifier of COMP — for antecedent government into COMP, or
   2. Clause CP has been marked with feature `noBarrier`.

Here, we deviate slightly from Lasnik's account of the *That-trace* effects (sections 4.1.2–4.1.4 in the reference text). Lasnik assumes that the trace in an example like '*Who[1] do you think [that [t[1] saw Bill]]' is not properly governed because there is no trace (in COMP) that antecedent governs it. The assumption is that there cannot be both 'that' and a trace in COMP. Furthermore, it is assumed that this is not a Subjacency violation despite the absence of an intermediate trace. The apparent conflict between the absence of an intermediate trace for ECP and Subjacency is reconciled by having movement *optionally* leave a trace. However, in this implementation, we stipulate that movement always leaves traces, but the same judgements are achieved by having 'that' (as the head of COMP) mark the trace in the specifier position as being 'invisible' to ECP.

Definition 45 (Invisible traces)

A trace is invisible to ECP (it need not be properly governed, nor can it be a proper governor) if it is marked `invisible`.

A further modification to the above concerns head movement (see [A.2.9] for details.) The ECP also applies to all traces of head movement. Unfortunately, the above definition also rules out all cases of INFL lowering to V (e.g. when AGR is "weak", but the verb is "strong"). That is, the trace of INFL is not antecedent governed when INFL adjoins to V. Hence, the actual definition has an extra clause tailored to permit precisely this case.

A.2.5 Free Indexation

In this theory, elements do not intrinsically possess indices. Instead, indices are assigned through one of three mechanisms. Binding Theory relevant elements may receive indices through movement [A.2.9], or through the coindexation of subject of IP and I [Defn. 66]. Also, at S-structure, free indexation stipulates that:

Definition 46 (Indexation at S-structure)

Indices are freely assigned to all elements in A-positions only.

The A-position restriction prevents operators such as wh-words in COMP from receiving an index at S-structure by any means other than through movement. For example, the preferred analysis for the sentence 'who left' is 'who[i] t[i] left' where 'who' has moved from the subject position occupied by the trace. In the case where the A-position stipulation is eliminated, there is another analysis 'who[i] e[i] left', where 'who' and the empty element are both 'base-generated' in the positions shown, and then (freely) assigned identical indices at S-structure. With free indexing restricted to A-positions, this second analysis becomes 'who e[i] left' which will be ruled by Full Interpretation [A.2.7].

A.2.6 Determination of Empty Categories

Empty noun phrases initially are devoid of the two Binding theory-relevant features, $\pm$`A` and $\pm$`P`. These two features are instantiated according to the following algorithm:

Definition 47 (Functional Determination)

Let A be an empty NP in an A-position. Then:

1. Let B be the nearest binder of A (if one exists). Then:
   (a) If B is in an A-position, then:
   A has feature `p(-)` if A and B do not have an independent θ-role, i.e. are part of the same chain. Otherwise, A has feature `p(+)`.
   (b) If B is in an $\overline{\text{A}}$-position, then:
   A is a variable [Defn. 21] (with features `[a(-),p(-)]`) if B is an operator. Otherwise, A has feature `p(-)`.
2. Otherwise A must be unbound. Then A has feature `p(+)`.

Except for the case where A is a variable, the feature `a(+/-)` is freely assigned as follows:

a. A may have feature `a(+)`.

b. A may have feature `a(-)` if the *pro*-Drop parameter is set.

Notes:

1. Functional Determination is deterministic only when the *pro*-Drop parameter is not set (as in English, but not Japanese).

2. Functional Determination does not apply to intermediate traces and empty operators in Comp. These do not receive anaphoric and pronominal features.

A.2.7 Full Interpretation

Full Interpretation requires that every element of the interface to syntax, that is, the two levels of PF and LF, must receive an appropriate interpretation. The following principles constitute a partial implementation of Full Interpretation for licensing of LF representations only.

Definition 48 (License Operator-Variable Constructions)

In this theory, we do not adopt the Bijection Principle. In other words, operators and variables need not be in strict one-to-one correspondence. For example, in the case of parasitic gap constructions such as *which report*[i] *did you file t*[i] *without reading e*[i], the operator *which report* binds both the trace variable, and the parasitic gap *e*[i].

a. An operator [Defn. 14] must bind at least one variable [Defn. 21], and

b. A variable must be *strongly bound* by an operator, and

c. A variable that is also a trace must be bound by the operator that heads its chain.

Definition 49 (Strong Binding) [pg108ff,KofL,Chomsky,86]

A variable x is strongly bound if:

a. x is bound by an overt operator, otherwise:

b. if x is bound by an empty operator, then x's range must be determined by being also bound by an overt, non-pleonastic element.

Definition 50 (License Syntactic Adjuncts) [pg94f,van Riemsdijk & Williams,86]

Basically, all syntactic adjuncts must be one-place predicates. The predicate status of adjuncts may be either determined structurally or through lexical features. (In the actual implementation, this condition is enforced at S-structure.) The following adjuncts are licensed as one-place predicates at LF:

a. A clause of the form $\lambda x.(\ldots x \ldots)$ where λx is an operator and x a variable.

The operator may be realized at LF as an empty operator *Op* as in the restrictive relative clause construction [*NP* [*NP the man*] [*CP Op I saw*]], or as an overt operator such as *who* in [*NP* [*NP the man*] [*CP who I saw*]]. Furthermore, the operator and the element to which it is adjoined cannot have different category labels — cf. **The man how fast we ran.*

b. An adjective with one (unsaturated) external θ-role in its θ-grid — e.g. *the sad man* versus *the man is sad.*

c. A head with lexical feature `predicate(Type)` where `Type` is one of `location, manner, time`, etc. — e.g. the adverb *sincerely* and the PP headed by preposition *by* have type 'manner' and 'agentive', respectively.

The above types do not cover all the possible cases of adjunction. *Wh*-adverbs that adjoin to VP are excluded from this condition. Furthermore, adjunction resulting from head movement, VP adjunction from scrambling and LF movement are also all excluded.

A.2.8 θ-Theory

Definition 51 (θ-Assignment)

Modified from Chomsky's definition ([pg142,KofL,Chomsky,86]) to accommodate the three-bar level scheme used for verbs. The configurations of θ-role-assignment are:

a. [$\overline{\mathrm{X}}$ X YP]

A head X with θ-grid $(\ldots, \langle\theta_1 \ldots \theta_n\rangle)$ (n=1,2) assigns its internal role θ_1 to its complement (i.e. direct object) YP.

b. [$\overline{\overline{\mathrm{X}}}$ $\overline{\mathrm{X}}$ YP]

A head X with θ-grid $(\ldots, \langle\theta_1 \ldots \theta_2\rangle)$ assigns its internal role θ_2 to its complement (i.e. indirect object) YP.

c. [*NP* NP-SPEC [$\overline{\mathrm{N}}$ ... N ...]]

A head noun N with a θ-grid $(\langle\theta\rangle, \ldots)$ *optionally* assigns its external θ-role θ (provided one exists), to its specifier NP-SPEC.

d. [*IP* IP-SPEC [$\overline{\mathrm{I}}$ I VP]]

A clausal subject *IP-SPEC* may receive the external θ-role of a predicate, either directly from the VP, or by percolation from a lower predicate. For example, in [*IP John* [*VP is* [*AP happy*]]], the external θ-role of the adjective *happy* will be assigned to the subject *John*.

Definition 52 (θ-criterion) [pg42,LGB,Chomsky,81]

The θ-criterion at S-structure is formulated as a property of movement chains. Given a chain $\langle E_1, \ldots, E_n \rangle$, $n \geq 1$. Then:

a. E_n must be assigned a θ-role, if and only if, the chain is an A-chain.

b. E_i, $1 \leq i < n$, must not be assigned a θ-role.

Definition 53 (A-chain)

The definition of an A-chain, that is, a chain headed by an argument, must take into account whether the head is in an A-position or not, because the argumenthood of some elements hinge upon this distinction — see [Defn. 3]. The reference text is unclear on this point. However, it seems that the correct definition of an A-chain must consider the argument status of its head *at D-structure*, not at S-structure — otherwise, examples involving Wh-movement such as *who*[i] *t*[i] *left* will be incorrectly ruled out. Hence:

> A chain $\langle E_1, \ldots, E_n \rangle$, $n \geq 1$, is an A-chain if E_1 is an argument at D-structure.

Definition 54 (D-structure θ-Condition) [pg97f,KofL,Chomsky,86]

By the projection principle, the θ-criterion must hold at every syntactic level of representation. The θ-criterion as formulated in the previous section only applies at S-structure. The fact that the θ-criterion must also hold at D-structure is separately expressed as a uniformity condition with respect to θ-marking:

> At D-structure, if α T-governs X then X is an argument if and only if α θ-marks X.

Note: For example, [*CP someone*$_i$ [*IP* x_i [*VP likes everyone*]]] is a candidate S-structure generated by the parser. Here, the variable x_i has been coindexed with the quantifier *someone* not through movement but through free indexation. (The θ-criterion [Defn. 52] will fail to rule this out.) However, the corresponding D-structure will violate the uniformity condition. The null element that occupies the position of the variable at D-structure is θ-marked by the VP but it is not an argument.

Definition 55 (T-government) [pg98,KofL,Chomsky,86]

There are two configurations where T-government obtains:

a. $[X^{i+1}\ \mathrm{X}^i\ \mathrm{YP}\ ]$, $\quad i = 0, 1$

A head X T-governs its complement(s) YP.

b. A predicate T-governs its subject.

A.2.9 Trace Theory

Movement in Syntax

We assume that empty operators in constructions such as restrictive relative clauses, as in "the man [*Op* [I saw *e*]]," are base-generated in A-positions at D-structure and fronted by Move-α (see [pg84f, KofL, Chomsky, 86]).

Definition 56 (Chain Formation)

Movement chains are formed in accordance with the "history of movement" hypothesis. Furthermore, we assume that traces are obligatory for movement.

Head Movement

A very simple theory of head movement was implemented for two basic reasons: (1) to allow cases of *Do-support* where an auxiliary verb raises to COMP, and (2) to account for the particular ordering of verbal adjuncts. In this theory, languages are parameterized according to whether AGR is *strong* or *weak*. In English, AGR is weak, but for languages like French, AGR is considered to be strong. For simplicity, AGR and TNS are not considered to be distinct entities from INFL. We adopt the following restrictions:

a. Only inflection and verb heads are allowed to move.

b. Movement of an I–V (or V–I) complex is permitted to the head of Comp.

c. Movement is by adjunction, not substitution.

d. V moves in preference to I wherever possible.

The traces left by head movement must obey the ECP. However, the case where Infl must lower to V, e.g. for regular (non-auxiliary) verbs in English, is problematic for the ECP — see [A.2.4] for details.

Conditions on Syntactic Movement

Definition 57 (Subjacency)

Let E_i and E_{i+1} be a pair of elements from a chain $\langle \ldots E_i, E_{i+1}, \ldots \rangle$. Then E_i and E_{i+1} are subjacent if there exists an element S such that:

a. S dominates both E_i and E_{i+1}, and

b. there does not exists two distinct *bounding nodes*, B_1 and B_2, such that B_1 and B_2 both properly dominate E_{i+1} and are properly dominated by S.

For English, IP and NP are considered to be the bounding nodes.

Definition 58 (*Wh*-movement in Syntax) [Lasnik & Saito,84]

Languages are parameterized according to whether they allow *wh*-movement in syntax; that is, between D- and S-structure and D-structure. For example, English is +WhInSyntax, and Japanese as -WhInSyntax. If +WhInSyntax, then:

a. A [+*wh*] Comp must have a [+*wh*] head.

b. A [-*wh*] Comp must not have a [+*wh*] head.

In the current implementation, "head" refers to both the head and the specifier position of Comp (the target position for *wh*-movement).

LF Movement

LF movement is restricted to the processes of Quantifier Raising (QR) and LF *wh*-movement.

Definition 59 (Quantifier Raising)

Quantifiers marked with lexical feature `op(+)` are freely raised to adjoin to the nearest IP.

Note: unlike LF *wh*-movement, QR is a local operation with no successive cyclic movement allowed.

Definition 60 (LF *Wh*-movement)

Elements marked with lexical feature `wh` are freely raised to Comp.

Movement is always accomplished by substitution unless the position in Comp is already occupied, in which case, moved elements are adjoined.

Conditions on LF Movement

The ECP and Operator–variable licensing operations are applied at LF. Note that Subjacency does not apply to LF *wh*-movement. The *Wh*-movement in Syntax filter also applies, but with one modification:

Definition 61 (*Wh*-Comp Requirement) [Lasnik & Saito,84]

Universally:

a. A [+*wh*] Comp must have a [+*wh*] head.

b. A [-*wh*] Comp must not have a [+*wh*] head.

There is an extra semantic parallelism requirement for the [+*wh*] head case to handle examples such as **Mary-wa John-ga nani-o katta ka do ka shiranai*, for which *ka do ka* ('whether or not') — a yes/no question element — and *nani* (meaning 'what') cannot both be present in the same Comp at LF.

A.2.10 $\overline{\text{X}}$-Theory

Definition 62 (Projection Levels)

In this theory we adopt a two-bar-level system for all categories except verbs (which makes use of a three-bar-level system). Heads constitute the zeroth projection level. N, V, A, and P are the *lexical heads.* I (Inflection) and C (Complementizer) are the *non-lexical* heads. *Intermediate projections* (or single-bar categories) are the first level projections of heads. Double-bar categories are *maximal projections* for N, A, P, I, and C, but still an intermediate projection for V. $\overline{\overline{\overline{\text{V}}}}$ is the maximal projection for V.

Definition 63 (Complements)

In general, complements of heads are attached as sisters of heads, and as sisters of intermediate projections *not* immediately dominated by maximal projections. The complement structure for lexical heads is primarily determined by individual θ-grid properties. For verbs, direct and indirect objects are attached to the $\overline{\text{V}}$ and $\overline{\overline{\text{V}}}$ projections, respectively. For non-lexical heads I and C, the possible complements are not determined by lexical properties, but simply stipulated as VP and IP, respectively.

Definition 64 (Specifiers)

Specifiers are attached as sisters of intermediate projections immediately dominated by a maximal projection. At most one specifier is permitted. The specifier position for I and C may be filled by NPs.[3] The specifier position of NP can be filled by an NP or a determiner. The specifier positions for V, P and A are currently left empty.

A.2.11 Miscellaneous Conditions

Definition 65 ($\overline{\text{S}}$-Deletion)

Heads (e.g. *believe* and *likely*) that normally take clausal complements (CPs), but are lexically marked as being $\overline{\text{S}}$-deleters will take IPs as complements instead.

In the current implementation, no deletion actually takes place. Instead, the CP node is marked as being "transparent" for conditions such as the ECP.

Definition 66 (Coindex subject of IP and Inflection)

$[_{IP}\ \text{NP}_i\ \ [_{\bar{I}}\ \ [_I\ ]_i \ldots\ ]]$

There are two reasons for coindexing the subject of IP and inflection:

[3] *Wh*-adverbs can also move to the specifier of Comp as the target site of *wh*-movement.

1. Subject–verb agreement in conjunction with head movement.
2. The determination of accessible SUBJECTS [Defn. 25].

A.3 Features

This section describes the lexical and syntactic features used in the current system.

A.3.1 Lexical Features

This section briefly lists the features used in lexical entries together with various examples of their use. The following table contains the basic lexical features:

Feature	Categories	Example	Explanation
`a(±)`	N	*himself*: `a(+)`	[±anaphoric]
`agr([P,N,G])`	N,det,V	*few*: `agr([3,pl,[]])`	Encodes person, number and gender features. Boolean combinations can be built using `not(_)` and lists (to represent disjunction). `[]` encodes a *don't care*, i.e. irrelevant, feature.
`count(+/-/_)`	det, N	*much*: `count(-)`	Used to associate determiners with particular classes of nouns.
`ecm(oblig/opt)`	V, A	*believe*: `ecm(oblig)`	Obligatory and optional Exceptional Case Markers.
`grid(ERs,IRs)`	N,V,A,P	*kau* ('buy'): `grid([agent], [theme,[location]])`	`ERs` and `IRs` hold the list of external and internal θ-roles, respectively. Optional roles are bracketed.
`left(C,F)`	mrkr	*'s* (possessive marker): `left(n,case(gen))`	A marker is an input element that becomes a feature `F` on the closest item of category `C` to its left.
`morph(R,F)`	V	*eaten*: `morph(eat,ed(2))`	Encodes the root `R` and particular form `F` of an individual entry.
`morphC(Case)`	N	*his*: `morphC(gen)`	Used to indicate morphologically realized Case on noun forms.
`noCasemark`	V	*seem*: `noCasemark`	Does not Case-mark its complement.
`noControl`	V	*promise*: `grid([agent], [patient$noControl, proposition])`	Indicates the unavailability of a θ-grid element as a controller for *PRO*.
`nonarg`	N	*it*: `nonarg`	Indicates the non-argument status of a noun.
`p(±)`	N	*him*: `p(+)`	[±pronominal]
`right(C,F)`	mrkr	*of* (mrkr): `right(n, case(gen))`	Similar to `left(C,F)`. Indicates that a marker becomes feature `F` on the nearest item of category `C` to its right.
`subcat(C$F`$_1$`,F`$_2$`)`	V(aux)	*be* (passive): `subcat( vp$[morph(_,ed(2))], [noCasemark])`	Subcategorization: `C` is the category of the complement which must have features `F`$_1$. `F`$_2$ are the features to be added to the complement.
`wh`	adv, N	*why* (adv): `wh`	[+wh]

More complex features can be used to construct arbitrary conditions from the basic features listed above. The following table lists the feature prototypes that are used to express various idiosyncratic or peripheral properties of lexical items not yet properly handled by the conventional set of principles.

Feature	Categories	Example	Explanation
`selR(R)`	all heads	*that* (C): `selR( not( feature(inf(_))))`	An escape mechanism that allows arbitrary restrictions to be placed on the complement of a head. `R` may be constructed using the forms `feature(F)` (complement must have feature `F`), `addFeature(F)` (add feature F to the complement), `goal(G,X)` (complement (named by `X`) must satisfy a goal `G`, and `not(R)` (negation) in conjunction with any of the previous forms.
`specR(R)`	all heads	*that* (C): `specR( addFeature(invisible))`	The corresponding mechanism for specifiers of heads.
`adjR(Rs)`	all heads	*by* (P): `adjR([goal( extRoleToAssign(XP, agent), XP), feature(passive)])`	Similar to `selR(R)` and `specR(R)`. Allows a sequence of conditions (`Rs`) to be imposed on adjuncts.

Appendix B

The Notation for Encoding Principles

The following definitions constitute a complete description of the primitive linguistic relations and macros used to encode the principles defined in the previous chapter. Since these definitions are implemented in Prolog, principle definitions will use Prolog's syntax and semantics. Type inference rules used by the principle interleaving process are also supplied where relevant. (See chapter 6 for an explanation of the notation used in the type definitions.)

B.1 Primitive Operations

In this section, we define the basic operations provided for manipulating constituent structure such as the extraction of complements and heads, category labelling, and feature set maintenance, e.g. membership, addition of new features, and chain link traversal. Many of these are 'one-way' or non-reversible in the sense that they have restricted modes of operation. For example, `complement_of` can only be used to extract complements but not to return the corresponding head (or projections of the head). In some cases, such restrictions are a consequence of the use of a simple term-based representation of phrase structure. Roughly speaking, sub-constituents are represented as arguments of a term. In this scheme, extraction of sub-constituents is straightforward. However, since no 'back-pointers' are used, higher constituents are not so easily referenced. In the other cases, simplicity rather than representational limitations has been the motivation for restricting the modes of operation. Note that not all primitives have been listed: in particular, there are more basic operations also available to the grammar writer that deal with set and open list manipulation. (The following definitions are given in alphabetical order.)

Definition 67 (`addFeature`)

Given a constituent X and a feature F, `addFeature`(F,X) (destructively) adds F to the feature set of X.

Definition 68 (`adjoin`)

Given two constituents X and Y, and a parameter `Dir` $\in$ {`left`, `right`}, `adjoin`(X,Y,Z,`Dir`) holds when Z is the constituent formed by adjoining X to Y according to `Dir`.

Definition 69 (`adjoined`)

Given two constituents X and Y, `adjoined`(X,Y) holds if Y is an immediate adjunct under X. Similarly, `adjoined`(X,Y,Z) holds if Y has been adjoined to Z forming the adjunction structure X.

Definition 70 (`baseTrace`)

Given a constituent E_n, `baseTrace`(E_n) holds if there exists a non-trivial chain $\langle E_1, \ldots, E_n \rangle, \quad n > 1$.

The corresponding type definition is:

> `baseTrace`($E_n : T : R$) holds $\Rightarrow T = U, R = \{\text{ec}\}$, i.e. E_n must be an empty category.

Definition 71 (`antecedent`)

Chain link chasing: given a trace E_{i+1}, `antecedent`(E_{i+1},E_i) holds if there exists a non-trivial chain $\langle \ldots, E_i, \ldots, E_{i+1}, \ldots \rangle, \quad 1 \leq i < i+1 \leq n, n > 1$, where n is the number of elements in the chain.

Definition 72 (`cat`)

Category labelling: given a constituent X and a label c, `cat`(X,c) holds if c is the category label of X. (Note, this may be used for both testing and setting the label of a constituent.)

There are two type definitions corresponding to the cases where c is instantiated or not:

> `cat`($X : T : R$,c) holds $\Rightarrow$
> $T = \{c\}, R = \emptyset$ if c is a category label, or
> $T = \text{typeValue}(c)$ if c is uninstantiated.

In the latter case, T is determined by direct constraints on c in the definition of the principle in which the goal `cat`(X,c) occurs. For example, the value of c may be constrained through $\overline{\text{X}}$-theory using a goal such as `head`(c).

Definition 73 (`coindex`)

Given two constituents X and Y, `coindex(`X`,`Y`)` first assigns indices to X or Y if they have not already been indexed. Next, it makes their indices identical wrt. `sameIndex` [Defn. 88].

Definition 74 (`complement_of`)

The following is a recursive definition for extracting complements from a non-head projection. Given a constituent X, Y `complement_of` X holds if:

a. Y is an immediate constituent of X st. Y is a complement of X according to $\overline{\text{X}}$-theory if X immediately dominates its head, or

b. Y `complement_of` X^i where X^i is an immediate constituent of X st. X is a projection of X^i, if X does not immediately dominate its head.

The following type definition applies to the second argument:

Y `complement_of` $X : T : R$ holds $\Rightarrow T = U, R = \{\neg\text{ec}, \neg\text{head}\}$,
i.e. X cannot be an empty category or a head.

Definition 75 (`consFeature`)

Given a constituent X and a feature F, `consFeature(`F`,`X`,`X'`)` creates a new constituent X' identical to X — the only difference being the addition of F to the feature set of X'.

Definition 76 (`ec`)

Given a constituent X, `ec(`X`)` holds if X has no sub-constituents, i.e. X is an empty category. The type definition for X is given by:

`ec(`$X : T : R$`)` holds $\Rightarrow T = U, R = \{\text{ec}\}$.

Definition 77 (`has_constituent`)

Designed to pick out an immediate constituent. Given a constituent X, X `has_constituent` Y holds if X immediately dominates the set of constituents C_X st. $Y \in C_X$.

The type of the first argument is described by the following definition:[1]

$X : T : R$ `has_constituent` Y holds $\Rightarrow T = U, R = \{\neg\text{ec}\}$.

[1] Obviously, a stronger (more constrained) definition can certainly be written if we make use of $\overline{\text{X}}$-theory to relate the types of X and Y. However, this was found to be unnecessary for the current set of principle definitions.

Definition 78 (`has_constituents`)

Given a constituent X, X **`has_constituents`** C_X holds if C_X is the set of constituents that X immediately dominates.

The type of the first argument is given by the following definition:[2]

$X : T : R$ **`has_constituents`** Y holds $\Rightarrow T = U, R = \{\neg\text{ec}\}$.

Definition 79 (`has_feature`)

Feature membership: given a constituent X (with associated feature set F_X) and an instantiated feature F, X **`has_feature`** F holds if $F \in F_X$.

Here is the definition for inferring the type of the first argument:

$X : T : R$ **`has_feature`** F holds $\Rightarrow$
$T = \{c | \ \forall c' \in V_T$ st. $\exists$ lexical entry E with feature set F_E st. $F \in F_E, c'\text{proj}^*c\}$ provided F is a lexical feature, else
$T = \text{typeValue}(F)$ if (non-lexical) feature F has given type, otherwise
$T = U$.
$R = \emptyset$ in each case.

Negation (`\+`) is handled specially:

`\+` $X : T : R$ **`has_feature`** F holds $\Rightarrow$
$T = \{c | \ \forall c' \in V_T$ st. $\exists$ lexical entry E with feature set F_E st. $F \notin F_E, c'\text{proj}^*c\}$ if F is a lexical feature, else
$T = U - T'$ where $T' = \text{typeValue}(F)$ if F has given type, otherwise
$T = U$.
$R = \emptyset$ in each case.

Definition 80 (`head_of`)

The following is a recursive definition for extracting a head. Given a constituent X, Y **`head_of`** X holds when:

a. Y is X if X is a head or has no sub-constituents, otherwise

b. Y **`head_of`** X^i where X^i is an immediate constituent of X st. X is a projection of X^i.

A type definition is provided for the second argument in terms of the first. An additional (independent) equation is provided for the first:

---

[2] Also see note 1.

$Y : T_Y : R_Y$ `head_of` $X : T_X : R_X$ holds $\Rightarrow$
$T_X = \{c|\ \forall c' \in T_Y, c'\text{proj}^* c\}, R_X = \emptyset.$
$T_Y = \{c|c \text{ is a head category label }\}, R_Y = \emptyset.$

Definition 81 (`headOfChain`)

Given a constituent E_1, `headOfChain(`E_1`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_n \rangle, \quad n > 1.$

Definition 82 (`index`)

Indexing: given a constituent X, `index(`X`,`I`)` holds if:

a. I is the index assigned to X if X does not have an index, otherwise

b. I is (or becomes) the index of X if X has already been indexed.

(This may also be used to return the index of a constituent.)

Definition 83 (`inheritFeature`)

Given two constituents X and Y with feature sets F_X and F_Y, respectively, and a feature F, then `inheritFeature(`X`,`F`,`Y`)` updates F_Y to be $F_Y \cup \{F\}$ provided $F \in F_X$. (F_Y is not updated when $F \notin F_X$.)

Definition 84 (`intermediateTrace`)

Given a constituent E_i, `intermediateTrace(`E_i`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_i, \ldots, E_n \rangle, \quad n \geq 3, 1 < i < n.$

Definition 85 (`partOfChain`)

Given a constituent E_i, `partOfChain(`E_i`)` holds if there exists a non-trivial chain $\langle \ldots, E_i, \ldots \rangle, \quad 1 \leq i \leq n$, where n is the length of the chain.

Definition 86 (`partOfSameChain`)

Given two constituents X and Y, `partOfChain(`E_i`)` holds if there exists a non-trivial chain $\langle E_1, \ldots, E_n \rangle, \quad n > 1$ such that X and Y are members. (Note that two constituents with identical indices need not be part of the same movement chain.)

Definition 87 (`projection_of`)

Given two constituents X and Y with category labels c_X and c_Y, respectively, X `projection_of` Y holds if $c_Y\text{proj}^* c_X$, and X and Y share feature sets.

A type definition in terms of the second argument is provided for the first argument:[3]

[3] Obviously, a symmetrical definition can be written for T_Y in terms of T_X. One is not provided to prevent 'race' conditions from occuring during type computation: see chapter 6 for details.

$X : T_X : R_X$ `projection_of` $Y : T_Y : R_Y$ holds $\Rightarrow$
$T_X = \{c | \forall c' \in T_Y, c' \text{proj}^* c\}, R_X = \emptyset$.

Definition 88 (`sameIndex`)

Given two indices I and J, `sameIndex(`I`,`J`)` holds if I and J are identical.

Definition 89 (`specifier_of`)

Given a maximal projection X, Y `specifier_of` X holds if Y is an immediate constituent of X st. Y is an specifier of X.

The following type definition applies to the second argument:

Y `specifier_of` $X : T : R$ holds $\Rightarrow T = U, R = \{\neg\text{ec}, \neg\text{head}\}$.

B.2 Macros

In this section, we define the macros used in defining principles. For each macro we first describe its syntax, and then present its expanded form (or expanded forms). Two macros, `in_all_configurations` [Defn. 90] and `compositional_cases_on` [Defn. 92], are particularly noteworthy as they define parser operations. These operations form the basic unit of principle ordering and interleaving. Hence, each of these two macros will have two possible expansions according to whether the principle being defined is to be interleaved or not. (This multiple-targeting is not necessary for the other (subsidiary) macros.) Note that no examples are provided in this section: for examples of principle definitions using these macros, see chapter refch:rep.

Definition 90 (`in_all_configurations`)

```
:- <Operation-Name> in_all_configurations <Variable>
      where <Pre-conditions> then <Conditions>.

:- <Operation-Name> in_all_configurations <Variable>
      where <Pre-conditions-1> then <Conditions-1>
      else <Pre-conditions-2> then <Conditions-2>...
```

(`<Operation-Name>` names the principle to be defined. `<Variable>` names the configurations over which the principle operates. Arbitrary combinations of Prolog goals may be substituted for all (pre-) conditions.) Roughly speaking, the first form may be read as:

> *"In every configuration (named by* `<Variable>`*), as defined by* `<Pre-conditions>`*,* `<Conditions>` *must hold."*

By using a variant of the 'if–then–else' construct, the second form allows alternative conditions to be placed on configurations. In either case, the `in_all_configurations` macro expands into two different Prolog forms. The first form is used when the principle is not interleaved. Macro expansion will generate a predicate `<Operation-Name>` with arity 1 that recursively descends a given phrase structure and imposes the specified conditions every time its pre-conditions are satisfied. This 'tree-walker' has the following general form:

```
<Operation-Name>(At) :- atomic(At). % case 1: leaf node.
<Operation-Name>([Head|Tail]) :-    % case 2: list of nodes.
        \+ cat([Head|Tail],_),
         !,
        <Operation-Name>(Head),
        <Operation-Name>(Tail).
<Operation-Name>(<Variable>) :-     % case 3: single node
        <Pre-conditions>,           % satisfying
        !,                          % pre-conditions.
        <Conditions>,
        <Variable> has_constituents Cs,
        <Operation-Name>(Cs).       % keep descending...
        etc...
<Operation-Name)(X) :-              % case 4: catch-all for
        X has_constituents Cs,      % single node.
        <Operation-Name>(Cs).
```

(The general form of the third clause is repeated as many times as necessary for alternative conditions in the definition.)

The second form is used when the principle is interleaved. In this case, it is not necessary to use a tree-walker to find the appropriate constituents to impose the conditions. When an appropriate LR reduce action (as defined by the type inference algorithm) is encountered during parsing, the newly-built phrase will be automatically sent to this form for evaluation. Hence, the interleaved predicate will take just one argument and will simply be composed of the 'core' if–then–else construct:

```
<Operation-Name>(<Variable>) :-
        <Pre-conditions>
        -> <Conditions>
        ;
        alternative conditions...
        ; true.
```

(Note that the type of the principle being defined is given by the type of `<Variable>`.)

The only exception to this 'chain' of conditions occurs when the interleaver is able to separate out (to a greater or lesser extent) the alternatives according to different equivalence classes of reduce actions. (See chapter 6 for a detailed description of this process.) In such cases, the interleaved predicate will have one clause for each equivalence class. Thus, each clause will contain a 'reduced chain', that is, an (order-preserving) subsequence of the various alternatives present in the original definition.

Definition 91 (`collect`)

```
:- <Name> collect <Variable-1>
      in_all_configurations <Variable-2>
         where <Conditions>.
```

This form is very similar to `in_all_configurations` [Defn. 90]. The only major difference is that, instead of imposing conditions on constituents that meet certain pre-conditions, this form is designed to collect a list of elements (`<Variable-1>`) from each constituent (`<Variable-2>`) satisfying those (pre-) conditions. For example, with an appropriate instantiation of `<Conditions>`, it can be used to collect, say, all traces or all indices from noun phrases in a given structure.

Its macro expansion is virtually identical to the non-interleaved translation for `in_all_configurations`. The basic difference is the presence of an additional argument that returns the elements being collected. Because of the similarity, we omit the description of the expanded form.

Definition 92 (`compositional_cases_on`)

```
:- <Operation-Name> produces <Items-1>
      compositional_cases_on <Variable> where
        <Variable> with_constituents <C-1> and <C-2> st
                     <C-1> produces <Items-2>,
                     <C-2> produces <Items-3>,
                     <Variable> produces <Items-4>
        then <Conditions-1>
      else <Pre-conditions> then <Conditions-2>...

:- <Operation-Name> produces <Items-1>
      compositional_cases_on <Variable> where
        <Variable> with_constituents <C-1> and <C-2> st
                     <C-1> produces <Items-2>,
                     <C-2> produces <Items-3>,
```

```
                  <Variable> produces <Items-4>
       then <Conditions-1>
     finally <Conditions-2>
     else <Pre-conditions> then <Conditions-3>...
```

(<Operation-Name> names the principle being defined. <Variable> names the configurations over which the principle operates. Arbitrary combinations of goals may be used for the conditions <Conditions-*i*> (and any pre-conditions.) Similarly, arbitrary terms may be substituted for <Items-*i*>.)

This basic form is used to encode the compositional definitions introduced in chapter 3. In a compositional definition, values are computed for each sub-phrase in a structure, with the value for an aggregate phrase being computed in terms of the values of its constituents. To ground the recursion, base cases must also defined. This is encoded in a case-by-case basis using either of the forms above. Roughly speaking, the first form may be read as:

> "<Operation-Name> *computes* <Items-1> *as follows:*
>
> 1. *Constituent* <Variable> *consists of two sub-constituents* <C-1> *and* <C-2>. *Let* <C-1> *and* <C-2> *have values represented by* <Items-2> *and* <Items-3>, *respectively. Then the value of the aggregate represented by* <Variable>, *namely* <Items-4>, *may be computed from* <Items-2> *and* <Items-3> *using* <Conditions-1>, *otherwise:*
> 2. *If the constituent* <Variable> *satisfies* <Pre-conditions>, *then* <Conditions-2> *computes its value, otherwise do nothing."*

Parts 1 and 2 represent the compositional and base cases, respectively. Situations involving more than one base case may be represented by appending extra 'else–then' forms. Although the form of the compositional case only mentions binary branching, the expanded form automatically handles general n-ary branching through iteration. For example, ternary branching will be treated as two instances of binary branching. First, an intermediate value is computed (using <Conditions-1>) for the two leftmost branches. Next, the value for the whole branching is computed from the intermediate value and the value for the rightmost branch. The only difference between the first and second form is that the latter allows extra conditions (via the `finally` form) to be invoked once all branches have been collected in this manner.

As for `in_all_configurations`, there will be two possible macro expansions corresponding to the cases where the operation is interleaved or not. In the non-interleaved case, macro expansion will generate a tree-walker that recursively descends a given phrase structure and applies the compositional case whenever branching occurs, and the base cases for all other constituents. The values for each sub-phrase will be computed in a bottom-up manner. Two predicates will be defined:

one to do the iteration for general n-ary branching, and one to apply the base cases. The general form is as follows:

```
<Operation-Name>(<Variable>|<Items-1>) :- % Compositional
        <Variable> has_constituents Cs,   % case.
        !,
        <Operation-Name-1>(Cs|<Items-1>).
        [<Conditions-2>]                  % Optional
                                          % finally form.
<Operation-Name>(<Variable>|<Items-1>) :- % Base case.
        <Pre-conditions>,
        !,
        <Conditions-2>.
        etc...
<Operation-Name>(<Variable>|<Null-Items>).% Catch-all case.
```

(The general form of the second clause is repeated as many times as necessary for alternative conditions in the definition.) The values maintained for each sub-phrase are returned as extra arguments to the predicate. For the last clause which catches any constituents not satisfying the pre-conditions of any of the defined cases, by convention, these values (<Null-Items>) are set to the empty list. The occurrence of <Operation-Name-1> in the first clause refers to the second predicate which has the following general form:

```
<Operation-Name-1>([]|<Null-Items>).          % Empty list case.
<Operation-Name-1>([Head|Tail]|<Items-4>) :-  % Non-empty
        <Operation-Name-1>(Tail|<Items-3>),   % list case.
        <Operation-Name>(Head|<Items-2>),
        <Conditions-1>.
```

This predicate is designed to iterate over a list of sub-constituents applying <Conditions-1> to compute the value of <Items-4> from <Item-2> and <Item-3>. Also, by convention, the values for the empty list are defined to be null.

The second form is used when the operation is to be interleaved. In this case, each time an appropriate reduce action is encountered, the newly constructed phrase will be passed to this form. If the phrase contains branching, then <Conditions-1> must be applied (possibly repeatedly) to compute the values for the new phrase from its sub-constituents. On the other hand, if the phrase is simple, then it must be tested against the base cases. In either case, the values computed must be 'saved' somehow — so that the next time the operation is called with a larger phrase, it can make use of values already computed for its sub-constituents.[4] In fact, the macro expansion for interleaving is virtually identical to that for the non-interleaved case, except that every time the values for a constituent are requested, it tests to see whether those values have already been computed — in which case, the saved values are used. Otherwise, it computes the values in the regular manner using `<Conditions-1>`. This is implemented by having an extra (initial) clause for `<Operation-Name>` that performs the memory retrieval function. Because of the similarity of the two forms, we omit the description of the second form.

B.3 Type Composition Rules

In this section, we define rules for propagating types in Prolog programs. For example, inference rules will be given for the conjunction, disjunction, and negation of goals.

In the following definitions, let G, G_1 and G_2 be Prolog goals. Let X be a variable. Let '`,`', '`;`', '`:-`', and '`\+`' be the conjunction, disjunction, conditional, and negation operators, respectively. Let U be the universal type.

Definition 93 (Conjunction)

$$\frac{X : T_1 : R_1(G_1), \qquad X : T_2 : R_2(G_2)}{X : T_1 \cap T_2 : R_1 \cup R_2(G_1 \texttt{,} G_2)}$$

The above rule is meant to be read as: "If X has type T_1 and restriction R_1 in G_1, and if X has type T_2 and restriction R_2 in G_2, then we can infer that X has type T_1 intersect T_2 and restriction R_1 union R_2 in (G_1 , G_2)."

Definition 94 (Disjunction)

$$\frac{X : T_1 : R_1(G_1), \qquad X : T_2 : R_2(G_2)}{X : T_1 \cup T_2 : R_1 \cap R_2(G_1 \texttt{;} G_2)}$$

Definition 95 (Negation)

$$\frac{X : T : R(G)}{X : U : R_1(\backslash{+}G)}$$

[4]The interested reader is referred to chapter 6 for the discussion on how values are actually preserved between invocations.

where R_1 is defined to be $\{\neg r | r \in R\}$. Note that X has universal type after negation — no matter what its original type is. This is a conservative rule which essentially provides no type information. Note however, that exceptions are defined for certain goals. See the definition of the primitive `has_feature` [Defn. 79] for an example where type information is not deleted across negation.

Definition 96 (Predicate Definition)

Let predicate P be defined by n clauses of the form $H_i\,\texttt{:-}\,G_i, \quad 1 \leq i \leq n$. Let G be a goal and let σ_i be a unifying substitution for H_i wrt. G. Then:

$$\frac{\forall i \text{ st. } \exists \sigma_i \text{ st. } G = H_i\sigma_i \quad X : T_i : R_i(H_i\,\texttt{:-}\,G_i)\sigma_i}{X : \bigcup_i T_i : \bigcap_i R_i(G)}$$

Definition 97 (Single Clause)

Let $H\,\texttt{:-}\,G_1$ be a clause. Let G be a goal and let σ be a unifying substitution for G_1 wrt. G. Then:

$$\frac{X : T : R(G)}{X : T : R(H\,\texttt{:-}\,G_1)\sigma}$$