A Theory of Syntactic Recognition for Natural Language

by

Mitchell Philip Marcus

A.B., Harvard University (1972)

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

February, 1978

Signature of Author Signature redacted
Department of Electrical Engineering and Computer Science, October 13, 1977

Certified by Signature redacted
Thesis Supervisor

Accepted by Signature redacted
Chairman, Departmental Committee on Graduate Students

A Theory of Syntactic Recognition for Natural Language

by

Mitchell Philip Marcus

Submitted to the Department of Electrical Engineering and Computer Science on October 13, 1977 in partial fulfillment of the requirements of the Degree of Doctor of Philosophy.

Abstract

This dissertation investigates the hypothesis that the syntax of natural language can be parsed by a left-to-right deterministic mechanism without facilities for parallelism or backup. This determinism hypothesis, explored in the context of the grammar of English, is shown to lead to a simple mechanism, a *grammar interpreter*, having the following properties:

-Simple rules of grammar can be written for this interpreter which capture the generalizations behind passives, yes/no questions, and other linguistic phenomena, despite the intrinsic difficulty of capturing such generalizations in the framework of a processing model for recognition, as opposed to a competence model.

-The structure of the grammar interpreter constrains its operation in such a way that, by and large, grammar rules cannot parse sentences which violate either of two constraints on rules of grammar currently proposed by Chomsky. This result depends in part upon the computational use of the notion of *traces* and the related notion of *Annotated Surface Structure*, which derive from Chomsky's work.

-The grammar interpreter provides a simple explanation for the difficulty caused by "garden path" sentences, such as "The cotton clothing is made of grows in Mississippi". Rules can be written for this interpreter to resolve local structural ambiguities which might seem to require nondeterministic parsing; however, the power of such rules depends upon a parameter of the mechanism. Most structural ambiguities can be resolved, given an appropriate setting of this parameter, but those which typically cause garden paths cannot.

To the extent that these properties, all of which reflect deep properties of natural language, follow from the determinism hypothesis, this paper provides indirect evidence for the truth of this assumption.

This paper also demonstrates that the determinism hypothesis necessitates semantic/syntactic interaction to test the *comparative* semantic goodness of differing structural possibilities for an input.

Thesis Supervisor: Jonathan Allen

Title: Professor of Electrical Engineering

To My Parents

Aoknowledgements

I would like to express my gratitude to everyone who contributed to this work, and provided support and encouragement to its author:

-to Jonathan Allen, my advisor, for much good advice, and especially for his constant focus on what is central and what is not;

-to my readers: to Ira Goldstein, for his careful attention during the course of this research, and his careful reading of several iterations of this document, and to Seymour Papert, for helping to narrow the range of the research, and for creating, with Marvin Minsky, an environment of great intellectual intensity and freedom;

-to Bill Martin, who will never let me forget how hard the problem really is;

-to Bob Moore and Chuck Rieger, for long talks spent trying to talk me out of various bad ideas;

-to Mike Genesereth, Gerry Sussman, and Mike Brady, for valuable suggestions at crucial phases of this research;

-to Bill Woods and Susumu Kuno, my undergraduate teachers, who taught me how to look at natural language;

and especially

-to Craig Thiersch, who first introducing me to much of the linguistic theory that underlies this paper, for many long discussions and tutorials on various points of that theory;

-to "The Natural Language Group", Candy Bullwinkle, Kurt VanLehn, Dave McDonald, and Beth Levin, for innumerable suggestions, as well as for helping me to get an education; I also thank Kurt VanLehn for recoding part of the parser, and Beth Levin for allowing me to quote extensively from her working paper;

-to Candy, Kurt, Dave, and especially Beth, and to Chuck Rich, Brian Smith, Bill Martin, and Bob Woodham for unflagging support and enthusiasm;

-and to Susie, with love, for accepting additional burdens while I completed this work and, most importantly, simply for being there, and to Joshua, who kept his father sane without even trying.

Stranger: Then since some will blend, some not, they might be said to be in the same case with the letters of the alphabet. Some of these cannot be conjoined; others will fit together.

Theatetus: Of course.

Stranger: And the vowels are specially good at combination - a sort of bond pervading them all, so that without a vowel the others cannot be fitted together.

Theatetus: That is so.

Stranger: And does everyone know which can combine with which, or does one need an art to do it rightly?

Theatetus: It needs art.

Stranger: And that art is?

Theatetus: Grammar.

-Plato, *The Sophist*
Translated by F.M. Cornford

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Determinism Hypothesis

All current natural language parsers that are adequate to cover a wide range of syntactic constructions operate by simulating nondeterministic machines, either by using backtracking or by pseudo-parallelism. On the face of it, this seems to be necessary, for a cursory examination of natural language reveals many phenomema that seem to demand nondeterministic solutions if we restrict our attention to parsers that operate left-to-right.

By "simulates a nondeterministic machine", I mean the following: In automata theory, a nondeterministic machine is said to accept an input if there is some legal sequence of transitions between machines states which leaves the machine in an accepting state when its input is exhausted, even if there are other legal transition sequences which do not leave the machine in an accepting state. In much the same way, a typical natural language parser will parse a given input if there is some sequence of grammar rules (however such rules are expressed) whose application yields a coherent analysis of the input, even if other legal sequences of rule application do not lead to such analyses. Since all physically existing machines must be deterministic, such a nondeterministic machine must be simulated by causing a deterministic machine to make "guesses" about what the proper sequence of actions for a given input should be. For many inputs, this necessarily leads to the creation of some syntactic substructures which are not constituents in

whatever final syntactic analysis is assigned to a given input.

To see that such an approach seems to be necessary, consider the sentences shown in Figure 1.1.1a and 1.1.2a below. In these two sentences, the phrase "sitting in the box" serves in two very different syntactic roles, as is evident from examining the corresponding declarative sentences 1.1.1b and 1.1.2b. In 1.1.1a "sitting in the box" is the VP (verb phrase) of the major clause, while in 1.1.2a it is a reduced relative clause modifying "the block".

(1a) Is the block sitting in the box?
(1b) The block is sitting in the box.

(2a) Is the block sitting in the box red?
(2b) The block sitting in the box is red.

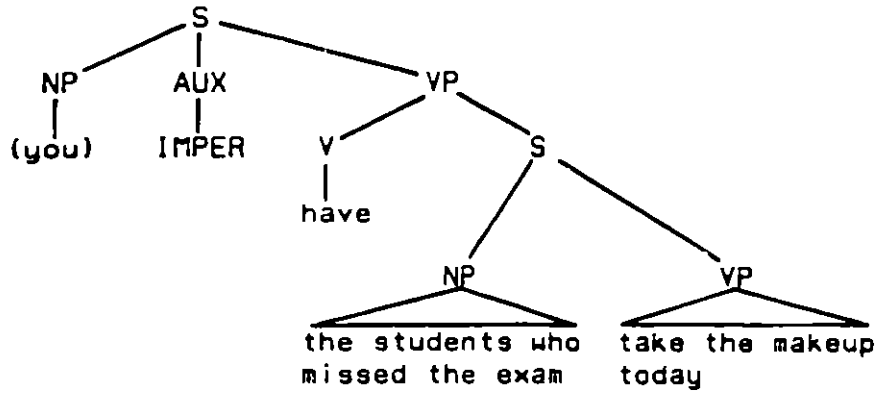
Figure 1.1 - An example which seems to require nondeterministic parsing.

It would seem that to analyze the structure of these sentences in a left-to-right manner, a parser must necessarily simulate a nondeterministic process. Not only is it impossible to determine what role the word "sitting" serves in either 1.1.1a or 1.1.2a upon first encounter, but the two structures are identical up to the end of the phrase "sitting in the box". There is no possible way to tell the two structures apart until it becomes clear whether or not the words following this phrase can serve as the predicate of the main clause.

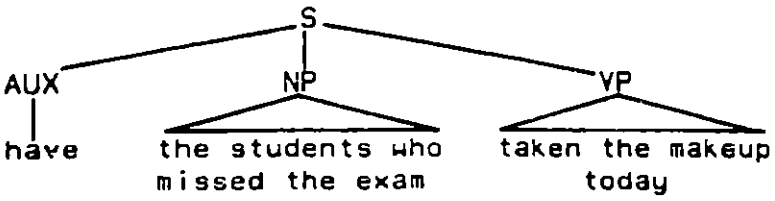
For another example of such an ambiguity, consider the following sentences:

1.2.a Have the students who missed the exam take the makeup today.
1.2.b Have the students who missed the exam taken the makeup today?

While the first seven words of both of these sentences are identical, their structures, which correspond to Figures 1.3.a-b are very different. In 1.2.a "have" is the main verb of an imperative sentence, with the rest of the sentence a subordinate clause. 1.2.b is a question, with "have" as an auxiliary of the main verb "taken", and "the students" as the subject of the main clause. Again, it would seem necessary to use some sort of nondeterministic process to analyze these sentences, since the first clue as to the correct structure appears far into the sentence.



(a) - The structure of an imperative starting with "have".



(b) - The structure of a question starting with "have".

Figure 1.3 - Similar word strings can have very different structures.

While these examples appear to be quite compelling, it is my belief that natural language need not be parsed by a mechanism that simulates nondeterminism, that the seeming necessity for such an approach is an illusion. This paper is based upon this hypothesis; the central idea behind the research reported here is that *the syntax of any natural language can be parsed by a mechanism which operates "strictly deterministically" in that it does not simulate a nondeterministic machine*, in a sense which will be made precise below. (Actually, this paper will investigate this hypothesis only for the particular case of English; nothing will be said about languages other than English in what follows below.) I begin by assuming that this hypothesis is true, and then pursue the consequences of this assumption.

It should be made clear at the outset that I intend to hypothesize only that the *syntactic component* operates strictly deterministically; as will be discussed in Chapter 10, there is a clear necessity for a strictly deterministic parser to ask questions of semantic/pragmatic components which by their very nature involve a limited amount of semantic parallelism. (Some of these semantic tests are *comparative* tests which involve the production of two competing structures, one of which will ultimately be discarded. Because these tests cannot be nested, however, the process is not combinatoric.) Figure 1.4 below is intended to delimit the scope of the Determinism Hypothesis; it should give an indication of exactly what phenomena I believe to be purely syntactic, what phenomena I believe is primarily syntactic with some semantic interaction, etc. The reader should note that this figure is intended more to convey an overall *gestalt* than to make any claims about specific phenomena. Many of the phenomena

mentioned here are discussed in this paper and have been investigated within the framework of the Determinism Hypothesis; others are included here for the sake of filling out the global framework. (Exactly what phenomena have not been investigated will be discussed later in this chapter.)

Phenomena handled by:

Syntactic Processing

"Basic" Clause Structure
NP Movement:
Passivization, NP Preposing with "Seems", etc.
NP Lowering into Complements (i.e. "Raising")
...

There-Insertion
Yes/No-Questions
Imperatives
Detecting Complement Constructions
"Deletion" of Specific Lexical Items
Reestablishing Subjects of some Complements
Some Lexical Ambiguity
...

Syntactic Processing with Limited Semantic Interaction

"Gap Finding" in Wh-Questions, Relative Clauses, "Tough Movement", etc.
Noun-Noun Modification
Some Comparatives
Some Lexical Ambiguity
...

Syntactic and Extensive Semantic Processing

PP Attachment
Conjunction
Reestablishing Subjects of Some Complements
(those with "delta" subjects)
Verb Phrase-Deletion
Some Comparatives
Ellipsis
...

Primarily Semantic Processing

Pronominalization
...

Figure 1.4 - An overview of "linguistic" processing.

It should also be made clear at the outset that this document will not attempt to *prove* that this hypothesis is true, i.e. I will not offer a proof that there are no constructions in English that violate this hypothesis. Instead, I will demonstrate that this assumption, which will be referred to as the Determinism Hypothesis, leads directly to a simple mechanism, a grammar interpreter, which has the following properties, among others:

-The grammar interpreter allows simple rules to be written which elegantly capture the significant generalizations behind such phenomena as passives, yes/no questions, imperatives, and sentences with existential "there". These rules are reminiscent of the sorts of rules proposed by linguists within the framework of the theory of generative grammar, despite the fact that the rules presented here must recover underlying structure given only the terminal string of the surface form of the sentence. The component of the grammar interpreter which allows such rules to be formulated follows directly from the Determinism Hypothesis.

-The structure of the grammar interpreter constrains its operation in such a way that only very complex, *ad hoc* grammar rules can parse sentences which violate several of the constraints on rules of grammar proposed within the last several years by Chomsky. (The syntactic structures that the parser creates are by and large *Annotated Surface Structures* of the type currently proposed by Chomsky, complete with traces, although each parse node is also annotated with a set of features *a la* Winograd. This will be discussed at length in Chapter 5.) Most of the structural properties of the

grammar interpreter upon which this result depends are directly motivated by the Determinism Hypothesis.

-The grammar interpreter provides a simple explanation for the difficulty caused by "garden path" sentences, sentences like "The horse raced past the barn fell". In essence, the grammar interpreter allows special diagnostic rules to be written which can diagnose between the alternative cases presented in examples 1.1 and 1.3 above (this is the focus of Chapter 9), but there is a limitation on the power of such rules which follows from a parameter of the mechanism. By appropriately setting this parameter, sentences like those in 1.1 and 1.3 above can be diagnosed, but those which typically cause garden paths cannot. The component of the mechanism which this parameter affects, and which is crucial for this diagnostic process, is the same component that allows the formulation of the linguistic generalizations discussed above. (These garden path sentences clearly disconfirm one possible form of the Determinism Hypothesis which would say that all sentences which are *grammatical according to a purely competence grammar* can be parsed strictly deterministically, but the explanation for such sentences afforded by this model is consistent with a more "psychological" formulation of the hypothesis: that all sentences *which people can parse without conscious difficulty* can be parsed strictly deterministically.)

To the extent that these properties, all of which reflect deep properties of natural language, follow from the Determinism Hypothesis, and in this sense are explained by it, this paper provides indirect evidence for the truth of the hypothesis.

A Note on Methodology

From the properties of the grammar interpreter which this paper will investigate, it should be clear that the theory presented herein is an *explanatory* theory of language, rather than merely a *descriptive* theory. (For the distinction between these see [Chomsky 65]. My central concern is not the particular grammar of English that has been developed in the course of this work, but rather the general properties of the grammar interpreter that enable a grammar of this form to be written. Thus, the central focus of what follows will not be on the particular rules of grammar presented as examples, but rather on the properties of the interpreter that allow rules like these to be written. Some of the grammar I present can be viewed as a formalization of various "perceptual strategies" of the sort suggested by psychologists [Bever 70]. But again, what is important is not the form of these strategies, but rather the properties of the grammar interpreter and its associated grammar language that allow strategies like these to be expressed and to be properly interpreted. Similarly, the discussion of garden paths sentences and Chomsky's constraints will focus centrally on relevant properties of the interpreter and not on particular rules of grammar.

I will also be much concerned with what *must* be the case, given certain assumptions, and I will attempt to show wherever possible that the mechanism I develop *necessarily* has the properties that it does. I will try to argue for the general characteristics of the model on the basis of strict necessity, as opposed to mere efficiency considerations. In the same vein, the grammar language that will be provided for the grammar interpreter is very

restricted; it quite intentionally includes a minimum of computational frills. The idea is to try to write a grammar within the most restricted framework that one can, to try to find the weakest possible machine capable of parsing English consistent with whatever set of assumptions must be made. In this respect, I believe this research to be unique within the field of natural language processing.

Given this focus, this paper will not discuss details of the implementation of the parser presented here at all, and will only discuss particular rules of grammar when relevant to a higher level point. From the point of view of this paper, the implementation of the parser is only of interest as a "scratchpad" on which the model of the grammar interpreter and the rules of grammar discussed here can be tested for adequacy. For exactly the same reasons, the development of a very large grammar was not a focus of this research; instead, the grammar that was developed concentrates on a small number of fairly complex grammatical phenomena and, crucially, on their interaction with one another. The development of a large grammar of English within this framework - a project to be undertaken quite soon - will be an important additional empirical test of the Determinism Hypothesis, but it is a test that would be meaningless unless the model was otherwise shown to have theoretical content.

This is not to say that the parser has not been implemented; indeed, *all of the snapshots of the parser's operation included in this document are taken directly from actual traces of the parser's operation and all of the code for grammar rules presented in this document has been tested in this*

implementation. For those interested, some notes on the implementation are included in Appendices D and E. Included in Appendix D are the current grammar and a list of sentences that exercises the syntactic phenomena that are included in this grammar; the list attempts to show that the grammar is robust enough to handle various complex interactions between these phenomena. This grammar can be viewed as the seed of an attempt to build a large, robust grammar of English.

And again, this is not to say that the grammar interpreter does not have powerful engineering applications, but that these applications are not relevant to the theoretical issues considered here. Indeed, it should be noted that an early version of this parser and grammar have been used as a front end for the Personal Assistant Project at the MIT A.I. Lab [Goldstein & Roberts 77].

This emphasis upon theory as opposed to grammar deserves some discussion; there are several fundamental points which must be made.

First of all, the presentation of a long list of sentences which a given parser handles does not *by itself* substantiate any claims for the theory of grammar or of language which that parser is claimed to embody. If the model has Turing machine power, as most parsers do, it is clear that a grammar of English can be coded for that mechanism - in principle - which will parse not only the range of constructions which have been handled to date, but the full range of English as well. Furthermore, since most of English can be expressed (although not particularly well) within a context-

free model of language, a grammar of English can certainly be coded within mechanisms limited only to the power of push-down automata. (Indeed, the Harvard Predictive Analyzer [Kuno and Oettinger 62] utilized just such a grammar.) In short, the presentation of a list of English sentences, by itself, shows only that some time and effort were devoted to the development of a grammar for that mechanism.

A further problem with the long list of sentences approach to the validation of a theory of parsing is well expressed by Petrick [Petrick 74], one of the first to build a parser which was firmly theory-based, although his comment was originally made in a narrower context:

> ...it is one matter to handle certain restricted types of sentences... and it is quite another matter to handle them in a sufficiently general way to provide for the interpretation of many sentences where their interaction is complex.

These points are usually acknowledged, at least implicitly, by those who present new parsing systems, in that appeal is often made to the notions of "perspicuity" and "extensibility" as the yardsticks against which a given theory of parsing is to be judged. It should be stressed that these notions are good and appropriate yardsticks for a parser when considered from an engineering viewpoint; these attributes are important attributes for any system which is to be used as a tool by others. These attributes are not so useful as yardsticks against which to judge a theory, however. The problem with the notion of "extensible" is that it is typically coupled to the notion of "flexible", and this later notion, while at the core of good engineering, is the antithesis of the notion of theory. The notion of "perspicuity" is problematic because there is no objective measure by which this quality can

be judged. Perspicuity, like the notion of "plausibility" and all other subjective notions of naturalness, is a useful adjunct by which to judge a theory for which one has other evidence, but it cannot stand as a primary yardstick against which a theory can be measured.

The position that I take here is similar to that of Petrick [Petrick 74], who argues that the primary measure of a particular parser as an embodiment of a theory of language is:

> whether that particular way of organizing a ... machine is well suited for meeting the criteria of adequacy which linguists require of models of natural language

which he states is equivalent to:

> the extent to which this model makes possible the expression and explanation of linguistic generalizations.

I take Petrick's position as a starting point, with one important modification:

The justification of a theory must indeed be in terms of its explanatory power (taking this term now in a wider sense than it is used in the theory of generative grammar) and the generalizations and universals of the competence linguists are one important form of explanation. However, any theory of parsing is necessarily a performance theory of language, in Chomsky's use of the term, and clearly *not* a competence theory [Chomsky 65]. To express this same distinction in terms which are perhaps more intuitive, any theory of parsing states how language is to be *processed*, going from the surface string to some underlying representation, while the theory of generative grammar is essentially a "proof checker" which states,

formally, how to decide if some given string of tree structures (or, more precisely, phrase markers) is a legal derivation, in the formal-language theoretic sense of "derivation". As a performance theory, there are criteria of adequacy for a parsing theory which go beyond - as well as include - the criteria which competence linguists impose upon generative theories of language.

Thus, it is true that a parsing theory should attempt to capture wherever possible the sorts of generalizations that linguistic competence theories capture; there is no reason in principle why these generalizations should not be expressible in performance terms. On the other hand, a parsing theory can also be justified in terms of additional criteria including such issues as the adequacy of the theory as a psychological model, and the computational properties of the mechanism. These computational properties can include formal limitations on resource utilization such as the time and space complexity of the computation, as well as less formal characterizations of such properties (e.g. the Determinism Hypothesis), etc. (It should be noted that the tradeoff between time and space bounds cannot make sense within a purely competence model.) Other computational properties can include such issues as the net effect of interactions between various subsystems upon the overall computational complexity of the problem, viewed formally or informally. The point is that a parsing theory should be justified in terms of *both* these sorts of criteria.

The key point to be made, however, is that the search should be a search for universals, even - and perhaps especially - in the performance

domain. For it would seem that the strongest parsing theory is one which says that *the grammar interpreter itself* is a universal mechanism, i.e. that there is one grammar interpreter which is the appropriate machine for parsing *all* natural languages. If this is true, then properties of that machine itself should be reflected in the structure of each language. And if a particular machine could be shown to capture such properties, *and* to have processing behavior which could be confirmed by psychological experiment, such a theory would be a giant leap forward toward an understanding of the cognitive processes which go on in the human mind.

The Notion of "Strictly Deterministic"

In the discussion above, the Determinism Hypothesis was loosely formulated as follows:

> Natural language can be parsed by a mechanism that operates "strictly deterministically" in that it does not simulate a nondeterministic machine...".

The notion of "strictly deterministic" is central to the meaning of this hypothesis; exactly what does it mean?

To avoid any possible misunderstanding, let me state explicitly at the outset that the Determinism Hypothesis cannot mean simply that language can be parsed by a deterministic machine. As noted above, any computational mechanism that physically exists is deterministic in the automata theoretic sense, and thus any process which is specified by an algorithm for such a machine must be deterministic in this sense. From this it follows that any parser, whether it simulates a nondeterministic machine or not, must itself be deterministic. (The reader should note that "deterministic" does *not* mean

non-probabilistic, nor does "nondeterministic" mean probabilistic. A nondeterministic machine, instead, can be conceptualized as a machine which has a magical oracle which tells it the right decision to make at any point at which its course of action is not strictly determined by the input and the state of the machine.)

Rather than attempting to formulate any rigorous, general explanation of what it means to "not simulate a nondeterministic machine", I will focus instead on several specific properties of the grammar interpreter which will be the focus of this paper. These properties are special in that they will prevent this interpreter from simulating nondeterminism by blocking the implementation of either backtracking or pseudo-parallelism. This discussion is not intended to be definitional, but rather to give the reader a better grasp of the computational restrictions which will be embodied in the grammar interpreter, whose structure will be sketched in the next section.

The first crucial property of the grammar interpreter is that all syntactic substructures created by the machine are permanent. This eliminates the possibility of simulating determinism by "backtracking", i.e by undoing the actions that were done while pursuing a guess that turns out to be incorrect. In terms of the sorts of structures that the interpreter creates and manipulates, this will mean that once a parse node is created, it cannot be destroyed; that once a node is labelled with a given grammatical feature, that feature cannot be removed; and that once one node is attached to another node as its daughter, that attachment cannot be broken.

The second crucial property of the interpreter is that all syntactic substructures created by the machine for a given input must be output as part of the syntactic structure assigned to that input. Since the structure the grammar interpreter will assign to a given input expresses exactly one syntactic analysis, this property eliminates the possibility of simulating nondeterminism via pseudo-parallelism. If a machine were to simulate nondeterminism in this manner, it would necessarily create structures for the alternative analyses that would result from each of the paths such a machine pursues in parallel. By insisting that all structure created by the machine must be output as part of a single, coherent analysis, this sort of pseudo-parallelism is prohibited.

(The reader may wonder how such an interpreter can handle true global ambiguity. The answer is that while such a machine can only build one syntactic analysis for a given input, and thus can only represent one interpretation of the input, it can also observe that the input was ambiguous, and flag the output analysis to indicate that this analysis is only one of a range of coherent analyses. This is equivalent to saying "The input is ambiguous, but here is its primary interpretation: ....". Some external mechanism will then be needed to force the interpreter to reparse the input, taking a different analysis path, if the other consistent analyses are desired.)

The third and final crucial property of the interpreter is that the internal state of the mechanism must be constrained in such a way that no temporary syntactic structures are encoded within the internal state of the

machine. While this does not mean that the machine's internal state must be limited to a finite state control (the grammar interpreter to be presented below uses a push-down stack, among other control structures), it must be limited - at least in its use - in such a way that structure is not hidden in the state of the machine.

One important implication of all this is that a grammar for any interpreter which embodies these properties must constrain that interpreter from ever making a mistake, since the interpreter can only correctly analyze a given input if it never creates any incorrect structure. This means that such a grammar must at least implicitly specify how to decide what the grammar interpreter should do next, i.e. it can never leave the grammar interpreter with more than one alternative. Since the grammar interpreter can only correctly analyze a given input if it never creates any incorrect structure, the grammar must constrain the interpreter from ever making a mistake.

This property of any adequate grammar for such a mechanism gets to the heart of what is intended by the notion of "strictly deterministic". Taken by itself, a grammar for a grammar interpreter which can simulate nondeterminism may specify a machine which is truly nondeterministic in that more than one grammar rule may legally apply at any point during the parsing process. It is up to the interpreter for such a grammar to convert it to a deterministic specification either 1) by imposing additional constraints upon the interpretation process (e.g. a gratuitous rule ordering), 2) by providing some algorithm which chooses one of the alternatives at such a

juncture, providing some mechanism for backup, or 3) by intepreting the grammar as a specification of a deterministic, parallel machine, where such a juncture is interpreted as an indication that the process should split into a collection of parallel processes, each attempting one of the alternatives.

On the other hand, the grammar for a strictly deterministic machine must itself specify a deterministic machine, since the grammar must specify exactly what action the interpreter should take at each and every point in the analysis process. The interpreter cannot use some general rule to take a nondeterministic grammar specification and impose arbitrary constraints to convert it to a deterministic specification (unless, of course, there is some general rule which will always lead to the correct decision in such a case). Such a grammar must therefore be *inherently deterministic itself*; it must absolutely determine what is to be done next at each point in the parsing process. In this sense, such a grammar cannot be a competence description of a language; it must completely constrain the performance of the parsing process. (One caveat to the above discussion must be noted: A grammar for a strictly deterministic machine need not be locally deterministic in those cases where either of two rules may apply first, neither of which affects the successive application of the other.)

The Structure of the Parser - Motivation

Taking the Determinism Hypothesis as a given, an examination of natural language quickly leads to a further set of properties which any deterministic grammar interpreter must embody. (*Henceforth, I will use the word "deterministic" to mean "strictly deterministic" in the sense discussed*

above. Please note this usage.) It is easy to show that any such interpreter *must* have the following properties:

1) It must be at least partially data driven, BUT....

2) It must be able to reflect expectations that follow from general grammatical properties of the partial structures built up during the parsing process.

3) It must have some sort of "look-ahead" facility, even if it is basically left-to-right.

To show that each of these properties is necessary, it suffices to show a pair of sentences of English that cannot be distinguished by a mechanism without the given property, but which speakers of English understand without difficulty. The sentences shown in Figure 1.5 below provide crucial pairs for each of the properties listed above.

The parser must:

Be partially data driven.
(1a) John went to the store.
(1b) How much is the doggie in the window?

Reflect expectations.
(2a) I called [$_{NP}$ John] [$_{S}$ to make Sue feel better].
(2b) I wanted [$_{S}$ John to make Sue feel better].

Have some sort of look-ahead.
(3a) Have [$_{S}$ the boys take the exam today].
(3b) Have [$_{NP}$ the boys] [$_{VP}$taken the exam today].

Figure 1.5 - Some examples which motivate the structure of the parser.

As will be demonstrated in the next chapter, a hypothesis driven parser cannot be deterministic, almost by definition, and thus a deterministic parser must necessarily be at least partially data driven. The essence of the problem is that any parser which is purely hypothesis driven, i.e. which is purely top-down, must hypothesize several nested levels of structure before positing any constituents which can be checked against the input string itself.

For example, a top-down parser, newly given an input, might begin by hypothesizing that the input is a sentence. It might then hypothesize that the input is a declarative, and therefore hypothesize that the input begins with a noun phrase. Assuming that the input begins with a noun phrase, it might finally hypothesize that the NP begins with a determiner, a hypothesis which is testable against the input string. At this point, the parser has created structures that correspond to the S and the NP, which will necessarily have to be discarded for at least some inputs. (These structures might be implicit in the state of the machine, but this is simply a matter of how the constituents are represented at this point in the parsing process.) To take a concrete example, even so different a pair of sentences as 1.5.1a and 1.5.1b above cannot be deterministically analyzed by a hypothesis driven parser. The problem, of course, is simply that any hypothesis driven parser must either attempt to parse a given input as a declarative sentence, beginning, say, with an NP, before it attempts to parse it as a question, beginning with an auxiliary, or vice versa. Whatever order the parser imposes upon these two possibilities relative to each other, the clause type attempted first must be at least occasionally wrong. It is clear that if a

parser is to be deterministic, it must look before it leaps.

A deterministic parser cannot be entirely bottom-up, however. (Let me remind the reader once more that by "deterministic", I mean strictly deterministic.) Any parser that is purely bottom-up must initially misparse one of the two sentences given as (2a) and (2b) above. The problem is that the string "John to make Sue feel better" can be analyzed in two different ways: as one constituent that is an infinitive complement, as in (2b), or as two unrelated constituents, as in (2a), with the NP "John" the object of the verb and the phrase "to make Sue feel better" an adverbial "purpose" clause. The difference in structure between (2a) and (2b) can be predicted, however, if the parser can note that "want" typically takes an infinitive complement, while "call" cannot take such a complement. Thus, a deterministic parser must have some capacity to use whatever information and expectations can be gleaned from an examination of the structures that have been built up at any given point in the parsing process. If a parser is to operate deterministically, it must use such information to constrain the analysis imposed on the remainder of the input.

Finally, if a deterministic parser is to correctly analyze such pairs of sentences as (3a) and (3b) above, it cannot operate in an entirely left-to-right manner. As was discussed earlier in this chapter, it is impossible to distinguish between this pair of sentences before examining the morphology of the verb following the NP "the boys". These sentences can be distinguished, however, if the parser has a large enough "window" on the clause to see this verb; if the verb ends in "en" (in the simple case

presented here), then the clause is an imperative, otherwise it is a yes/no question.. Thus, if a parser is to be deterministic, it must have some facility for look-ahead. *It must be stressed, however, that this look-ahead ability must be constrained in some manner; otherwise the determinism claim is vacuous.*

The Structure of the Parser - an Overview

This paper will present a grammar interpreter called *PARSIFAL*, whose structure is motivated by the three principles discussed above. To show the relation between these principles and the structure of PARSIFAL, I will briefly sketch the structure of the parser here; this description will be expanded at length in Chapters 3 and 4. The diagrams presented here are taken from these later chapters; they are intended to give the reader an overall *gestalt* for the parser and contain detail that will not be explained here.

PARSIFAL maintains two major data structures: a push down stack of incomplete constituents called *the active node stack*, and a small constituent buffer which contains constituents which are complete, but whose higher level grammatical function is as yet uncertain. While the length of the buffer will eventually be expanded to five cells in Chapter 8, it will be assumed in the next several chapters that the buffer is only three cells long. As one sort of constituent whose function has not yet been determined, the individual words which make up the parser's input first come to the parser's attention when they are automatically inserted into the buffer as needed in the course of the parsing process.

Figure 1.6 below shows a snapshot of the parser's data structures taken while parsing the sentence "John should have scheduled the meeting.". At the bottom of the stack is an auxiliary node labelled with the features *modal*, *past*, among others, which has as a daughter the modal "should". (This stack grows *downward*, so that the structure of the stack reflects the structure of the emerging parse tree.) Above the bottom of the stack is an S node with an NP as a daughter, dominating the word "John". There are two words in the buffer, the verb "have" in the first buffer cell and the word "scheduled" in the second. The two words "the meeting" have not yet come to the attention of the parser.

The Active Node Stack

S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)
AUX1 (MODAL PAST VSPL AUX) / (BUILD-AUX)
MODAL : (should)

The Buffer

1 : WORD3 (*HAVE VERB TNSLESS AUXVERB PRES V-3S) : (have)
2 : WORD4 (*SCHEDULE COMP-OBJ VERB INF-OBJ v-3s
ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words: the meeting .

Figure 1.6 - PARSIFAL's two major data structures.

These data structures are acted upon by a grammar which is made up of pattern/action rules whose patterns match against the constituents in the buffer and a specific subset of the constituents in the push down stack, as shown in Figure 1.7.a below. These rules are partially ordered by a

priority scheme, and are organized into *packets of rules*. At any given time during the parsing process, the grammar interpreter only attempts to match those rules which are in *active* packets. Any grammar rule can activate a packet by associating that packet with the constituent at the bottom of the active node stack. If a node at the bottom of the stack is pushed into the stack, the active packets remain associated with it, but are only active when that node is again at the bottom of the stack. For example, in Figure 1.6 above, the packet BUILD-AUX is associated with the bottom of the stack, and is thus active, while the packets PARSE-AUX and CPOOL are associated with the S node above the auxiliary. Some rules that make up the packet PARSE-AUX are shown in Figure 1.7.b below. The rules are written in an English-like grammar language called PIDGIN; they will be explained in detail in Chapter 4.

The grammar:
Matched Against The Buffer and the Stack.

Priority	Pattern	Action
	PACKET1	
5:	[ ] [ ] [ ]	--> ACTION1
10:	[ ] [ ]	--> ACTION2
10:	[ ] [ ] [ ] [ ]	--> ACTION3
	PACKET2	
10:	[ ] [ ]	--> ACTION4
15:	[ ] [ ]	--> ACTION5
	PACKET3	
5:	[ ] [ ]	--> ACTION6

THE BUFFER:

1st	2nd	3rd

(a) - The structure of the grammar.

```
{RULE START-AUX PRIORITY: 10. IN PARSE-AUX
[=verb] -->
Create a new aux node.
Label C with the meet of the features of 1st and pres,
        past, future, tnsless.
Activate build-aux.}

{RULE TO-INFINITIVE PRIORITY: 10. IN PARSE-AUX
[="to, auxverb] [=tnsless] -->
Label a new aux node inf.
Attach 1st to C as to.
Activate build-aux.}

{RULE AUX-ATTACH PRIORITY: 10. IN PARSE-AUX
[=aux] -->
Attach 1st to C as aux.
Activate parse-vp. Deactivate parse-aux.}
```

(b) - Some sample grammar rules that initiate and attach auxiliaries.

Figure 1.7 - The structure of the grammar and some example rules.

The parser (i.e. the grammar interpreter interpreting some grammar) operates by attaching constituents from the buffer to the constituent at the bottom of the stack until that constituent is complete, at which time it is popped from the stack. If the constituents in the buffer provide clear evidence that a constituent of a given type should be initiated, a new node of that type can be created and pushed onto the stack; this new node can also be attached to the node currently at the bottom of the stack before the stack is pushed, if the grammatical function of the new constituent is clear at the time it is created. When popped, a constituent either remains attached to its parent, if it was attached to some larger constituent when it was created, or else it falls into the constituent buffer (which will cause an error if the buffer was already full).

This structure embodies the principles discussed above in the following ways:

1) *A deterministic parser must be at least partially data driven.* A grammar for PARSIFAL is made up of pattern/action rules which are triggered, in part, when lexical items or unattached higher level constituents fulfilling specific descriptions appear in the buffer. Thus, the parser is directly responsive to the input it is given.
2) *A deterministic parser must be able to reflect expectations that follow from the partial structures built up during the parsing process.* Since PARSIFAL only attempts to match rules that are in active packets, grammar rules can activate and deactivate packets of rules to reflect the properties of the constituents in the active node stack. Thus grammar rules can easily be written that are

constrained by whatever structure the parser in attempting to complete.

3) *A deterministic parser must have some sort of constrained look-ahead facility.* PARSIFAL's buffer provides this constrained look-ahead. Because the buffer can hold several constituents, a grammar rule can examine the context that follows the first constituent in the buffer before deciding what grammatical role it fills in a higher level structure. This document will argue that a buffer of quite limited length suffices to allow deterministic parsing. The key idea is that the size of the buffer can be sharply constrained if each location in the buffer can hold a single complete constituent, regardless of that constituent's size.

Is This Machine Really Strictly Deterministic?

While I demonstrate above that PARSIFAL embodies the secondary characteristics that follow from the Determinism Hypothesis, it does not necessarily follow that such a machine really does operate strictly deterministically. In this section, I will argue that the grammar interpreter, given the sort of grammar which will be discussed in succeeding chapters, operates in such a way that it does fulfill the three central criteria for a strictly deterministic machine presented early in this chapter.

First of all, once PARSIFAL has completed the analysis of an input sentence, all the syntactic structure it has on hand is part of the output, fulfilling the first of the three determinism criteria. While a perverse grammar could be written which intentionally left structure in the stack or

sitting in the buffer when an analysis was complete, it is clear that the interpreter does not have the sorts of data structures which would easily allow the simulation of a parallel mechanism, which was the original motivation for this restriction.

That PARSIFAL does not destroy syntactic structure in the course of its operation may be less clear, but it is in fact true that the grammar interpreter fulfills this criterion as well. Because the reader may suspect that implicit backup lurks in either the pattern matching process or the simple existence of the buffer, I will discuss these matters in some detail.

The crucial fact about the pattern matching process is that a pattern that triggers on a specific constituent, say an NP or an S, does *not* cause the parser to attempt to parse a constituent of that sort. Instead, to a first approximation, the pattern will trigger only if a constituent of that sort is already in the specified buffer location. This should be contrasted with the "PUSH" arc of Woods' ATN formalism which is exactly and precisely a nondeterministic subroutine call (this will be discussed at length in Chapter 8).

Because the mechanism by which noun phrases are parsed will not be discussed until Chapter 8, I should particularly assure the reader that a pattern that triggers on an NP does *not* cause the parser to attempt to parse an NP, i.e. such a pattern does not amount to a function call to an NP parsing mechanism, much less to a call to any mechanism that may fail, requiring some sort of backup. The question of the formation of NPs can be

factored out from the operation of "clause-level" processes because the process of parsing NPs is totally bottom-up, i.e. totally data-driven, and operates in such a way that it is transparent to clause-level processes. It must be said that the parser may abort the "clause-level" pattern matching process to construct an NP, but this is not initiated by any particular details of the clause-level patterns being matched, nor is any structure lost when the pattern matching process is aborted.

This last point gets at the heart of why the particular sort of pattern matching that PARSIFAL uses does not hide a form of backup, and thus the simulation of a nondeterministic machine. The pattern matching process would necessarily incorporate a form of backup if it involved any actions that had side-effects that had to be undone if a pattern match failed, but this is not the case. The pattern of each rule is actually nothing more than a Boolean predicate on the state of the grammar interpreter; there are, for example, no variables in rule patterns that would force the pattern matcher into a search process to find consistent values for the variables in a given pattern. In this sense, the pattern matching process can be conceptualized as nothing more than evaluating a large conditional expression, an expression of the form "If <predicate1> then <action1> else if <p2> then <a2> else ... if <pn> then <an>". The grammar interpreter can be viewed as operating by first imposing some full ordering on all rules in active packets that is consistent with the partial order imposed by the priorities, forming a large conditional expression from the ordered rules, evaluating that expression and then repeating the process.

(This model is not quite accurate in that the grammar interpreter will cause a lexical item to be input into the buffer if a pattern attempts to examine the contents of a previously empty buffer cell, but this has no effect upon the underlying conceptual structure of the pattern matching process. The crucial fact here is that once a lexical item has been input into a buffer cell, it stays there.)

One last place that the reader may suspect that the equivalent of backup lurks is in the buffer itself. A machine with arbitrary look-ahead would seem to be equivalent to a machine with arbitrary backup. Thus, if PARSIFAL's buffer was arbitrarily long, the claim of strict determinacy would be completely vacuous. If the size of the buffer is limited, the reader may feel, the extent of the backup may be limited, but it remains backup just the same, and thus the machine is simulating a machine which is nondeterministic in some limited sense.

The fallacy with this viewpoint is that it assumes that the test of whether a system is doing backup or not is a question of how much of the input the system can examine at any given point in the parsing process. It is exactly true that advancing an input pointer three tokens into the input string while noting what the tokens are, then backing up three tokens is exactly equivalent to looking ahead three tokens, given that the same computations are allowed in either case. What this viewpoint fails to realize is that the key question as to whether a system is doing backup - in the sense of using this technique to simulate a nondeterministic process - is whether or not the system can either discard structure or ignore some of the

structure that it generates.

If a system can either discard or ignore structure, then it can simulate a nondeterministic machine, and whether the process is called backup or look-ahead is quite irrelevant. As Martin Kay has so aptly put it, a system is doing backup whether or not one considers there to be a "You Are Here" pointer which advances and then backs up in the course of the parsing process.

If, on the other hand, a system neither discards nor ignores any of the structure that it creates, then it would seem that such a machine is not doing backup by doing look-ahead. It cannot simulate what it will know about the next token of its input when it has advanced several tokens further into the input because it cannot build and then discard the structures that would have to be constructed to simulate its state when it advanced those extra steps.

Indeed, it might be a mistake to use the term "look-ahead" at all to describe such a machine. Perhaps a better image might be that of a *window* of a fixed size that scans across the input string. Just as the transition function of a typical deterministic automaton is determined by its current state and the next token of the input string, so PARSIFAL's transition function is determined by its internal state and the next "windowful" of the input string. It is true that each "windowful" for PARSIFAL consists of several constituents, not of several tokens, but the image is essentially the same. The central point is that there is in some sense a simple function that

pairs the initial state of the machine, including the contents of the buffer, with a resultant state, and that this function can be computed without any side effects except for those inherent in the change of state itself.

From a slightly different point of view, PARSIFAL does not operate using look-ahead at all. Instead, it maintains two rather symmetric data structures, one - the stack - for constituents that it is currently attempting to add daughters to, and another - the buffer - for constituents whose daughters it knows, but whose mother must be determined. Just as it is necessary to simultaneously hold onto many nodes whose daughters have not yet been fully determined, so it is necessary to hold onto several nodes whose mothers have not yet been determined. To determine the mother of a given node in the buffer or the next daughter of the node at the bottom of the stack, it turns out to be more useful to examine more of the nodes in the buffer than the nodes in the stack, but the differences in the functional role of the two structures are not so great as they first appear.

The Limits of This Research

To conclude this chapter, it is appropriate to list exactly which of the phenomena listed in Figure 1.4 above will *not* be discussed in this paper:

-*Comparatives* will not be discussed. The syntax of these constructions is most complex; for a most insightful discussion, see [Bresnan 73].

-Only some of the phenomena included as purely syntactic in that figure will be discussed; most of the phenomena listed there have been

incorporated in the existing grammar for this parser, but are not particularly relevant to the more general points to be discussed in this paper. As noted above, the existing grammar does not have extensive coverage; instead it handles a range of fairly difficult phenomena, and is intended to robustly handle all interactions between these phenomena. (I should note that though the grammar is very small, the range of "clause-level" constructions handled by this grammar, including, in particular, complement constructions, exceeds the coverage of either the LUNAR or SHRDLU grammars. This is because both of these grammars concentrate primarily upon "phrase-level" phenomena.

-No phenomena listed in Figure 1.4 above as requiring "extensive semantic" processing, such as *conjunction*, or *PP attachment* will be discussed. These phenomena await future work; I believe that they will require mechanisms outside of the basic grammar interpreter presented here. It should be noted that Woods does not handle these two phenomena within the framework of his ATN mechanism itself, but rather he posits special purpose mechanisms to deal with them. I share his intuitions.

-I will not discuss *lexical ambiguity*, but rather will focus on *structural ambiguity*. An ambiguity is structural when two different structures can be built up out of smaller constituents of the same given structure and type; an ambiguity is lexical when one word can serve as various parts of speech. While part of the solution to this very pervasive problem seems to involve notions of *discourse context*, I believe that much of the problem can be solved using the techniques presented in this paper, but have not yet investigated this problem.

-While not included in Figure 1.4 above, it is appropriate to note here that I will not discuss the psychological difficulty caused by *center embedded sentences* such as "The rat the cat the dog chased bit ate the cheese.". For a recent theory of this phenomena which is consistent with the model presented here, see [Cowper 76]. (This paper includes an excellent critique of why any parsing model which operates in a purely top-down, i.e. purely hypothesis driven, manner cannot account for this phenomena.)

The Structure of this Paper

This paper consists of a central core, followed by a series of topics. Each of these topics investigates some related set of insights into the structure of natural language that follow from the Determinism Hypothesis and the structure of the grammar interpreter. While the topics are not totally independent, the reader should be able to read some subset of them, if he so chooses, with little difficulty. Whereever there are interdependencies between topics, they are clearly indicated in the text.

The following is a rough outline of the contents of the rest of this paper:

-Chapter 2 provides a historical perspective on this research. It also documents the motivations that led to this work.

-Chapters 3 and 4 present in detail the grammar interpreter itself. These chapters are prerequisite for understanding the chapters that follow.

-Chapters 5, 6 and 7 discuss the implications of the Determinism Hypothesis and the structure of the grammar interpreter *vis a vis* the theory of generative grammar. Chapter 5 demonstrates that the grammar interpreter makes possible the formulation of individual rules of grammar that have much the same descriptive power and formal elegance as the transformations of the theory of generative grammar. Chapters 6 and 7 show that many of the phenomena that are accounted for by the constraints of Chomsky's current theory of grammar fall out from the structure of the grammar interpreter, suggesting that these purported linguistic universals have their roots in a performance theory of language.

-Chapter 8 extends the core of the paper, extending the grammar mechanism to parse noun phrases in a manner transparent to "clause-level" processes. This chapter is prerequisite to understanding Chapter 9.

-Chapter 9 turns to a purely performance phenomenon, the perceptual difficulty caused by so-called "garden path" sentences. This chapter presents a theory to account for why it is that these sentences cause difficulty when read.

-Chapter 10 demonstrates that one consequence of the Determinism Hypothesis is the necessity for certain types of semantic/syntactic interactions. This chapter investigates the interactions necessary to allow deterministic "ungapping" of wh-questions and relative clauses.

-Chapter 11 summarizes what has gone before, and points the way to future research.

CHAPTER 2

HISTORICAL PERSPECTIVE - A CRITIQUE OF HYPOTHESIS-DRIVEN PARSING

Introduction

As was discussed early in Chapter 1, it would seem that the syntax of natural language must be parsed by some mechanism which simulates a nondeterministic machine. This chapter will present the "classical" solution to this problem, the paradigm of top-down, hypothesis-driven search, and will discuss some of the inefficiencies that are inherent in this paradigm. The point of this chapter is not only to provide historical and intellectual perspective, but also to give the reader a feel for exactly where the grammar interpreter to be presented in the immediately succeeding chapters differs from this classical model. The research that underlies this paper resulted originally from the author's frustration in watching just such a system blindly backtracking when it seemed so clear - from the outside - just what the right course of action should be. This chapter is also an attempt to convey exactly where the source of this frustration lies.

The Classical Solution - Hypothesis-Driven Search

While there are many possible computational methods of handling this seeming nondeterminism, almost all of the most powerful existing natural language parsers use approximately the same approach. The STRING system of Naomi Sager [Sager 73], the LUNAR system of Bill Woods [Woods 72], and Terry Winograd's SHRDLU [Winograd 71] all use parsers which can

be characterized as top-down, depth-first search algorithms. To say the same thing a little differently, all three are purely hypothesis-driven, and simulate nondeterminism by providing some sort of backtracking mechanism, a general style which can be termed *guess-and-then-backup*. Since Woods' Augmented Transition Network (ATN) model [Woods 70] provides the clearest model of this paradigm, and is the best known of the three parsers, I will briefly outline its control structure as an exemplar of the entire class. (I will ignore the fact that Woods' ATN, as implemented, is also capable of operating in a top-down pseudo-parallel mode. Pseudo-parallel search is less efficient than depth-first search, if a good *a priori* ordering can be given to the alternatives at each point in the search, which seems to be true of natural language.)

The ATN mechanism can be derived by successive augmentations of the finite state transition graph representation of the finite state machine model. Such a mechanism consists of a set of states, each of which is connected to other states by an ordered set of arcs. On each of the arcs is a test, which specifies the conditions under which this arc should be traversed by the parser, and an action, which specifies what the parser should do when it does traverse the arc. Tests in this model consist of simple predicates on the next word in the input string.

Since finite state machines are computationally too weak to adequately characterize natural language, this finite state model is augmented to yield what Woods calls a *Recursive Transition Network* (RTN). In an RTN, an arc may consist of a call to a sub-network which is an independent

RTN, and this subnetwork can in turn have arcs which call sub-networks. These pushes to sub-networks can be recursive in that a network may have arcs which push to the same network or to other networks which can themselves push to that network. The addition of this recursive mechanism yields a model with the formal power of a context free grammar.

The RTN model is augmented once more by providing sets of registers which can be set and examined by the tests and actions of the grammar, yielding an *Augmented Transition Network* (ATN). The ATN mechanism has Turing machine power, if totally arbitrary functions are allowed as tests and actions on arcs, or it can be limited to recursive computations, if only recursive functions are allowed as tests and actions and no cycles of arcs are allowed that do not advance the input pointer (i.e. no cycles of what Woods terms *jump arcs* are allowed). The resulting ATN model is clearly theoretically adequate to parse natural language, if natural language is parsable by any Turing machine at all.

An ATN grammar, i.e. a particular ATN mechanism, can be viewed in the abstract as accepting a sentence if there is some path through the state network which exhausts the input and ends in a final state. (One action that can be placed on an arc is an operation called BUILDQ which builds a tree fragment out of tree fragments stored in the ATN's registers. An ATN parses while accepting by executing the BUILDQ operations placed on its arcs by the grammar writer.) Such a grammar is inherently non-deterministic, however, in that many arcs at any given state may be traversable at the same time, given the same input. Thus, an ATN grammar, viewed abstractly,

may be said to accept an input if for each state reached during a derivation (however the grammar is interpreted) there is some arc leaving that state which if taken will lead to a final state with the input string exhausted. As stated above, the interpreter of an ATN must simulate this inherently non-deterministic machine with an appropriate computational mechanism, and the mechanism of choice seems to be some variation of a backtracking control structure, which utilizes the *a priori* ordering of the arcs leaving each state to make the search process reasonably efficient.

Given a grammar of the form specified above, an ATN interpreter begins at a distinguished initial state and evaluates the tests on the arcs out of that state in the order that the grammar writer has imposed on the arcs. When a test on one of these arcs succeeds, the interpreter executes the action on the arc and proceeds from the original state to whatever state the arc specifies as a successor state. Before executing the action, however, the interpreter creates a fail-point by pushing a "possibilities list" of the remaining untested arcs onto a stack of untried possibilities.

The interpreter proceeds in this manner until it either exhausts the input string and simultaneously arrives at a final state in the network, thus completing a grammatical analysis, or it reaches a state where there is no arc whose test succeeds. This indicates that the grammatical analysis corresponding to the path that the interpreter was following is in fact incorrect, and that an alternative analysis must be found. The interpreter then turns to the most recently added set of alternatives on the stack of untried possibilities, restores the environment that existed when that fail-

point was created, and proceeds to test the untried arcs from the state the parser has now backed up to. The interpreter proceeds in this fashion until it either completes an analysis or it has backed up through all the stored fail-points, which indicates that the input string cannot be recognized by this particular ATN.

This mechanism can be viewed as operating by picking one of the set of hypotheses which is consistent with both its current state and the next word in the input string and attempting to push this hypothesis through to completion. Since this hypothesis must be at best a guess, the ATN interpreter will back up and choose the most recent alternate hypothesis if it discovers evidence which puts the current hypothesis into doubt. If the *a priori* ordering of the hypotheses with respect to likeliness is accurate, initial hypotheses will often turn out to be right, and large amounts of backup can be avoided.

I believe that it is fair to conceptualize an ATN grammar as a specification of a space of grammatical possibilities which says relatively little about how to search that space. The ordering of arcs, the only explicit control information in an ATN grammar, is quite minimal compared with the arc tests and actions that make up the bulk of the grammar. By and large, an ATN grammar specifies *what* is a grammatical sentence while saying little about *how* to find one of them. It is up to the interpreter to decide *how* to utilize *what* the grammar has to say. In the straightforward top-down depth-first search paradigm I discuss here, the interpreter does this by imposing a general, totally uniform control structure on top of the grammar

that guarantees to exhaust the search space specified by the grammar. (One could speak of this division of knowledge into separate grammar and interpreter components as a kind of *competence* vs. *performance* distinction *a la* Chomsky [Chomsky 65], but I would like to avoid these terms, the source of much confusion, unless they are used in a well defined sense.)

While much of this paper argues, at least implicitly, against dividing knowledge into *how* and *what* components, one important practical advantage of an ATN system is that the writer of an ATN grammar need only worry about specifying the right space of possible grammatical structures for his application; the interpreter frees him of the need to say much about how to search this space. While most earlier parsing systems also have this property (since they too have grammars that specify non-deterministic machines), the ATN notion is historically significant in that it, for the first time, coupled this property with a grammar formalism that seems to be adequate for representing the space of grammatical possibilities of natural language. From a theoretical point of view, the ATN model is significant in that it is an excellent representation for effectively and perspicuously representing the search space of the syntax of natural language. The understanding such a representation engenders is an essential prerequisite to any investigation into what underlying structure there might be that can be utilized to efficiently search the space of grammatical possibilities.

A Critique of Hypothesis-Driven Parsing

One problem with such a scheme is that some sentences, such as the sentences 2.1.1 and 2.1.2 below, repeated from Figure 1.1, will lead the

parser arbitrarily far afield before the parser realizes that something is amiss.

2.1.1 Is the block sitting in the box red?
2.1.2 Is the block sitting in the box?

When this happens, there is no way for the mechanism to know exactly where it made the wrong turn. This is because at any given time the parser is investigating not one hypothesis, but a whole set of hypotheses simultaneously; each time the parser takes an arc from a state and leaves the others untried it is committing itself to an additional hypothesis. When the parser eventually does realize that something is amiss, it may have to back up arbitrarily far until it falls back to a point where its entire set of hypotheses is correct. This is of course true of any backtracking control structure, but the nature of language greatly amplifies the problem because the number of states that a parser must go through to analyze a sentence, and thus the number of hypotheses to which the parser is simultaneously committed is usually very large.

Consider for example, the number of hypotheses that the parser will have to reject if it initially attempts to analyze 2.1.2 above as if it were 2.1.1, i.e. if it tried to analyze "sitting in the box", the predicate of the major clause, as if the participial phrase were a reduced relative clause modifying "the block".

The parse will proceed without problem until the parser reaches the final punctuation mark, with "the block sitting in the box" misparsed as a single NP. The parser will then try to parse this punctuation mark as the predicate of the question, and fail. The parser must now back up to where

it went astray and then find the correct path. In terms of the transition network model, this means that the parser must back up over each arc that it traversed while it analyzed "sitting in the box", one at a time, the most recent first. For each state in the network that the interpreter visited during this analysis, it must attempt to take each of the remaining untried arcs from that state. The interpreter must attempt to find an alternate hypothesis that proceeds from that state and completes the sentence, exhausting the arc set, before it can finally back up to the immediately previous state and repeat the process. This will continue until it has backed up to the state at which it had just finished parsing "block" as the head of the (as yet incomplete) NP "the block".

Specifically, the parser will have to attempt all alternative ways of parsing "the box" as a noun phrase and necessarily fail, since the original analysis was quite correct, before attempting to find some alternative analysis which completes the participial phrase "sitting...". This too will fail, since the original analysis was exactly right here too. The parser must then attempt to find some alternative analysis, starting with "sitting", which will produce a verb phrase completing the reduced relative clause. It will next attempt to find some alternative way to parse a relative clause, and then, finally, discard the hypothesis that "sitting" starts a relative clause at all.

Since the interpreter has no idea where it went astray, it must successively discard the right hypotheses made in parsing subconstituents until it finally discards the higher level hypothesis which was originally in error. (Note, by the way, that the well formed substring table notion does

not make this hypothesis discarding process any easier; keeping track of these substrings helps later, when the parser, pushing forward after the incorrect hypothesis is discarded, would otherwise have to reconstruct many of the valid structures earlier cast aside. The problem is not so much that valid hypotheses are discarded as it is that much effort must be spent in attempting to find some other analysis for what was in fact a correct structure.)

But this is not the only inefficiency of the hypothesis-driven paradigm. While it was implicitly assumed above that the interpreter could always trivially reject bad hypotheses by examining the next word in the input string, this is often not the case. This is because many arcs in the network do not actually attempt to utilize the next word in the input at all, but rather push to a subnetwork that will attempt to build some multi-word constituent, as when the noun phrase network attempts to parse a relative clause by pushing for a sentence. Often this push itself will push for a smaller multi-word constituent, as when the push for a relative clause is itself immediately followed in the sentence network by a push for an NP. Only within the NP network will tests on arcs actually look to the next word in the input string, trying to find a role for it in the hypothesized NP. It may take some effort to discover that the next word in the input string is not consistent with the NP hypothesis at all, and a larger amount of effort to discover, perhaps, that this word is not even consistent with the relative clause hypothesis that gave rise to the NP hypothesis originally. This inefficiency follows necessarily from the fact that the guess-and-then-back-up paradigm is entirely hypothesis driven, i.e. that it is totally top-down.

Thus, such a parser must generate subgoal under subgoal, until a hypothesis is suggested at the bottom of the hypothesis tree that can actually be tested immediately against the input string.

Because of this property, rejecting a high level hypothesis which is quite obviously wrong can take a fair amount of time. This is compounded with the fact that as ATN, being entirely hypothesis-driven, can utilize information about properties of the next word in the input string only by asking questions which will show if the next word is or is not consistent with the next hypothesis enumerated in the grammar. In essence, an ATN must play a game of Twenty Questions with its environment to discover what grammatical possibilities the next word in the input string allows.

Consider the behavior of the parser in the "sitting in the box" example discussed above, at the point when the parser has just completed (mis)parsing "the block sitting in the box" as a single NP. At this point the parser is attempting to parse the final punctuation mark as some sort of predicate. While it is obvious that a final punctuation mark cannot initiate any sort of predicate phrase at all, there are many, many sorts of predicate phrases that it does NOT start, and each of these must be attempted and rejected in turn. The final punctuation mark will of course block every attempted hypothesis reasonably quickly, but a large number of hypotheses must be rejected before the hypothesis that a predicate phrase starts here will finally prove to be untenable.

One patch that may suggest itself to the reader is to put in a test

which explicitly disconfirms this high level hypothesis. Thus, an arc might be added to the grammar which embodies the information that if a predicate phrase is about to be parsed and the next thing in the input string is a final punctuation mark, then immediately reject the predicate phrase hypothesis. One problem with any such patch as a general solution to this class of problems is simply that while there are many ways of beginning a given high level constituent, there are even more ways of not beginning such a constituent, and it would be expensive to test for all the terminal symbols which do NOT start a constituent of a given sort.

Another problem that this, and any such patch, fails to come to terms with is that there is a limitation on the utility of being able to look only one terminal symbol ahead in general. As it turns out, a single symbol often does little to constrain the sorts of constituents that it can begin. It is true that a word like "the" provides solid evidence that a noun phrase must immediately follow in the input string, but many words do not have this property; as Figure 2.2 below shows, for example, the word "as" can initiate as many as five different sorts of constituents: subordinate clauses (2.2.a), quantifiers (2.2.b), adjectives (2.2.c), adverbs (2.2.d), and prepositional phrases (2.2.e):

(a) *As these examples show*, "As" initiates various constituents.
(b) *As many as ten* different explanations were proffered.
(c) No one could ever be *as big as big bad John.*
(d) He left *as quickly as he could.*
(e) Bill offered his advice *as an expert in such matters.*

Figure 2.2 - "As" can initiate many types of constituents.

As Figure 2.3 below shows, even a simple number like "five" can initiate adverbs (2.3.a), adjectives (2.3.b), and of course quantifiers (2.3.c):

(a) Theses get written *five times more slowly than I thought possible.*
(b) He tried to jump a *five foot high* wall.
(c) There are *five* good reasons for not even trying.

Figure 2.3 - Even "five" can initiate many sorts of constituents.

So while the word currently available in the input string can provide some constraint on what hypothesis should be tested next, it provides a much less certain constraint than might appear to be the case at first glance.

Summary

To summarize the above discussion, it seems that the inefficiencies of the guess-and-then-back-up paradigm follow mainly from two interrelated properties of the paradigm: 1) this paradigm is totally hypothesis driven, and therefore irresponsive to the actual data; 2) even if the paradigm was modified to make it more data driven, one word does not seem to be an adequate unit of context against which to test the plausibility of any given hypothesis.

These observations suggest several possibilities to improve the efficiency of a natural language parser. If a parser was less hypothesis driven, and somehow more sensitive to the data, perhaps the parser could be kept from posing high level hypotheses that require several levels of subhypotheses before the data can provide any confirmation of the top level hypothesis at all. Perhaps it is somehow possible to get a better indication

of what lies ahead in the input string than simply by looking at the next word, thereby providing a much stronger filter for suggested hypotheses. This might mean, for example, attempting to parse a little bit ahead in the input string to get a better view of what sort of constituent in fact lies ahead. In short, if a parser could be more directly sensitive to the input data, and if that data could be made to reveal solid information about what sort of syntactic context lay ahead, then perhaps these inefficiencies could be overcome. These characteristics, of course, are embodied in the grammar interpreter which was sketched in the last chapter, and which is the central focus of this paper.

CHAPTER 3

THE GRAMMAR INTERPRETER

PARSIFAL's Major Data Structures

With the motivation for the parser's structure behind us, I will focus on the parser's two major data structures in the remainder of this chapter, and the structure of the grammar in the next.

For reasons of exposition, I will assume in this and the immediately following chapters that all noun phrases are constructed by processes that are transparent to the basic grammar interpreter. This will allow the discussion to focus on "clause-level" processes before considering the mechanisms by which noun phrases are constructed. As we shall see in Chapter 8, a straightforward extension of the interpreter described in this chapter will take the magic out of this assumption.

The remainder of this chapter will discuss the two major data structures of the grammar interpreter, the push-down stack of incomplete constituents, and the three place constituent buffer that the interpreter uses as a workspace.

The Active Node Stack

I will call the first-in first-out stack of incomplete constituents the *active node stack*, thinking of the topmost node of each incomplete constituent as being in some sense active. This stack allows the parser to

deal with the well known recursive properties of natural language in much the same way that the stack of a push-down automaton enables that mechanism to deal with the recursive properties of context free languages. The parser operates by attempting to add constituents to the node at the bottom of the stack (thinking of the stack as growing downwards). It will push this incomplete constituent into the stack while it is building the lower level constituents that are its daughters. When a node is completed, it is popped from the stack, and the parser will then continue adding constituents to the node which was immediately above it in the stack. Thus, in general, if one node is immediately under another node in the parse tree, it will be immediately under that other node in the active node stack as well.

(Note, by the way, that the fact that the parser can handle the recursive properties of natural language does *not* entail that the grammar interpreter itself be recursive. In much the same way that the finite state control of a push-down automaton need not be recursively invoked for that mechanism to parse recursively embedded context free languages, there does not seem to be any need for recursive invocation of the grammar interpreter.)

For the sake of illustration, a snapshot of the active node stack is shown in Figure 3.1.a below, with the corresponding completed parse tree shown in Figure 3.1.b. (It should be noted that this figure is simplified for expository reasons; much information contained in the active node stack and the parse tree is not shown in this diagram.) In this snapshot, there are two nodes in the active node stack, the node VP22, at the bottom of the stack,

and the node S20, at the top of the stack above VP22. At the time this snapshot was taken, S20 had 3 daughters attached to it, one of which is VP22, and VP22 had one daughter attached. Since VP22 was at the bottom of the stack at the time this snapshot was taken, the parser was attempting to add constituents to VP22 at that time. Once VP22 is complete, the parser will pop the node from the stack, and continue to add constituents to S20. One can see from Figure 3.1.b that S20 will remain above VP22 in the completed parse tree.

The Active Node Stack

```
| S20                    |
|    NP : (i)            |
|    AUX : (will)        |
|    VP : ↓              |
| VP22                   |
|    VERB : (schedule)   |
```

(a) - A snapshot of the active node stack.

```
S20
    NP47  i
    AUX20
        WORD112 will
    VP22
        WORD113 schedule
        NP50 a meeting
    WORD116 .
```

(b) - The finished parse tree.

Figure 3.1 - The active node stack and a completed parse tree.

There are only two nodes in the active node stack that the parser can modify or directly examine. These two nodes are the bottom node on the stack, which will be referred to as the *current active node*, and the S or NP node closest to the bottom of the stack which will be called the *dominating cyclic node*, following the terminology of generative grammar, or sometimes the *current cyclic node* or, if an S, the *current S node*. (In transformational theory, S and NP nodes are distinguished in that the transformational mechanism is applied cyclically to the constituents under each node of these two types; these nodes are thus called *cyclic nodes*. A node which is above another node in constituent structure is said to *dominate* that node. Thus, I will call the nearest cyclic node above the current active node the *dominating cyclic node*.) In Figure 3.1 above, VP22 is the current active node, and S20 is the current S node.

Besides these two nodes, the parser is also free to examine their descendents, i.e. the nodes they dominate, although the parser cannot modify them. Since some of the descendents of the current cyclic node will most likely still be in the active node stack, the parser can in fact examine more than two nodes in the stack, although in the case of descendents, it cannot tell whether or not they are in the stack. As we shall see later, making the dominating cyclic node explicitly available for examination and modification seems to eliminate the need for any tree climbing operations that ascend tree structures. Empirically, there seems to be no need for the parser to examine any nodes already incorporated into the parse tree except for those which are dominated by the current cyclic node.

Nodes

The primary data type in this parser is the *parse node*. Grammatical structures are represented by tree structures of parse nodes, each node representing one grammatical constituent. Each node in a complete tree is of a given *type* (NP, VP, S, etc.), has a *parent* (except for the root of the tree), a list of *daughters*, and an associated set of *grammatical features*. Each node also has a name by which we can refer to it, a system generated symbol like "NP35", "WORD10", etc.

The set of grammatical features of a node summarize the primary grammatical properties of the constituent it represents. These properties are typically those that are important in deciding the grammatical role of the node (i.e. the node's attachment) in a larger constituent, or that affect the overall grammatical behavior of whatever larger constituent the node itself is attached to. For example, the sorts of complements a given verb takes, i.e. whether its objects are simple NPs, infinitive phrases, that-complements, or the like, greatly affects the grammatical behavior of the VP it is part of, and so features that mark these different options are included in the feature set of the verb. The general idea is that a constituent's grammatical features include whatever properties should be visible at a glance to processes dealing with higher level constituents. In line with this philosophy, the type of a node will also be one of its features. (From this point onwards, I will not distinguish between a node and the constituent that it represents, but will use the two words interchangebly.)

As an example of the use of features to label parse nodes, the tree

shown above in Figure 3.1.b is repeated below in Figure 3.2, with features added. Thus, node S20 is labelled *S*, *DECL*, and *MAJOR*, indicating that it is a declarative, major S. NP47 is labelled as an *NP*, which is singular (*NS*), first person (*N1P*), an NP which dominates a pronoun (*PRON-NP*) and which is therefore not modifiable (*NOT-MODIFIABLE*). Similarly, the auxiliary node AUX20 is labelled as an auxiliary which is future tense, and will agree with any NP subject, either singular or plural (*VSPL*), and the NP NP50 is labelled as a singular, determined, indefinite NP.

```
S20 (DECL MAJOR S)
    NP47  (NS N1P PRON-NP NOT-MODIFIABLE NP)
          I
    AUX20 (FUTURE VSPL AUX)
          WORD112 will
    VP22  (VP)
          WORD113 schedule
          NP50  (NS INDEF DET NP)
                a meeting
    WORD116 .
```

Figure 3.2 - A parse tree showing the features on nodes.

The Buffer

The constituent buffer is really the heart of the grammar interpreter; it is the central feature that distinguishes this parser from all others. The words that make up the parser's input first come to its attention when they appear at the end of this buffer after morphological analysis. After the parser builds these words into some larger grammatical structure at the bottom of the active node stack, it may then pop the new constituent from the active node stack and insert it into the buffer if the grammatical role of this larger structure is as yet undetermined. The parser is free to

examine the constituents in this buffer, to act upon them, and to otherwise use the buffer as a workspace.

In general, the parser uses the buffer in a first-in, first-out fashion (although, as we shall see, there is no necessity that it do so). It typically decides what to do with the constituent in the leftmost buffer position after taking the opportunity to examine its immediate neighbors to the right. The availability of the buffer allows the parser to defer using a word or a larger constituent that fills a single buffer cell until it has a chance to examine some of the right context of the constituent in question. Thus, for example, the parser must often decide whether the word "have" at the beginning of a clause initiates a yes/no question such as in fig. 3.3.a below or an imperative as in fig. 3.3.b. This issue must be resolved before the grammatical role of "have" can be decided because in the first case "have" is an auxiliary verb, while in the second case it is the main verb of the major clause. (A similar example, complete with parse trees, can be found in Chapter 1.) The parser can often correctly decide what sort of clause it has encountered, and thus how to use the initial verb, by allowing several constituents to "pile up" in the buffer. Consider for example, the snapshot of the buffer shown in Figure 3.3.c below. By waiting until the NP "the boys", NP25, is formed, filling the 2nd buffer position, and WORD37, the verb "do", enters the buffer, filling the 3rd buffer position, the parser can see that the clause must be an imperative, and that "have" is therefore the main verb of the major clause. (The details of this diagnostic process can be found in Chapter 9.)

(a) Have the boys done it yet?
(b) Have the boys do it.

WORD32	NP2S	WORD37
HAVE	THE BOYS	DO

(c) - A snapshot of the parser's buffer.

Figure 3.3 - The buffer allows the parser to examine local context.

The Buffer as a Virtual Machine

To make the buffer and the operations associated with it more precise, it will be useful to think of the buffer as a kind of virtual machine. This section will present a general class of mechanisms which will be further constrained in later sections to provide the precise mechanism utilized by this parser. First, a mechanism called a *random-access, compacting buffer* will be presented, and then that mechanism will be extended to yield a *random-access, compacting buffer with offset.* In the course of the discussion, both mechanisms will be referred to merely as *buffers*, as long as the type of mechanism is made clear by context.

The data structure underlying both sorts of buffer is a linear list of length n. By linear list I mean (following Knuth [Knuth 68], but changing terminology slightly to avoid conflicts):

> a set of $n \geq 0$ cells $x[1], x[2], \ldots x[n]$ whose structural properties involve only the linear (one-dimensional) relative positions of the cells: the facts that, if $n > 0$, $x[1]$ is the first cell; when $1 < k < n$, the kth cell $x[k]$ is preceded by $x[k-1]$ and followed by $x[k+1]$; and $x[n]$ is the last cell.

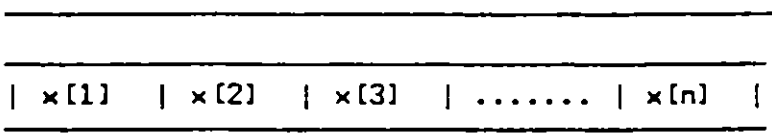


Figure 3.4 - A linear list of length n.

Each cell in the linear list of the buffer mechanism may either be *occupied*, containing data, or it may be *empty*, being free of data. As we shall see below, the operations that are performed on the buffer are such that all the cells in the buffer that contain data are contiguous, beginning with cell x[1]. Thus there will always be a cell x[limit], $1 \leq limit \leq n$, which is the *last occupied cell*, i.e. that for $i \leq limit$, x[i] is occupied and for $limit < j \leq n$, x[j] is empty. For this reason, I will call the buffer mechanism a *compacting* buffer.

There are three operations that the buffer will perform on command. These three operations are:

Read(i):	Return the item in cell x[i].
Delete(i):	Delete the item in cell x[i] and then shift the contents of the cells to the right of x[i] to fill the "gap".
Insert(w,i):	Insert the item w into cell x[i] after shifting left to create a "gap" in the buffer.

Each of these operations will now be described in detail. Pseudo-code for each of these operations is given in Figure 3.5 below. Note that data can be read from, inserted into or deleted from any cell in the buffer. For this reason, I will call the buffer mechanism a *random access* buffer, thus the mechanism described in this section can be termed a *random access, compacting buffer.*

The command Read(i) causes the buffer to return the item in cell x[i] as long as x[i] is occupied, i.e. as long as i ≤ limit, where *limit* is the index of the last occupied cell. If x[i] is not occupied, i.e. i > limit, then the buffer will take successive "unused" items from an *input list* (see below) and insert these items into successive unoccupied cells until x[i] is occupied, at which point its contents are returned.

Associated with the buffer is an auxiliary linear list called an *input list*, into which some external process places the items which are to be input into the buffer. The buffer maintains a pointer into this list, initially pointing to the first element in the list, and inserts the item pointed at into the buffer whenever it needs to fill an unoccupied cell, as described above. It does this using the auxiliary buffer command Input (i), for which pseudo-code is given in Figure 3.5 below. The buffer increments this pointer each time it "uses" an item in the input list. The buffer also has the index of the last occupied cell in the input list; it is an error to cause the buffer to attempt to read past the end of the input list. In the pseudo-code below, the contents of the cells of the input list are the values the *input-list[i]*, the pointer into the input list is the value of *input-pointer*, and the index of the last occupied cell in the input list is *input-limit*.

The command Delete(i) deletes the item in cell x[i] by shifting the contents of all occupied cells to the right of x[i] one cell to the left (thinking of x[1] as the leftmost cell), thus filling x[j] with the data item previously in x[j+1], for i ≤j < limit. I will say that the Delete operation

compacts the buffer to refer to the fact that after the contents of x[i] have been deleted, the previous contents of x[i-1] and x[i+1] are in adjacent buffer cells.

The command Insert(w,i) inserts w into cell x[i] by shifting the contents of x[i] and of all cells to the right of x[i] one cell to the right and then inserting w into x[i].

```
Read (i)
    tag: if i ≤ limit then return x[i]
         if i > n then error
         if limit < i ≤ n then
              Insert (Input (), x[limit+1])
              go to tag

Input
         if input-pointer ≥ input-limit then error
         input-pointer <- input-pointer + 1
         return input-list[i]

Delete (i)
         if i > limit then error
         for k = i + 1 step 1 until k = limit
               x[k] <- x[k+1]
         limit <- limit - 1

Insert (w,i)
         if i > limit + 1 then error
         for k = limit step -1 until k < i
               x[k+1] <- x[k]
         limit <- limit + 1
         x[i] <- w

Initially:
         limit = 0
         input-pointer = 0
         input-limit = {the index of the last occupied cell in input-list}
```

Figure 3.5 - Pseudo-code for the buffer commands.

A Buffer with Offset

While the buffer mechanism presented above will suffice for the construction of the basic parser model that will be presented in the remainder of this chapter, and in the immediately following chapters, the complete parser model will require one further extension of this mechanism. This extension will cause the index given to the Read, Insert, and Delete commands to refer to a cell whose location is offset from the beginning of the buffer by some fixed amount. This offset is specified by a new command Offset(j), which causes the offset parameter *m* to be set to the sum of j plus the old value of the offset parameter. (The offset parameter *m* is initially stipulated to be 0.) After a command of this form has been executed, the command Read(i), for example, will return the item in cell x[i+m] rather than the item in cell x[i]. (This offset mechanism will ultimately be used to allow the parser to "shift its attention" from the beginning of the buffer to the beginning of a suspected noun phrase by offsetting the buffer by an appropriate amount.)

The offset mechanism will in fact be coupled to a *push-down stack* of such offsets, so that offsets may be pushed and popped from the stack. This stack, the third data structure of the full parser model, will be referred to as the *buffer pointer stack*, or, more often, just as the *pointer stack*. The current offset parameter *m* is thus the offset at the bottom of the buffer pointer stack, again thinking of the stack as growing downward. This means that two new commands are needed for this extended buffer mechanism:

Offset(j): Push the sum of j and m_{old}, the current value of the bottom

of the pointer stack, onto the pointer stack, making m_{new} the new offset.

Pop-offset: Pop the current value off the pointer stack, making the previous value pushed onto the stack the current offset. If the top of the stack has already been reached, executing this command causes an error.

By stipulating that the stack is initialized with an offset parameter of 0, this extended mechanism will behave exactly like the buffer mechanism presented above as long as no Offset commands are executed.

Note that the Read, Insert, and Delete commands are constrained even in this extended buffer model to take only positive buffer indices. Thus, if the current offset is m, there is no way to read, insert or delete any item in cells $x[i]$, for $i \leq m$. Furthermore, since no mechanism is provided to examine the pointer stack itself, there is no way to even note whether there is any offset at all, e.g. whether the command Read(1) will return the item in $x[1]$ or some other cell. Because it will often be useful to refer to the effective beginning of the buffer, i.e. the cell whose contents are returned by the command Read(1), I will often use the terms *the effective buffer start* to refer to this cell. When discussing the extended buffer model, this cell will be referred to as "the first buffer cell", rather than the absolute first cell in the buffer; in general, "the *i*th buffer cell" will refer to the cell whose contents are returned by the buffer command Read(i) rather than the cell $x[i]$.

Below in Figure 3.6 is pseudo-code which specifies the operations performed by the three basic commands in this extended model.

Given that m is the value at the bottom of the pointer stack:

```
Read (i)
    tag: if i+m ≤ limit then return x[i+m]
         if i+m > n then error
         if limit < i+m ≤ n then go to tag

Delete (i)
         if i+m > limit then error
         for k = i+m + 1 step 1 until k = limit
               x[k] <- x[k+1]
         limit <- limit - 1

Insert (w,i)
         if i+m > limit+1 then error
         for k = limit step -1 until k < i+m
               x[k+1] <- x[k]
         limit <- limit + 1
         x[i+m] <- w
```

Figure 3.6 - Pseudo-code for the buffer commands given an offset m.

The Buffer as Utilized by the Parser

The virtual machine model presented above is very general; details like the length of the buffer, what the data items in the buffer are, etc. remain to be specified for the buffer mechanism utilized by the parser.

As mentioned above, the complete parsing mechanism will be developed in two steps: the mechanism presented in this chapter assumes that noun phrases enter the buffer fully parsed, the complete parser extends this mechanism in such a way that noun phrases are built in a manner which is transparent to the sorts of "clause-level" processes that will be discussed in this chapter. In the full parser, the buffer is stipulated to be five cells long for empirical reasons that will be discussed in Chapter 8. However, the parser is further constrained such that it can only access the

first three cells after the offset, again for empirical reasons: this is done by stipulating that the parser can only use the indices 1,2,3 in commands to the buffer. The simplified mechanism presented in this chapter differs from the complete model primarily in that the offset mechanism does not come into play. Given the stipulation that only the first three cells after the offset can be accessed, this means that for the time being the reader can think of the buffer as being exactly three cells long.

Each cell in the buffer can hold a *grammatical constituent* of any type, from a single word to a complete subordinate clause. The input list in the parser holds lexical items that have undergone morphological processing, thus, from the point of view of grammatical processing, lexical items enter the end of the buffer on demand. It should be noted that each buffer cell can hold exactly one constituent, no more and no less, where by constituent I mean any tree that the parser has constructed under a single root node. The size of the structure underneath the node is immaterial; both "that" and "that the big green cookie monster's toe got stubbed" are perfectly good constituents once the parser has constructed the latter phrase into a subordinate clause. After the parser has decided that the current active node is complete, it is always free to insert that node into a single buffer cell and then pop the node from the active node stack if it does not know what higher level structure the newly completed current active node is attached to. (This will be discussed in far more detail below.) The crucial point is that once such a constituent has been constructed, the size of the tree under the root node is totally irrelevant.

Building Structure is Irrevocable

The key limitation on what the parser can build as a single constituent and then drop into the buffer is thus not the length of the constituent, but rather the fact that there is no facility within the grammar interpreter for backtracking of any kind. The only way the parser can construct a constituent is to attach all subconstituents to the topmost node of that constituent; there are no registers or other mechanisms that the parser can use to declare the bounds of a constituent while storing its pieces until the parser is sure of their role. The parser must be absolutely sure that the structure it is going to assign to a constituent is correct before building it. Thus, the parser can build a constituent only at the peril of being mistaken about its internal structure. A further restriction on the parser is that feature assignment as well as node attachment is permanent; features can be added to a node but not removed.

This is the sense in which this parser can be said to be deterministic, that the structure assigned to any constituent in terms of subconstituent attachment and feature labelling is final and irrevocable. The ability to buffer constituents potentially gives the parser a great deal of information to use in deciding what higher level role should be assigned to a constituent, but there is no recourse should the parser make a wrong decision.

The Buffer and the Stack Reflect Different Aspects of the Parser's Operation

The reader might wonder why it is that the parser is based upon

two, rather than one, data structures.

Most parsers work in one of two modes: top-down (hypothesis driven) or bottom-up (data driven). For each of these modes, there are data structures that are natural and convenient. Thus, a push-down stack is a natural data structure upon which to base a top-down parser. Such a parser is constantly attempting to add subconstituents to some specific node in the parse tree, which it does by postulating a subconstituent of that node and recursively repeating the process until it postulates as a subconstituent some terminal category that can be checked against the input string. This can be naturally implemented by using a push-down stack, treating the node most recently pushed onto the stack as the node to which subconstituents must be added. For each possible subconstituent of that node, a node of the appropriate type can be pushed onto the stack, and the mechanism, if appropriately constructed, will automatically attempt to complete it in similar fashion. When a node is complete, it can be popped from the stack and attached to its immediate predecessor on the stack, after which the parser will automatically continue to add further constituents to its predecessor.

Similarly, for a bottom-up parser, the data structures which are most natural are those which allow easy access to contiguous constituents. Such a parser typically operates by attempting to incorporate a sequence of contiguous constituents into a higher level constituent, repeating the process until a root node has been constructed that incorporates all the terminals in the input string. (See, for example, the Cocke-Kasami-Younger algorithm [Aho & Ullman 72] and Pratt's elegant modification of that algorithm [Pratt

75].) One of the key operations in such a parser is the process of examining a sequence of contiguous constituents in the course of establishing whether or not they form some higher level constituent. A data structure that allows random access to contiguous constituents is the natural choice for implementing such a process.

The parser that is the central focus of this document has both top-down and bottom-up aspects, and for this reason it is most clearly conceptualized as using two different data structures.

The active node stack captures the top-down aspect of the parser. A node is pushed onto the active node stack when the parser is actively attempting to find its subconstituents, to add to its internal structure. However, even here, rather than postulating possible sub-constituents of the current active node and then attempting to complete their internal structures in a purely top-down manner, the parser activates a number of pattern-action rules that attempt to recognize and then build subconstituents of the current active node by examining contiguous sequences of constituents in the buffer. The activation of rules which recognize appropriate subconstituents for a given current active node is a top-down process, while the triggering of these rules once activated is bottom-up. Thus, this process itself has both bottom-up and top-down aspects. (By "activate", I mean that the grammar interpreter will only attempt to match rules that are declared to be "active". The operation of the parser will be specified in detail in the following chapter. The discussion here is meant to give only a quick sketch of the parser's operation.)

Constituents can also be initiated in a purely bottom-up manner by pattern-action rules that are always active regardless of the features of the current active node. Such a rule recognizes the presence of a constituent by noting a sequence of buffered subconstituents that starts a given sort of constituent regardless of the current grammatical environment. However, once a constituent is started by such a rule, it is pushed onto the active node stack where the parser will then operate in the mixed mode described immediately above, attempting to complete its internal structure by activating pattern-action rules appropriate for that type of node.

In short, the parser maintains two different sorts of data structures because each is a natural structure for one aspect of the parser's operation. The active node stack contains nodes to which the parser is attempting to add daughters, an essentially top-down process; the buffer contains nodes for which the parser is attempting to find a father, an essentially bottom-up process. While there are single data structures which can be used for both types of operation, such as Martin Kay's *charts* [Kay 73], using two separate data structures helps to keep the two aspects of the parser's operation conceptually clear.

A Snapshot of the Parser

Figure 3.8 is a snapshot of the parser, providing an overall view of the parser's data structures. It was taken while the parser was processing the sentence 3.7 at the point when the parser has seen the initial segment "John should have scheduled...":

3.7 John should have scheduled the meeting.

At this point there are two nodes in the active node stack, an S node, S1, and the current active node, AUX1. There are two constituents in the buffer, WORD3, the verb "have", and WORD4, the verb "schedule".

The Active Node Stack

S1 (S DECL MAJOR)
NP : (John)
C: AUX1 (MODAL PAST VSPL AUX)
MODAL : (should)

The Buffer

1 : WORD3 (*HAVE VERB TNSLESS AUXVERB PRES V-3S) : (have)
2 : WORD4 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S
ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words: the meeting .

Figure 3.8 - A snapshot of the state of the parser.

Before explaining this figure in detail, a few words about the representation of parse nodes in this and following snapshots will be helpful. The buffer will be displayed vertically with x[1] on top, with the absolute index of the cell on the far left. Each node in the buffer is represented in the diagram by a configuration of the form of Figure 3.9.a, while each node in the active node stack is represented by a configuration of the form of 3.9.b:

(<nodename> <features> : <terminal word string>
(a) - The representation of nodes in the buffer

<nodename> <features>
<type of daughter> : <terminal string under daughter>
...
<type of daughter> : <terminal string under daughter>
(b) - The representation of nodes in the active node stack

Figure 3.9 - Node configurations in parser snapshots.

At the instant that the snapshot of Figure 3.8 was taken, the parser is in the process of building AUX1, the current active node, which is an *AUX* (auxiliary) node with the word "should" attached as a MODAL. This node has been labelled with the person feature *VSPL* (i.e. this verb agrees with any NP, either *S*ingular or *PL*ural) and the tense feature *PAST* (past tense). (An auxiliary is more or less the clump of helping verbs that precedes the main verb of a clause, plus the tense marking of the entire verb cluster.) The node S1, a *DECL* (declarative) *MAJOR S* (sentence) with the NP "John" attached to it, has been pushed into the interior of the active node stack. The first position in the buffer is filled by WORD3, the *VERB* "have", with the root HAVE (**have* is a feature meaning "has the root *have*"). It can be an *AUXVERB* (an auxiliary verb, i.e. a "helping verb"), and it can either be a tensed verb with the features *V-3S* (it agrees with any NP that is not 3rd singular) and *PRES* (present tense) or it can be *TNSLESS* (tenseless). The second word in the buffer is WORD4 "scheduled", which is labelled *COMP-OBJ*, *INF-OBJ* (it takes infinitive phrases as complement objects), *ED*, *EN*, *ED=EN* (this verb is the "ed" form of the verb, but since the "en" and "ed" forms are equivalent, it is also the "en" form, and *PART* (participle).

The Basic Operations of the Parser

At each point in the parsing process, there are fundamentally only three things that the parser can do in terms of building structure:

1) The parser can *attach* a constituent in the buffer to the current active node (which I will henceforth refer to as C). The parser attaches a node X to a node Y by making X the rightmost daughter of Y; i.e. it *right-adjoins* X to the rightmost daughter of Y, in the sense of Peters & Ritchie [Peters & Ritchie 73]. In the simplified parse model presented here, as soon as the grammar interpreter notices that a constituent in some cell x[i] in the buffer has been attached, it removes it from the buffer by executing the command Delete(i). Most often, grammar rules will use the buffer as if it were a first-in, first-out mechanism, thus the first constituent in the buffer, rather than the second or third constituent, will most commonly be the constituent attached to C.

2) The parser can decide that the first constituent or constituents in the buffer begin a new constituent, in which case it will *create* a new active node. A node is created by pushing a new node of a specified type onto the active node stack, which means that each new node immediately becomes C, the current active node. A newly created node has a previously unused name, and no properties other than its type; it has no daughters, no father, and no features, although it may be attached immediately after its creation (see immediately below). The parser is then free to attach constituents to this new active node and to assign it features.

3) The parser can decide that the current active node is complete, and pop it from the active node stack, resulting in one of the two following configurations:

The parser often knows exactly what the higher level role of a constituent is at the time that the node is first created. In this case, the parser can attach the newly created node to the old current active node at the instant of its creation. This happens before the active node stack is pushed, and the new node itself becomes C. When this node is completed and popped off of the stack, it remains attached to its parent, as would be expected.

Sometimes, however, the parser can be sure that the constituents in the buffer begin a higher level constituent of a given sort without knowing what the higher level role of that constituent is. Often the problem is that the parser cannot be sure that the newly created node attaches to the node immediately above it on the stack. This happens, for example, when the newly created constituent may in fact attach to some active constituent above its immediate predecessor on the stack. In this case, the newly created node pushes the stack and becomes C, but without being attached to any higher node at all.

When such a node is completed and popped from the stack, it surely cannot remain hanging from its nonexistent parent node. Instead, the grammar interpreter inserts the node into the first buffer cell by executing

the instruction Insert(C,1), and then popping the node from the stack, a composite operation which I will call *dropping a node into the buffer*. Usually, this occurs immediately after the last daughter of the node has been attached, so the node can be thought of as filling the buffer position that its last daughter just vacated. The grammar interpreter does not provide explicit commands in the grammar writing language for inserting nodes on the active node stack into the buffer, nor for popping a node from the active node stack, thus this operation of dropping C into the buffer is the only operation the interpreter allows for moving constituents from the active node stack into the buffer. The grammar interpreter has been constrained to only allow constituents to be inserted into the first buffer cell because, empirically, it seems that the extra degree of freedom is simply not necessary.

A good example of this sort of situation is the problem of prepositional phrase attachment. In sentence 3.10.a, for example, the parser can be sure that "with" starts a prepositional phrase when this word enters the buffer, but it cannot possibly decide at that time whether the resulting PP should be attached to "the man" (the correct reading here) or to the clause itself, as is the most plausible reading of 3.10.b.

3.10.a I saw the man with the red hair.

3.10.b I saw the man with the telescope.

To solve this attachment problem deterministically requires access to some sort of semantic (or if the reader prefers, pragmatic,) reasoning capability, but a necessary precursor to the application of this knowledge is the ability to first parse the prepositional phrase independent of its higher

level role. This part of the problem can be solved simply by creating an unattached PP node when the preposition comes into the first buffer position, attaching its preposition and object, and dropping the resulting PP into the buffer, where whatever rules will bring non-syntactic knowledge into play can examine the PP at their leisure.

Another good example of a situation of this sort is provided by 1.1, repeated here as 3.11

3.11.a Is the block sitting in the box?
3.11b Is the block sitting in the box red?

In these sentences, as discussed at length in Chapter 1, the function of "sitting in the box" cannot be decided until the following grammatical context can be examined. But note that in either case, we can be sure that "sitting in the box" can be parsed as a VP. This VP can then be attached as either the VP of a relative clause attached to "the block", or as the VP of the major clause itself.

The ability to parse a constituent before its higher level grammatical role can be determined and then drop that constituent back into the buffer is exactly the mechanism needed to cut through the seeming nondeterminism of the problem above. The full solution will be formulated in a later chapter, but I will sketch the general outline of the solution here. (I will take some liberties with the sketch to avoid some details of the parser that have not yet been discussed.)

I will assume here that the parser has recognized that the input

sentence is a yes-no question, with "is" as auxiliary. I will also assume that the parser is in the process of parsing the NP beginning with "the block" when it encounters the participle "sitting" in the buffer. I will also assume that the parser has already attached the incomplete NP "the block" to the clause as subject.

Given these assumptions, the parse can solve the participial phrase problem by proceeding as follows: First, the parser builds an unattached VP with the participle as main verb. When this VP is complete (ignoring here the general problem of how *this* is to be decided), the parser drops it back into the buffer, where it will take up the first of the three buffer positions. Now the parser can examine the constituent that follows the participial phrase. If the following constituent can serve as the main predicate of the clause (as "red" can), then the parser should attach a reduced relative clause to the NP which will be the current active node at this point, and attach the participial phrase as its VP. If the following constituent cannot serve as the main predicate of the clause (as "?" certainly cannot), then the parser should complete the NP by popping it off of the active node stack, leaving the S node as the current active node. The parser can then attach the participial phrase as the VP of the major clause, and continue on its way.

CHAPTER 4

THE STRUCTURE OF THE GRAMMAR

Introduction

In the previous chapter, I discussed the structure of the grammar interpreter itself. In this chapter, I will first discuss the format for grammars that this interpreter expects, and then some of the details of a particular grammar for English written for this mechanism. The emphasis here will not be on the complexities of the grammar, but rather on the form of the grammar in broad outline and on the details of interpreting such a grammar.

To this end, the last two-thirds of the chapter consists of two examples of increasing complexity. The first discusses the parsing of an auxiliary phrase, involving only a small handful of simple grammar rules. This example primarily stresses the details of the interpreter's functioning. The second example traces the parsing of a simple declarative sentence. Here the focus is not only on the functioning of the interpreter, but also on the overall organization of a set of grammar rules sufficient for parsing an entire sentence, albeit a simple one. The reader should be warned that these examples discuss the parsing process in fine detail, and therefore are fairly lengthy; in subsequent chapters I will assume that the reader is familiar with these details, and will focus on the details of a grammar of English written within this framework.

The Grammar

A grammar for this parser is made up of pattern/action rules, and can be viewed as an augmented form of Newell and Simon's production systems [Newell & Simon 72]. Each rule is made up of a pattern, which is matched against some subset of the constituents of the buffer and the current active node, and an action, a sequence of operations which acts on these constituents. Each rule is assigned a numerical *priority*, which the grammar interpreter uses to arbitrate simultaneous matches. The grammar as a whole is further structured into *rule packets*, clumps of grammar rules which can be turned on and off as a group; the grammar interpreter only attempts to match rules in packets that have been activated by the grammar. These rules are written in a language called PIDGIN, an English-like formal language that is translated into LISP by a simple grammar translator based on Pratt's notion of top-down operator precedence [Pratt 73]. Figure 4.1 gives a schematic overview of the organization of the grammar.

The grammar:

Packets Of Pattern/Action Rules
Matched Against The Buffer, the Current Active Node,
and the Current Cyclic Node
Ordered By Priority

Priority	Pattern Description of: 1st	2nd	3rd	C	Action
		PACKET1			
5:	[ ]	[ ]	[ ]		--> ACTION1
10:	[ ]			[ ]	--> ACTION2
10:	[ ]	[ ]	[ ]	[ ]	--> ACTION3
		PACKET2			
10:	[ ]	[ ]			--> ACTION4
15:	[ ]			[ ]	--> ACTION5
		PACKET3			
5:	[ ]	[ ]			--> ACTION6
15:	[ ]	[ ]	[ ]		--> ACTION7

THE BUFFER:

1st	2nd	3rd

Figure 4.1 - The structure of the grammar.

Grammar Rules

The pattern of a grammar rule is made up of a list of partial descriptions of parse nodes which must all be fulfilled if the pattern is to match. There can be up to five partial descriptions in each pattern: one to match against each of the three nodes in the buffer, and two to match against the two accessible nodes in the active node stack, the current active node and the current cyclic node. (Actually, the grammar notation is such that the two node specifications which are matched against the stack are combined into one pattern.) Each description is made up of a Boolean

combination of tests for given grammatical features, equivalent in power to an arbitrary expression in the propositional calculus (as distinguished, of course, from the predicate calculus). Each description can also include boolean feature tests on the daughters of the target node; the grammar language provides a tree walking notation for indicating specific daughters of a node. While this richness of specification seems to be necessary, it should be noted that the majority of rules in even a moderately complex grammar have patterns which consist only of tests for the positive presence of given features.

Rule actions consist of rudimentary programs that do the actual work of building constituent structures. These actions are built up of primitives that perform such actions as:

-creating a new parse node.

-inserting a specific lexical item into a specific buffer cell.

-attaching a newly created node or a node in the buffer to the current active node or the current cyclic node, causing the grammar interpreter to remove the node from the buffer.

-popping the current active node from the active node stack, causing it to be dropped into the buffer if it is not attached.

-assigning features to a node in the buffer or one of the accessible nodes in the stack. (PIDGIN also provides an operator which returns a pointer to a specific daughter of a given node (specified by type), allowing features to be assigned to these nodes as well.)

-activating and deactivating packets of rules.

PIDGIN also provides primitives from which boolean tests of the features of a node can be constructed, as mentioned above. These predicates can be used within conditional "if...then...else..." expressions to conditionally perform

various operations.

With the exception of allowing conditional expressions, PIDGIN rules fall into the simple class of programs that Goldstein [Goldstein 74] calls *fixed-instruction programs*. The PIDGIN language imposes the following constraints on rule actions:

1) There are no variables within the rule actions. The constituents in the first three buffer cells are available as the values of the parameters *1st*, *2nd* and *3rd* within each rule, but these parameters are given values by the grammar interpreter before each rule action is called, and are not resettable within an action. The values of the parameters *C* and *the current cyclic node* do change within a rule as nodes are pushed and popped from the active node stack, but their values cannot be set by a grammar rule.

2) PIDGIN allows no user-defined functions; the only functions within actions are PIDGIN primitives. (There is a limited ability for a rule to circumvent the pattern-matching process by explicitly naming its successor, but this is formally only a device for rule abbreviation, since the specification of the successor rule could simply be replaced with the code for the action of the rule named.)

3) While conditional "if...then....else..." expressions are allowed, there is no recursion or iteration within actions.

4) The only structure building operations in PIDGIN are a) attaching

one node to another, and b) adding features from a node's feature set. In particular, the list building primitives of LISP are *not* available in PIDGIN.

For a complete specification of PIDGIN, see Appendix B.

Each grammar rule is assigned a numerical priority; these priorities impose a partial ordering on the rules in the grammar. (Priorities are assigned as non-negative integers, with 0 the highest priority.) At each point in the parsing process, the parser executes the action of the rule of highest priority whose pattern matches. If there is more than one rule with this priority that matches, then the parser is free to arbitrarily choose one of them. (If this happens, presumably, the rules involved do not interact with each other, the other rules will be executed on following pattern matches, and thus the order of their execution is irrelevant.) In the present grammar, with a few exceptions, three numeric values that correspond roughly to high, low, and normal priorities seem to suffice to order the grammar rules. Most often, this priority mechanism is used when the pattern of a rule R_1 is a further specification of the pattern of a rule R_2. In this case, R_1 will only match if R_2 does as well, and yet in such a situation it is almost always true that the rule with the more detailed pattern should be executed. By giving R_1 a higher priority than that assigned to R_2, the grammar interpreter will execute R_1 when both rules potentially match. During the pattern matching process, words enter the end of the buffer as they are needed, so a rule is never blocked from matching because the buffer does not contain sufficient constituents to match its pattern.

Note that instead of this numeric priority scheme, priorities could have been implemented by explicitly representing pairwise which rules have priority over other given rules if more than one rule can potentially match. In essence, the numeric scheme used here is a compiled form of such a pairwise scheme.

Packets of Rules

The grammar as a whole is structured into *packets* of rules, each of which is activated or deactivated as a unit. This packeting mechanism allows the parser to dynamically add and remove rules from the active grammar during the course of processing a sentence. This is useful because most grammar rules are applicable only under particular circumstances which usually reflect global properties of the structures built up so far. At any time, the parser only attempts to match rules in active packets, thus ignoring all rules known to be irrelevant to the current global environment. (This notion of packets derives here from [Fahlman 73]; the idea of segmenting sets of productions (into "run modes") appeared independently in the PAS-II system of Waterman and Newell [Waterman & Newell 73].)

To see how the rules in the grammar are partitioned into packets, consider Figure 4.2 below. Under each of the two simple phrase structure rules given are the names of the packets that can be thought of as being associated with that node. (The symbol *COMP* in (2) stands for the various NPs, PPs and Ss that can complete a VP.) Packets associated with the left hand side of a rule include rules that are generally applicable while parsing

the subconstituents to the right hand side of the rule; packets associated with a symbol on the right hand side of a rule either initiate a constituent of that type or handle grammatical phenomena that are best considered at approximately that point in the parsing process.

1) S	—>	NP	AUX	VP	(PP)*
CPOOL		PARSE-SUBJ	PARSE-AUX	PARSE-VP	SS-FINAL EMB-S-FINAL
2) VP	—>	VERB	COMP		
		SUBJ-VERB	SS-VP, EMBEDDED-S-VP WH-VP, INF-COMP THAT-COMP, ...		

Figure 4.2 - Packets can be associated with phrase structure rules.

Here are some examples of the types of rules contained within some of the packets shown in Figure 4.2:

-The packet *PARSE-SUBJ* contains the grammar rules which pick out the subjects of various types of clauses.

-The packet *CPOOL* contains rules that are active whenever any clause level constituent (as opposed to an NP) is being parsed. It contains rules that start NPs, pick up adverbs, and the like.

-*SS-FINAL* contains rules that attach final PPs (and the like) to simple sentences; *EMB-S-FINAL* is similar, except that it contains rules that must decide whether a given PP attaches to an embedded clause currently

under construction or to some higher level constituent. Exactly one of these two packets will be activated for any given clause.

-The packets *WH-VP*, *EMBEDDED-S-VP*, and *SS-VP* all parse the objects of the verb as well as PPs dominated by the VP node; they differ in that the first is activated for clauses which have WH-heads that must be "put back" (i.e. for clauses which are WH-questions, relative clauses, etc.), the second for embedded clauses that are not relative clauses or indirect questions, and the third for major clauses that are not WH-questions.

-The packets *INF-COMP* and *THAT-COMP* contain rules that initiate infinitive complements and that complements. Note that more than one packet will often be active to pick up the objects and complements of a verb, since many verbs (e.g. "believe") can take either a simple object or any of various sorts of complements.

(It should be noted that the explicit association of packets with node types in phrase structure rules is *not* part of the current implementation of the parser. This association could be made explicit, I believe, relegating the question of packet switching, currently done explicitly in grammar rules, to the grammar interpreter. This potential extension of the parser is also interesting in that it would add a component analogous to the generative grammarian's *base component* to the parsing model. As it is, this component can be considered to implicitly exist in the current parser. A characterization of the set of grammar rules introduced in this chapter will be given at the end of the chapter in the form of a diagram like that shown in Figure 4.2

above; such a diagram can be considered as the equivalent of an ATN transition diagram for this sort of parser.)

Given this motivation for partitioning the grammar rules into packets, it should not be surprising that a packet is not activated or deactivated globally, but rather with respect to a given active node. Associated with each node in the active node stack is a set of packets to which grammar rules can add or subtract packets. These packets will be said to be *associated with* that active node. When an active node is the current active node, the packets associated with that node are also active, in that the grammar interpreter will attempt to match the rules in those packets and only those packets. Thus, whenever the active node stack is popped and a node previously pushed into the stack becomes the current active node, all the packets previously associated with that node become active. If a new node is created, pushing the previous current active node back into the stack, the packets associated with the previous current active node are no longer active until either they are associated with the new current active node or the new node is popped off the stack, restoring the old current active node to the bottom of the stack.

Figure 4.3 below is a more complete version of the snapshot shown in Figure 3.8. The packets associated with each active node are shown after the node description, following a slash. At the time this snapshot was taken, the packets *PARSE-AUX* and *CPOOL* have been associated with S1, the top level clause. The packet *BUILD-AUX*, associated with AUX1, the current active node, is the only packet active. The rules in this packet, written in

PIDGIN, are shown in Figure 4.4; the rule PERFECTIVE had matched and was about to be run when the snapshot was taken.

The Active Node Stack (1. deep)

S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)
C: AUX1 (MODAL PAST VSPL AUX) / (BUILD-AUX)
MODAL : (should)

The Buffer

1 : WORD3 (*HAVE VERB TNSLESS AUXVERB PRES V-3S) : (have)
2 : WORD4 (*SCHEDULE COMP-OBJ VERB INF-OBJ v-3s
ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words: the meeting .

Figure 4.3 - A set of packets is associated with each active node.

{RULE PERFECTIVE PRIORITY: 10. IN BUILD-AUX
[=*have] [=en] --> Attach 1st to c as perf. Label c perf.}

{RULE PROGRESSIVE PRIORITY: 10. IN BUILD-AUX
[=*be] [=ing] --> Attach 1st to c as prog. Label c prog.}

{RULE PASSIVE-AUX PRIORITY: 10. IN BUILD-AUX
[=*be] [=en] --> Attach 1st to c as passive. Label c passive.}

{RULE MODAL PRIORITY: 10. IN BUILD-AUX
[=modal] [=tnsless] --> Attach 1st to c as modal. Label c modal.}

{RULE DO-SUPPORT PRIORITY: 10. IN BUILD-AUX
[=*do] [=tnsless] --> Attach 1st to c as do.}

{RULE AUX-COMPLETE PRIORITY: 15. IN BUILD-AUX
[t] --> Drop c into the buffer.}

Figure 4.4 - Some rules that build auxiliaries.

The grammar rules above should be by and large self-explanatory, but a few comments on the grammar notation are in order. The general form of each grammar rule in this example is:

{Rule <name> priority: <priority> in <packet>
<pattern> --> <action>}

Each pattern is of the form :

[<description of 1st buffer constituent>] [<2nd>] [<3rd>]

The symbol "=", used only in pattern descriptions, is to be read as "has the feature(s)". Features of the form "*<word>" mean "has the root <word>", e.g. "*HAVE" means "has the root "have"". The tokens "1st", "2nd", "3rd" and "C" (or "c") refer to the constituents in the 1st, 2nd, 3rd buffer positions and the current active node, respectively. (I will also use these tags in the text below as names for their respective constituents.) The "t" used in the pattern of AUX-COMPLETE is a predicate that is true of any node, thus "[t]" is the always true description. The PIDGIN code of the rule actions should otherwise be fairly self-explanatory.

The rules used here and in the first example which will be presented in the following sections are taken from a fairly large grammar, and are not "toy" rules. However, this first example has been carefully chosen so that only rules that use a small subset of PIDGIN are involved. These rules should not be taken as representative in their simplicity. The final example presented in this chapter will use rules more representative in their complexity.

The Packeting Mechanism is Redundant

While the division of the grammar into packets increases both the perspicuity of the grammar and the efficiency of the pattern matching process, it should be noted that the packeting mechanism is not, strictly speaking, necessary. While each packet reflects some global property of the grammatical structure currently under construction, this property can without exception be detected by a boolean combination of feature tests on nodes accessible to a grammar rule. In this sense, each packet can be considered to be a shorthand for a predicate testing global properties of the structure under the current cyclic node. For example, the pattern of each rule in the packet *PARSE-SUBJ* mentioned above could test to see if there was an NP under the current S node; these rules are only applicable if there is no such NP. Similarly, the patterns of rules in packet *SS-FINAL* could include a test for the presence of a VP node under the current S, and the rules in packet *CPOOL* would include no such test at all.

To note that the packeting mechanism is not a necessity is not to say that this mechanism is purely an efficiency measure. First of all, the division of the grammar into packets of rules adds perspicuity to the grammar and aids in conceptualizing how the grammar operates. Secondly, the packeting mechanism is made redundant only by the availability of powerful operations that allow grammar rules to: 1) examine much of the tree structure built up so far, 2) to test for the presence *and absence* of daughters of a given type attached to a given node, and 3) to perform arbitrary boolean feature tests on the accessible nodes. If it is possible to do so while still capturing a concise grammar of English, one would like to

further restrict the grammar mechanism by eliminating some of these operations. In this sense, it is useful to think of the grammar mechanism presented here as an extension of a simpler mechanism without some of these operations, and therefore in which the packeting mechanism *is* a necessity. In short, while the packeting mechanism is redundant, it is not entirely peripheral.

In this regard, it is interesting to note that the states of an ATN are redundant in exactly this sense, given the register mechanism that the ATN model provides. All the state information in an ATN can be encoded into the register mechanism, leaving a one-state ATN. To dispense with the states of a given "source" ATN, the following procedure can be executed: First, construct a one-state ATN with a register named S_j corresponding to each state S_j in the source machine. For each arc ARC_i in the original ATN, call the state it leaves state S_{into-i} and the state it goes to state $S_{outof-i}$. Now, for each arc ARC_i in the original machine, construct an arc ARC_i' in the one-state machine whose test is identical to the test of ARC_i except that it also tests to see that register S_{into-i} has been set non-zero. The action of ARC_i' is identical to the action of ARC_i except that it also 1) sets register S_{into-i} to zero and 2) sets register $S_{outof-i}$ non-zero. This one-state ATN is initialized with all registers S_i set to zero, except for the register which corresponds to the starting state of the original ATN.

It is clear that this one-state ATN encodes the state of the equivalent multi-state ATN in its registers by simply setting, at each point in the parsing process, exactly one of its "state registers" non-zero, corresponding

to the state that the source machine is in at the analogous point in its parsing process. While the state mechanism is essential for the underlying finite state machine mechanism that the ATN extends, the addition of the registers to the model has made the state mechanism redundant. Again, the key point here is not that the states of an ATN should be discarded as unnecessary, but rather that they play an important role in simplifying the structure of the ATN's grammar and in conceptualizing the operation of the machine.

An Example

To illustrate the operation of the parser I will trace the construction of the auxiliary of sentence 4.5 repeated below, which was almost complete when the snapshot in Figure 4.3 above was taken.

4.5 John should have scheduled the meeting.

We start our example with the parser in the state depicted in snapshot 4.6. At this point, S1 is the current active node, and WORD2 "should" is in the 1st buffer position. The NP "John" has been attached to node S1 as its subject. The packets PARSE-AUX and CPOOL are both associated with the current active node, and thus both are currently active. (The packet CPOOL does not enter into our example and it will be ignored below.)

About to run: STARTAUX

The Active Node Stack (0. deep)

C: S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)

The Buffer

1 : WORD2 (*SHOULD VERB AUXVERB MODAL VSPL PAST) : (should)

Yet unseen words: have scheduled the meeting .

Figure 4.6 - Before the auxiliary is initiated.

The active packet PARSE-AUX, shown in Figure 4.7 below, contains three rules: START-AUX, TO-INFINITIVE, AND AUX-ATTACH. The first two of these rules initiate the building of two different sorts of auxiliaries; the third rule attaches completed auxiliaries to the dominating S node. Since the three rules are all of the same priority, priority considerations have no effect on the pattern matching process when only these rules are applicable. (Note that the first two of these rules could equally well have been written so that the AUX node was attached when created. They are written otherwise mainly to illustrate the operation of the parser.)

The reader is reminded that Appendix B discusses the details of PIDGIN syntax. Also, Appendix C contains a compendium of all the rules that are mentioned in this chapter, organized by packet.

Given the contents of the buffer in Figure 4.6, the rule START-AUX will match, since "should" is indeed labelled an *AUXVERB*. The rules AUX-ATTACH and TO-INFINITIVE cannot match, since WORD2 is neither **TO*, nor

AUX. Since only START-AUX can match, it does match, and its action is run.

```
{RULE START-AUX PRIORITY: 10. IN PARSE-AUX
[=verb] -->
Create a new aux node.
Label C with the meet of the features of 1st and vspl, v1s,
        v+13s, vpl+2s, v-3s, v3s.
%(The above features are "person/number codes", e.g. "vpl+2s"
means that this verb goes with any plural or 2nd person singular
np as subject. The verb "are" has this feature.)%
Label C with the meet of the features of 1st and pres,
        past, future, tnsless.
Activate build-aux.}

{RULE TO-INFINITIVE PRIORITY: 10. IN PARSE-AUX
[="to, auxverb] [=tnsless] -->
Label a new aux node inf.
Attach 1st to c as to.
Activate build-aux.}

{RULE AUX-ATTACH PRIORITY: 10. IN PARSE-AUX
[=aux] -->
Attach 1st to c as aux.
Activate parse-vp. Deactivate parse-aux.}
```

Figure 4.7 - Some rules that initiate and attach auxiliaries.

START-AUX first creates a new aux node, node AUX1 in fig. 4.8, which thus becomes C, the current active node. Since this node is newly created, no packets are associated with AUX1, i.e. no packets are active, until START-AUX activates the packet BUILD-AUX, thus associating it with AUX1. The rule action next labels the new aux node with the tense and person/number features of 1st, WORD2, the auxiliary "should". It does this by intersecting the features of WORD2 with a full set of person/number features and labelling AUX1 with the resulting intersection. Similarly, the intersection of the features of WORD2 and a full set of tense features is added to the feature set of AUX1. This leaves C in the state shown in fig.

4.8 below, with no daughters, but labelled with the tense feature *PAST*, the person/number feature *VSPL*, and the type feature *AUX*.

It should be pointed out that the snapshots used in this example were taken immediately before a rule action was about to run, after the pattern matching process which triggered it, and not immediately after the previous rule finished. While the active node stack is not affected at all by the pattern matching process, the buffer is. Two things happen to the buffer between the finish of one rule action and the beginning of the next: 1) After every rule action reaches completion, the grammar interpreter scans the buffer to remove nodes attached by the previous rule action. 2) During the pattern matching process new words enter the buffer as they are needed to match against pattern descriptions. Thus, we will see in a moment that WORD3, shown in the second buffer position in Figure 4.8, enters the buffer during the pattern matching process following the completion of the action of START-AUX.

```
About to run: MODAL

          The Active Node Stack ( 1. deep)

            S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                    NP : (John)
 C:         AUX1 (PAST VSPL AUX) / (BUILD-AUX)

          The Buffer

 1 :        WORD2 (*SHOULD VERB AUXVERB MODAL VSPL PAST) : (should)
 2 :        WORD3 (*HAVE VERB TNSLESS AUXVERB PRES V-3S) : (have)

Yet unseen words:  scheduled the meeting .
```

Figure 4.8 - After rule START-AUX has been run.

Packet BUILD-AUX is now the only packet active. It contains the rules shown in Figure 4.4 earlier in this chapter.

Note that as long as this packet is active, the rule AUX-COMPLETE will potentially always match, since its vacuous description is met by any constituent that fills the first buffer position. However, since AUX-COMPLETE, with a priority of 15, has a lower priority than any other rule in this packet, it will only match if the pattern of no higher prioritied rule is fulfilled by the contents of the buffer. This rule, with always true pattern and low priority, serves as a *default action*; when no other rule applies, this rule will trigger, completing the auxiliary by dropping it into the buffer.

After the action of START-AUX is complete, WORD2 "should" fills the 1st buffer position and no constituent fills either 2nd or 3rd. The rule MODAL is now the only rule in BUILD-AUX whose pattern is consistent with the contents of the buffer, since WORD2 is labelled with the feature *MODAL*. After the description of 1st in MODAL's pattern has been fulfilled, there is now demand for the next word in the input string to fill 2nd, and WORD3, "have", enters the buffer. WORD3 fulfills the description of 2nd in MODAL's pattern, since it is labelled with the feature *TNSLESS* (tenseless). MODAL's pattern thus matches, with the buffer now exactly in the configuration shown in fig. 4.8 above.

The action of MODAL now runs, attaching WORD2 to AUX1 as a

MODAL, and labelling AUX1 with the feature *MODAL*. This leaves C as shown in Figure 4.9.

```
About to run: PERFECTIVE

          The Active Node Stack ( 1. deep)

            S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                    NP : (John)
C:          AUX1 (MODAL PAST VSPL AUX) / (BUILD-AUX)
                    MODAL : (should)

          The Buffer

1 :       WORD3 (*HAVE VERB TNSLESS AUXVERB PRES V-3S) : (have)
2 :       WORD4 (*SCHEDULE COMP-OBJ VERB INF-OBJ v-3s
                              ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words:  the meeting .
```

Figure 4.9 - After rule MODAL has been run.

After MODAL is finished, the grammar interpreter notices that WORD2 has been attached, and removes it from the buffer, thus leaving WORD3, "have", as 1st.

The rule PERFECTIVE now matches, its pattern fulfilled by WORD3 with the feature **HAVE*, and WORD4 "scheduled", entering the buffer on demand, with the feature *EN*. (Note that the strong verbs of English, like "scheduled", have "en" forms that are morphologically equivalent to their "ed" forms, thus the feature *ED=EN* on WORD4.) The action of PERFECTIVE now attaches WORD3 to AUX1, and labels AUX1 with the feature *PERF*. The grammar interpreter, upon the completion of the rule, removes WORD3 from the buffer, leaving the parser in the state depicted in Figure 4.10, which is the same state depicted in Figure 4.3 above.

```
About to run: AUX-COMPLETE

          The Active Node Stack ( 1. deep)

           S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                    NP : (John)
C:         AUX1 (PERF MODAL PAST VSPL AUX) / (BUILD-AUX)
                    MODAL : (should)
                    PERF : (have)

          The Buffer

1 :        WORD4 (*SCHEDULE COMP-OBJ VERB INF-OBJ v-3s
                          ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words:  the meeting .
```

Figure 4.10 - After the rule PERFECTIVE has been run.

The rule PERFECTIVE provides a good example of how the parser uses the ability to buffer constituents to help decide what grammatical role a constituent in the buffer fills.

At the instant the snapshot 4.9 was taken, the parser is implicitly confronted with the problem of deciding if WORD3 "have" serves here as an auxiliary verb, as in 4.11.1 below, or as the main verb of the clause, as in 4.11.2:

4.11.1 John had left yesterday.
4.11.2 John had a book.

Note that a simple test will serve to decide the issue, if the parser can only look at the word following "have": if the following word has the features *VERB* and *EN* then "have" is an auxiliary, otherwise it is the main verb of the clause. WORD4, "scheduled", the second word in the buffer in this example, does have the features *VERB*, and *EN*, so by this test "have" serves

as auxiliary in our example.

Now note that the rule PERFECTIVE will match in exactly the case when "have" or one of its forms is followed by a verb which fulfills the perfective branch of the test above. The action of this rule will then attach "have" as an auxiliary verb marking the perfective. Otherwise, the rule AUX-COMPLETE will apply, completing the auxiliary without attaching "have". Later rules will then interpret "have" as the main verb of the clause.

It is thus the case that the simple rules shown in this example embody the test discussed immediately above, but do so in an implicit form that causes the right structure to simply "fall out" in most cases. There are cases where this test fails, as in sentences like 4.12 below, causing the parser to misparse its input, but most interestingly, most people will make exactly the same mistake as the grammar presented here when first reading such a sentence. For more on this sort of "diagnostic failure", see Chapter 9, which discusses the phenomenon of "garden path sentences".

4.12 The store has assembled models, as well as kits, in stock.

Returning to our example, we now see that none of the rules in packet BUILD-AUX with the normal priority of 10 can apply. Since no rule of higher priority matches, the default rule AUX-COMPLETE triggers at long last. This rule pops the current active node, AUX1, from the the active node stack. Since AUX1 is not attached as daughter to any other node, it drops into the first position in the buffer, as discussed earlier, shifting WORD4 into

the second buffer position. Node S1 is once again the current active r and the packets PARSE-AUX and CPOOL associated with S1 are once again active packets. This situation is shown in Figure 4.13.

About to run: AUX-ATTACH

The Active Node Stack (0. deep)

C: S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)

The Buffer

1 : AUX1 (PERF MODAL PAST VSPL AUX) : (should have)
2 : WORD4 ("SCHEDULE COMP-OBJ VERB INF-OBJ v-3s
ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words: the meeting .

Figure 4.13 - After the rule AUX-COMPLETE has been run.

The rule AUX-ATTACH in packet PARSE-AUX will now match, will attach AUX1 to S1. It will then deactivate PARSE-AUX and activate packet PARSE-VP, which will initiate the parsing of the verb phrase of sentence, as its name implies. After the rule AUX-ATTACH is done, grammar interpreter will remove AUX1 from the buffer, leaving the pa in the state depicted in Figure 4.14, and bringing our example to a close.

About to run: MVB

The Active Node Stack (0. deep)

C: S1 (S DECL MAJOR S) / (PARSE-VP CPOOL)
NP : (John)
AUX : (should have)

The Buffer

1 : WORD4 (*SCHEDULE COMP-OBJ VERB INF-OBJ v-3s
ED=EN EN PART PAST ED) : (scheduled)

Yet unseen words: the meeting .

Figure 4.14 - After AUX-ATTACH has been run, completing the auxiliary.

As a final point, it is worthwhile to consider the auxiliary structure that the rules discussed above produce if the verb cluster consists of a single verb, as in the sentence given as Figure 4.15.a below. The observant reader may have noted that in this case an auxiliary node will be created and dropped into the buffer with no daughters attached at all; he might not have noticed that this result is entirely correct. Given the sentence of 4.15.a, the parser will be in the state shown in Figure 4.15.b after rule START-AUX has been executed. The action of this rule has (quite correctly) not attached any terminal to the aux node, but it has marked the auxiliary with the tense and person/number codes of the verb. (The reader should note that the pattern for START-AUX triggers on any verb at all, not only auxiliaries.) At this point the packet BUILD-AUX is active, but the only rule that matches is AUX-COMPLETE. This rule will drop the auxiliary into the buffer, resulting in the state shown in Figure 4.15.c. But this state is completely correct; the state shown in 4.15.c is completely analogous to that depicted in Figure 4.13 of the previous example. The auxiliary carries the

tense and person/number codes, as it should, and also carries the aspect features of the verb cluster, in this case a null set of features.

(a) John scheduled the meeting for Wednesday.

The Active Node Stack (1. deep)

S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)
C: AUX1 (PAST V-3S AUX) / (BUILD-AUX)

The Buffer

1 : WORD2 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words: the meeting for Wednesday .

(b) - After START-AUX has been run.

The Active Node Stack (0. deep)

C: S1 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)

The Buffer

1 : AUX1 (PAST V-3S AUX) : NIL
2 : WORD2 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words: the meeting for Wednesday .

(c) - After AUX-COMPLETE has been run.

Figure 4.15 - Parsing a verb cluster with no auxiliary verbs.

Parsing a Simple Declarative Sentence - A Small Grammar

Let us now turn our attention to a more extensive example. In this section, I will present a small grammar which is just sufficient to parse a very simple declarative sentence, and then trace through the process of parsing the sentence 4.16 immediately below, given this grammar.

4.16 John has scheduled the meeting for Wednesday.

This will be the last example in which I will trace through the particulars of the grammar interpreter's operation in detail. In the examples given in subsequent chapters, the focus will be on the grammar rules and how they interact; the operation of the interpreter will be taken for granted.

As a convention, the parser begins every parse by calling the grammar rule named INITIAL-RULE. This rule creates an S node and activates two packets: CPOOL (Clause POOL), which contains those rules which are always active at the clause level, and SS-START (Simple Sentence-START), which contains rules which decide on the type of simple sentences. The parser's state after this rule is executed is depicted in Figure 4.17. At this point there is no current active node and nothing in the buffer. (The LISP expression "(setq s c)" in rule INITIAL-RULE sets the LISP variable S to the newly created S node. After the parser is finished the parsed structure will remain hanging from this variable for later examination by other subsystems of a full natural language understanding system. This operation cannot, and should not be expressible in PIDGIN, so LISP is used here. While I could edit out such implementation details in the examples given below, I do not; all the rules given below are exactly as they are in the implementation at the time the relevant examples were parsed.)

The Active Node Stack (0. deep)

C: S16 (S) / (CPOOL SS-START)

The Buffer

```
{RULE INITIAL-RULE IN NOWHERE
[t] -->
Create a new s node.
!(setq s c).
Activate  cpool, ss-start.}
```

Figure 4.17 - After INITIAL-RULE has been run.

The packet CPOOL will be ignored in the discussion below; it contains no rules which bear on our example. Furthermore, the contents of the packets shown below will not be exhaustively displayed; only those rules relevant to the discussion will be exhibited. The current grammar, in its complete form, is included as an appendix to this document.

I remind the reader once again of the grammar interpreter's matching rules: that before the grammar interpreter will attempt to match a rule of a given priority, all higher prioritied rules must explicitly fail to match. The reader should also remember that constituents enter the buffer on demand; i.e. that the buffer mechanism will get the next constituent from the sentence list when a rule pattern must be matched against a buffer cell that is currently empty. Thus, throughout the course of the examples in this chapter, constituents will often enter the buffer for no apparent reason. These constituents were requested by rules that then failed to match, leaving no trace of why each constituent entered the buffer. I will not comment further on the entry of constituents into the buffer.

The packet SS-START, some of whose rules are shown in Figure 4.18 below, contains rules which determine the type of a major clause. If the clause begins with an NP followed by a verb, then the clause is labelled a declarative; if it begins with an auxiliary verb followed by an NP, it is labelled a yes/no question. If the clause begins with a tenseless verb, then not only is the clause labelled an imperative, but the word "you" is inserted into the buffer, where it appears at the beginning of the buffer by convention. In this example we will investigate how a simple declarative is parsed; in Chapter 5 we will see what the effect of the rules IMPERATIVE and YES-NO-Q is.

```
{RULE MAJOR-DECL-S IN SS-START
[=np] [=verb] -->
Label c s, decl, major.
Deactivate ss-start. Activate parse-subj.}

{RULE YES-NO-Q IN SS-START
[=auxverb] [=np] -->
Label c s, quest, ynquest, major.
Deactivate ss-start. Activate parse-subj.}

{RULE IMPERATIVE IN SS-START
[=tnsless] -->
Label c s, imper, major.
Insert the word 'you' into the buffer.
Deactivate ss-start. Activate parse-subj.}
```

Figure 4.18 - Some rules that determine sentence type.

After INITIAL-RULE has been executed and packet SS-START has been activated, the rule MAJOR-DECL-S matches. The action of the rule is now run, labelling the clause a major declarative clause, deactivating the packet SS-START, and activating the packet PARSE-SUBJ. The result of this is

shown in Figure 4.19 below.

The Active Node Stack (0. deep)

C: S16 (S) / (CPOOL SS-START)

The Buffer

1 : NP40 (NP NAME NS N3P) : (John)
2 : WORD125 (*HAVE VERB AUXVERB PRES V3S) : (has)

Yet unseen words: scheduled a meeting for Wednesday .

Figure 4.19 - After the rule MAJOR-DECL-S is run.

The packet PARSE-SUBJ contains rules whose job it is to find and attach the subject of the clause under construction. It contains two major rules which are shown in Figure 4.20 below. The rule UNMARKED-ORDER picks out the subject in clauses where the subject appears before the verb in surface order; the rule AUX-INVERSION picks out the subject in clauses where an element of the auxiliary occurs before the subject. As we will see later, the rule UNMARKED-ORDER not only suffices for declarative clauses, but also for imperatives. Though the relevant rules will not be discussed here, the rule UNMARKED-ORDER will pick up the subject of WH-questions where the subject of the clause is questioned, while AUX-INVERSION will pick up the subject of WH-questions that question other than the subject of the clause.

```
{RULE UNMARKED-ORDER IN PARSE-SUBJ
[=np] [=verb] -->
Attach 1st to c as np.
Deactivate parse-subj.
Activate parse-aux.}

{RULE AUX-INVERSION IN PARSE-SUBJ
[=auxverb] [=np] -->
Attach 2nd to c as np.
Deactivate parse-subj. Activate parse-aux.}
```

Figure 4.20 - Two subject-parsing rules.

At this point in the parse, the rule UNMARKED-ORDER now matches, and its action is run. This rule attaches 1st, i.e. NP40, the NP "John", to C, the node S16, as subject. It also activates the packet PARSE-AUX after deactivating PARSE-SUBJ. After this rule has been executed, the interpreter notices that the NP has been attached, and removes it from the buffer. Figure 4.21 shows the state of the parser after this rule is executed.

```
        The Active Node Stack ( 0. deep)

C:      S16 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                NP : (John)

        The Buffer

1 :     WORD125 (*HAVE VERB AUXVERB PRES V3S) : (has)

Yet unseen words:  scheduled a meeting for Wednesday .
```

Figure 4.21 - After UNMARKED-ORDER has been executed.

The process of parsing the auxiliary verbs in this example is exactly parallel to the previous example. Rather than repeat an identical commentary here, the reader is referred to that example. Figure 4.22 is a trace of the application of rules during the parsing of the auxiliary phrase, through the attachment of the auxiliary to the VP node. All the rules referred to in the trace are included in Figures 4.4 and 4.7 in previous sections.

It should be noted that the rules of packet BUILD-AUX, shown previously in Figure 4.4, are the equivalent of the transformational rule of affix-hopping. Note that these rules concisely state the relation between each auxiliary verb and its related affix by taking advantage of the ability to buffer both each auxiliary verb and the following verb. It might seem that some patch is needed to these rules to handle question constructions in which, typically, the verb cluster is discontinuous, but this is not the case, as will be shown in the Chapter 5.

About to run: STARTAUX

The Active Node Stack (0. deep)

C: S16 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
NP : (John)

The Buffer

1 : WORD125 (*HAVE VERB AUXVERB PRES V3S) : (has)

Yet unseen words: scheduled a meeting for Wednesday .

```
About to run: PERFECTIVE

          The Active Node Stack ( 1. deep)

           S16 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                     NP : (John)
C:         AUX14 (PRES V3S AUX) / (BUILD-AUX)

          The Buffer

1 :       WORD125 (*HAVE VERB AUXVERB PRES V3S) : (has)
2 :       WORD126 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words:  a meeting for Wednesday .

-------------------------------------------------------------------------
About to run: AUX-COMPLETE

          The Active Node Stack ( 1. deep)

           S16 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                     NP : (John)
C:         AUX14 (PERF PRES V3S AUX) / (BUILD-AUX)
                     PERF : (has)

          The Buffer

1 :       WORD126 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words:  a meeting for Wednesday .

-------------------------------------------------------------------------
About to run: AUX-ATTACH

          The Active Node Stack ( 0. deep)

C:         S16 (S DECL MAJOR S) / (PARSE-AUX CPOOL)
                     NP : (John)

          The Buffer

1 :       AUX14 (PERF PRES V3S AUX) : (has)
2 :       WORD126 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words:  a meeting for Wednesday .

-------------------------------------------------------------------------
```

The Active Node Stack (0. deep)

C: S16 (S DECL MAJOR S) / (PARSE-VP CPOOL)
NP : (John)
AUX : (has)

The Buffer

1 : WORD126 ("SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words: a meeting for Wednesday .

Figure 4.22 - Parsing the auxiliary of 4.16.

The packet PARSE-VP is now active. This packet contains, among other rules, the rule MVB (Main VerB), which creates and attaches a VP node and then attaches the main verb to it. This rule now matches and is run. While the action of this rule is rather simple grammatically, it clearly demonstrates the behavior of the active node stack and the association between packet activation and the current active node. For this reason, we will examine the action of this rule rather closely. The rule itself, and the resulting state of the parser, is shown in Figure 4.23 below.

```
{RULE MVB IN PARSE-VP
[=verb] -->
Deactivate parse-vp.
If c is major then activate ss-final else
If c is sec then activate emb-s-final.
Attach a new vp node to c as vp.
Attach 1st to c %which is now the vp% as verb.
Activate subj-verb, cpool.}
```

The Active Node Stack (1. deep)

```
          S16 (S DECL MAJOR S) / (SS-FINAL CPOOL)
                    NP : (John)
                    AUX : (has)
                    VP : ↓
C:        VP14 (VP) / (SUBJ-VERB CPOOL)
                    VERB : (scheduled)
```

The Buffer

```
1 :       WORD127 (*A NGSTART DET NS N3P ...) : (a)
```

Yet unseen words: meeting for Wednesday .

Figure 4.23 - The rule MVB and the parser's state after its execution

At the time MVB is executed, the packets PARSE-VP and CPOOL are associated with S16, the current active node, as shown in the last frame of Figure 4.22. The action of MVB first deactivates the packet PARSE-VP, then activates either SS-FINAL, if C is a major clause, or EMB-S-FINAL, if it is an embedded clause. As mentioned earlier in Chapter 3, these two packets both contain rules that parse clause-level modifiers such as prepositional phrases, adverbs, and the like (which is not to say that *all* PPs and adverbs are clause modifiers). They differ in that the rules in EMB-S-FINAL must decide whether a given modifier should be attached to the current clause, or left in the buffer to be attached to a constituent higher up in the parse tree after the current active node is completed. Whatever packet is activated, the

newly activated packet will be associated with C, S16, and thus the grammar interpreter will attempt to match the rules in it whenever S16 is the current active node.

The next line in MVB, which reads "Attach a new vp node to c as vp.", is a special PIDGIN construction. Normally, whenever a new node is created, as is done by the phrase "a new vp node", the new node is pushed onto the active node stack immediately after its creation, thus immediately becoming a new current active node. If this were *always* true, however, there would be no way to immediately attach a new node to the node that was the current active node at the time of its creation. It would be necessary to always drop the new node into the buffer and only then attach it to the old (and once again) current active node. Thus, the construction

Attach a new <type> node to c as <type>

is specially defined to mean that 1) a new node should be created but *not* immediately pushed onto the stack, after which 2) it should be attached to the current active node, and only then 3) it should be pushed onto the active node stack, becoming the new current active node. The execution of the instance of this construction in the action of MVB results in a new VP node, VP14, attached to S16 and now the current active node, while S16 is immediately above it in the stack.

WORD126, the verb "scheduled" is now attached to the VP.

Finally, the action of MVB activates the packets SUBJ-VERB and CPOOL. As is always the case with packet activation, these are associated

with the node which is the current active node at the time of their activation, in this case VP14. Thus, this rule leaves the parser with the packets CPOOL and SS-FINAL associated with S16 (the prior association of CPOOL with S16 having been unaltered by the execution of this rule), and the packets CPOOL and SUBJ-VERB associated with VP14. Since VP14 is the current active node, CPOOL and SUBJ-VERB are now active. Once VP14 is popped from the stack, leaving S16 the current active node, CPOOL and SS-FINAL will be active.

The packet SUBJ-VERB contains rules which involve the deep grammatical relation of the surface subject of the clause to the verb, and which set up the proper environment for parsing the objects of the verb. The major rule in this packet, the rule SUBJ-VERB is shown in Figure 4.24 below. (More of the rules in this packet will be presented in later sections.) While most of the code of the action of the rule SUBJ-VERB does not apply to our example, I have refrained from abbreviating this rule to give a feel for the power of PIDGIN.

```
{RULE SUBJ-VERB PRIORITY: 15. IN SUBJ-VERB
[t] -->
%The next line really belongs in the case frame demons.
It appears here for reasons having to do with implementation
peculiarities and will not be discussed in the text.%
If c is not np-preposed
        and there is a np of the s above c then
        it fills the subj slot of the cf of c.
%Activate packets to parse objects and complements.%
If the verb of c is inf-obj then activate inf-comp.
If the verb of c is to-less-inf-obj then activate to-less-inf-comp.
If the verb of c is that-obj then activate that-comp.
If there is a wh-comp of the s above c
        and it is not utilized then activate wh-vp
        else if the s above c is major then activate ss-vp
        else activate embedded-s-vp.
Deactivate subj-verb.}
```

Figure 4.24 - The rule SUBJ-VERB.

The rule SUBJ-VERB is now the rule of highest priority that matches, so its action is now executed. The action of this rule is rather complex (indeed, it is currently the most complex rule in the grammar), so I will discuss what it does in some detail.

The first sentence of the rule actually does not belong in this rule at all; it should be part of the case frame mechanism, which is discussed briefly in Appendix E. This sentence appears in this rule because of a weakness in that mechanism which is irrelevant to the issues discussed in this paper, and nothing more will be said about it.

Most of the remainder of the rule activates the appropriate packets to parse the objects and complements of a clause, some of which depend on the verb of the clause and some of which depend on global properties of the

clause. Thus, the next several lines of the action activate packets for various sorts of complement constructions that a verb might take: infinitive phrases in general (the packet INF-COMP), infinitive phrases that are not flagged by "to", e.g. "I saw John give Jane a kiss." (the packet TO-LESS-INF-COMP in addition to INF-COMP), and that-phrases (the packet THAT-COMP). The next long clause activates one of a number of packets which will attach the objects of the verb. The packet activated depends on the clause type: whether this clause still has a WH-head that needs to be utilized (WH-VP), whether this clause is secondary without a WH-head (EMBEDDED-S-VP), or whether this clause is a major clause (SS-VP).

This rule provides a good example of one difference between the packets used to organize this grammar and the states of an ATN. Packets do *not* simply correspond to ATN states, for several packets will typically be active at a time. For instance, if parsing the sentence "Who did John see Jane kiss?", this rule would activate three packets in addition to CPOOL: INF-COMP, TO-LESS-INF-COMP, and WH-VP.

Of the complement-initiating packets, only the packet INF-COMP is activated in our example, since "schedule" can take infinitive complements, as in "Schedule John to give a lecture on Tuesday.". The packet SS-VP is then activated to attach the verb's objects, the packet SUBJ-VERB is deactivated, and the rule SUBJ-VERB is through. This rule thus changes the state of the parser only by activating and deactivating packets; the packets now associated with the current active node are shown in Figure 4.25, the state of the parser after SUBJ-VERB has been executed.

```
          The Active Node Stack ( 1. deep)

           S16 (S DECL MAJOR S) / (SS-FINAL CPOOL)
                    NP : (John)
                    AUX : (has)
                    VP : ↓
C:         VP14 (VP) / (SS-VP INF-COMP CPOOL)
                    VERB : (scheduled)

          The Buffer

1 :        NP41 (MODIBLE NS INDEF DET NP) : (a meeting)
2 :        PP11 (PP) : (for Wednesday)
3 :        WORD133 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)
```

Figure 4.25 - After the rule SUBJ-VERB has been executed.

The packet SS-VP is shown below in Figure 4.26; all of its rules will come into play in the example below. Note that this packet contains a default rule, the rule VP-DONE, which will become active when no other rule can apply, completing the VP.

```
{RULE OBJECTS IN SS-VP
[=np] -->
Attach 1st to c as np.}

{RULE VP-DONE PRIORITY: 20 IN SS-VP
[t] --> Drop c.}

{RULE PP-UNDER-VP-1 IN SS-VP
[=pp] -->
If 1st fits a pp slot of the cf of c
        then attach 1st to c as pp
else run vp-done next.}
```

Figure 4.26

The rule OBJECTS is now triggered by NP41, and attaches the NP to VP14. The state of the parser at this point is shown in Figure 4.27 below.

```
                The Active Node Stack ( 1. deep)

                S16 (S DECL MAJOR S) / (SS-FINAL CPOOL)
                        NP : (John)
                        AUX : (has)
                        VP : ↓
C:              VP14 (VP) / (SS-VP INF-COMP CPOOL)
                        VERB : (scheduled)
                        NP : (a meeting)

                The Buffer

1 :             PP11 (PP) : (for Wednesday)
2 :             WORD133 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)
```

Figure 4.27 - After the rule OBJECTS has been run.

Now the rule PP-UNDER-VP-1 is triggered by PP11 sitting at the beginning of the buffer. This rule is one of a large set of PP attachment rules. These rules have the responsibility for calling semantics to decide whether a given PP is appropriate at a given location in the parse tree or whether that PP is best attached elsewhere. This rule is the simplest of the set, since a PP that is encountered while parsing the VP of a major, i.e. "toplevel", clause, must either attach to the VP or to the S node above it. (The difference in attachment points, briefly, is this: If a PP attaches to a VP node, then it fills an "intrinsic case" slot of the verb that is the head of the VP. If it attaches to the S node, then it serves as a general modifier of the clause. As examples, PPs that serve as general place and time modifiers attach to the S node, while PPs that serve as Instruments, Datives, and the like attach to the VP node. It should be noted that "schedule" takes a time

both as an intrinsic case, and as a modifier, thus the time phrase in 4.28.a below serves as intrinsic time in that sentence's primary reading, i.e. it fills a case slot of schedule, while the time phrase in 4.28.b serves as a general modifier of the event itself, parallel to the role filled by the time phrase in 4.28.c. Some people may get a secondary reading for 4.28.a that is a paraphrase of 4.28.b, but that is rare.

a) Schedule an appointment for John before 5 o'clock today.
b) Before 5 o'clock today, schedule an appointment for John.
c) Take out the garbage before 5 o'clock today.

Figure 4.28

The rule PP-UNDER-VP-1 includes a test that calls semantics to see if 1st, i.e. the PP, can serve semantically in any of the (intrinsic) case slots of the case frame associated with the VP node. If it can, it is attached to the VP, otherwise it is left for clause level PP rules to pick up. Essentially, this rule states that if a PP can serve as a case filler of the verb of a major clause, then it will always serve as a case filler even if it could serve as a modifier of the entire clause. Sentence 4.28.1 serves as evidence that this rule is approximately correct. As was pointed out above, one of the intrinsic cases of the verb "schedule" is a TIME, and this PP can serve to fill this slot, so the semantic test succeeds, and the PP is attached to the VP. The result of this rule is shown in Figure 4.29 below.

It should be noted that the form of semantic test involved in this rule is the simplest of the range of tests that seem to be necessary to do deterministic parsing. As noted in Chapter 1, the general problem of PP

attachment seems to require extremely complex semantic interactions, and will not be taken up in this paper. For discussion of some of the types of semantic interactions that are necessary for a deterministic parser and that have been included in this research, see Chapter 10 and Appendix A.

The Active Node Stack (1. deep)

S16 (S DECL MAJOR S) / (SS-FINAL CPOOL)
NP : (John)
AUX : (has)
VP : ↓
C: VP14 (VP) / (SS-VP INF-COMP CPOOL)
VERB : (scheduled)
NP : (a meeting)
PP : (for Wednesday)

The Buffer

1 : WORD133 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)

Figure 4.29 - After rule PP-UNDER-VP-1 has been executed.

Completing the parse is now a simple matter. The default rule in packet SS-VP, VP-DONE now triggers, popping the VP node from the active node stack. Since VP14 is attached to S16, the S node above it on the stack, it remains hanging from the S node. The resulting state of the parser is shown in Figure 4.30.a. The S node is once again the current active node, and packet SS-FINAL is now active. The default rule in this packet, SS-DONE, shown in Figure 4.30.b, immediately triggers. This rule attaches the final punctuation mark to the clause, pops the S node from the stack, and signals the grammar interpreter that the parse is now complete. This final state is shown in Figure 4.30.c.

The Active Node Stack (0. deep)

C: S16 (S DECL MAJOR S) / (SS-FINAL CPOOL)
NP : (John)
AUX : (has)
VP : (scheduled a meeting for Wednesday)

The Buffer

1 : WORD133 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)

(a) - Before the rule S-DONE is executed.

```
{RULE S-DONE  IN SS-FINAL
[=finalpunc] -->
Attach 1st to c as finalpunc.
%The next line is really part of the cf mechanism.%
Finalize the cf of s.
Parse is finished.}
```

(b) - The rule S-DONE

The Active Node Stack (0. deep)

C: S16 (S DECL MAJOR S) / (SS-FINAL CPOOL)
NP : (John)
AUX : (has)
VP : (scheduled a meeting for Wednesday)
FINALPUNC: (.)

The Buffer

Yet unseen words: (none)

(c) - After S-DONE has been run, completing the parse.

Figure 4.30

In conclusion, Figure 4.31 below is a summary of the rules introduced in this chapter presented in the format introduced in Figure 4.2. Node types and packets are shown in upper case; rule names are shown in lower. Priorities are only shown for rules with a priority other than the default priority of 10. The asterisk marks rules that were used in this chapter.

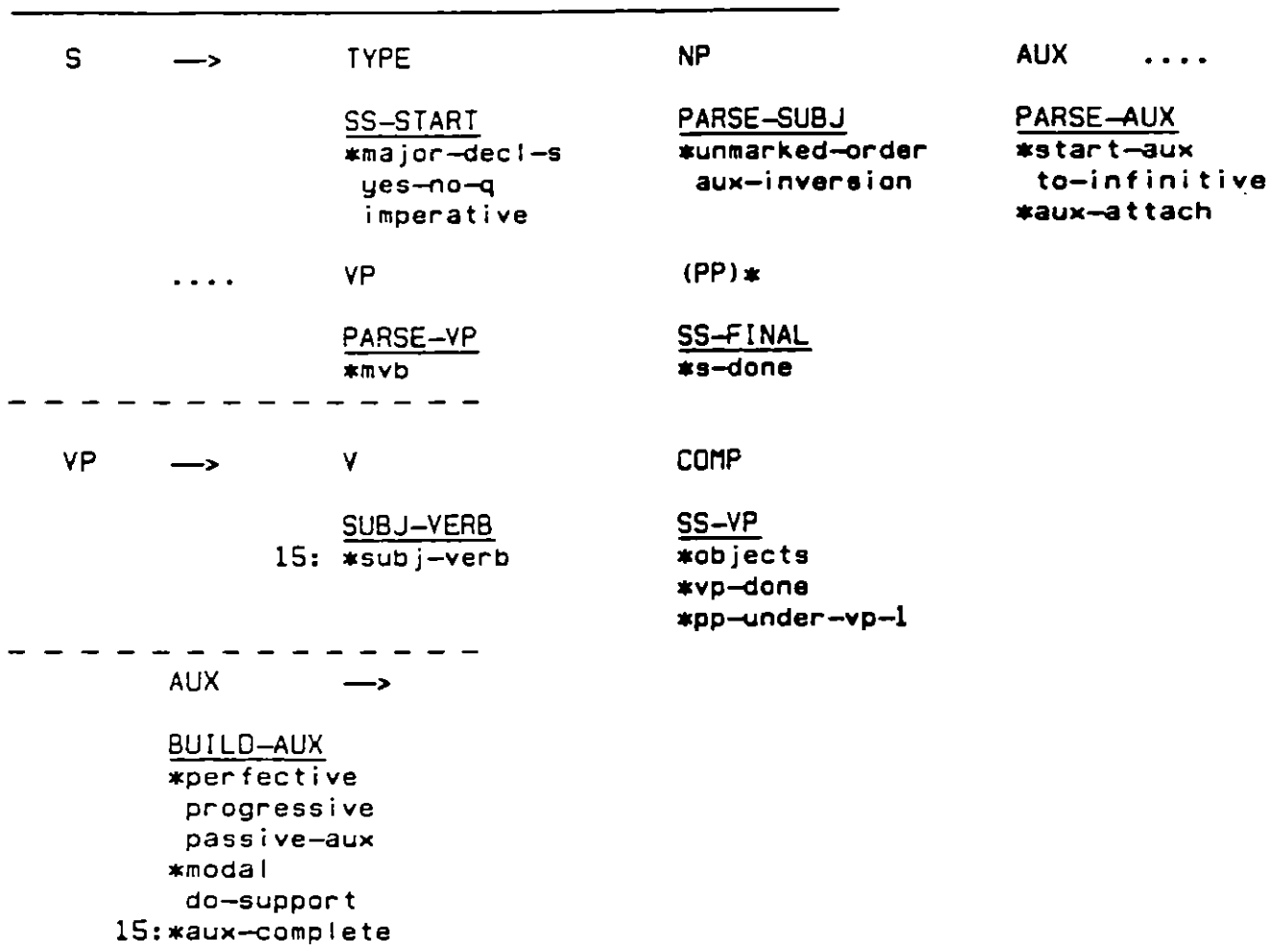


Figure 4.31 - The structure of the rules presented in this chapter.

CHAPTER 5

CAPTURING LINGUISTIC GENERALIZATIONS

Introduction

In this chapter we will shift our attention from the grammar interpreter to the organization of the grammar itself and the form of the linguistic structures that it builds. In the first part of the chapter, I will discuss the general grammatical framework upon which the grammar rests. The following several sections will show that the grammar formalism is capable of capturing many of the same generalizations that are captured by traditional generative grammar, with much the same elegance.

As I argued in Chapter 1, showing that a parser for English can parse a range of English sentences cannot by itself serve as an argument for any particular parsing technique. What must also be demonstrated is that the parsing technique lends itself to the writing of grammars that succinctly capture the generalizations that underlie the grammar of the language. This chapter is intended to be the first part of a demonstration of exactly this point. The demonstration will continue with the next two chapters, in which I will argue that two of the three constraints on transformations that stand at the core of Chomsky's current theory of grammar [Chomsky 73, 75a, 76, to appear; Chomsky & Lasnik 77] fall out naturally from the structure of the grammar interpreter, at least in part.

The General Grammatical Framework

The form of the structures that the current grammar builds is based on the notion of *Annotated Surface Structure*. This term has been used in two different senses by Winograd [Winograd 71] and Chomsky [Chomsky 73, etc].; the usage of the term here can be thought of as a synthesis of the two concepts. Following Winograd, this term will be used to refer to a notion of surface structure *annotated by the addition of a set of features* to each node in a parse tree. Following Chomsky, the term will be used to refer to a notion of surface structure annotated by the addition of an element called *trace* to indicate the "underlying position" of "shifted" NPs. I believe that a grammatical representation that is annotated in both these ways can be made to explicitly represent more grammatical information for use in subsequent semantic processing than can be captured by either annotation method in isolation.

In an interesting convergence of viewpoints, the belief that the semantic interpretation of grammatical structure is best done from a properly annotated surface structure analysis, rather than a deep structure analysis, now seems to be shared by both linguists and workers in artificial intelligence. Winograd rejected an explicit deep structure analysis as the basis for semantic interpretation in favor of an annotated surface structure in 1970; Chomsky took the same step, although for different reasons, and with a different style of annotation several years later. (It should be noted, however, that most linguists and many natural language researchers still regard deep structure as the proper level for semantic analysis.)

There are several reasons for choosing a properly annotated surface structure as a primary output representation for syntactic analysis. While a deeper analysis is needed to recover the predicate/argument structure of a sentence (either in terms of Fillmore case relations [Fillmore 68] or Gruber/Jackendoff "thematic relations" [Gruber 65; Jackendoff 72]), phenomena such as focus, theme, pronominal reference, scope of quantification, and the like can be recovered only from the surface structure of a sentence. By means of proper annotation, it is possible to encode in the surface structure the "deep" syntactic information necessary to recover underlying predicate/argument relations, and thus to encode in the same formalism both deep syntactic relations and the surface order needed for pronominal reference and the like. This realization has led Chomsky to advocate semantic analysis from annotated surface structure, although a level of deep structure still figures prominently in his current formulation of grammatical theory.

(It should be mentioned that while both surface order and deep grammatical relations are encoded in the surface structure of the sentence, the parsing system that I have implemented includes a component that builds an explicit case representation of the sentence in parallel with building an annotated surface structure. Rather than continually decoding the case information implicitly available in the annotated surface structure, it makes sense in terms of system design to explicitly represent this case information in a separate representation. The case frame mechanism also provides a computational check that the predicate-argument structure of an utterance can in fact be easily retrieved from an annotated surface structure

representation. In this sense, the output of the parsing system is *both* an annotated surface structure representation and a case frame representation of the input sentence.

The case frame builder runs in parallel with the parser by means of a set of programs that monitor some of the types of actions the parser can take; a monitor program can be written so that it will be executed whenever a node of a given type is created or whenever a node of type A is attached to a node of type B. These monitors are written in an extension of PIDGIN that amounts to a command language for an independent case frame interpreter. The case frame interpreter accesses case frame and semantic marker information stored in the parser's lexicon. All interaction with "deep semantics" that occurs during the parsing process (as will be discussed in Chapter 10) is done with this case frame component as intermediary. See Appendix E for further description of the case frame component; however the structure of this case frame mechanism is independent of the issues discussed here.)

Winograd's Use of Features

Winograd uses the term annotated surface structure to refer to surface structures in which deep grammatical relations are encoded by the inclusion of a set of grammatical features at each node of the parse tree. These features encode both grammatical properties of the constituent under that node, and the functional use of the constituent in the surrounding grammatical context. (I should note here that Winograd uses features within a grammatical framework based on the theory of systemic grammar developed

by Halliday and others of the London school of linguistics [Halliday 67; Hudson 71]. Explicit reasons why this approach was not utilized in this work will be mentioned throughout the chapter as they become relevant.)

As an example of Winograd's use of features, consider the features, shown in Figure 5.1.b below, that his system assigns to the relative clause shown in 5.1.a. The relative clause as a whole is marked as a *CLAUSE* that serves functionally as a *Rank Shifted Qualifier (RSQ)*, by virtue of being a WH-phrase (*WHRS*). The feature *DOWNREL* flags the fact that the relative head has been "moved" from a lower clause, in this case the infinitive phrase "to pick up". The infinitive phrase itself is given the feature *RSNG* to flag the fact that it serves functionally as a Rank Shifted Noun Group. The features *UPREL* and *OBJ1REL* encode the fact that this clause utilizes the relative head of a dominating clause as this clause's first object.

(a) ...which I wanted you to pick up

(b) (CLAUSE RSQ WHRS ... DOWNREL)
which I wanted you to pick up
(CLAUSE RSNG ... UPREL OBJ1UPREL)
to pick up

Figure 5.1 - An example of Winograd's use of features.

The use that PARSIFAL makes of features is somewhat different than Winograd's. Features are used here neither to indicate the functional use of constituents in a larger context (which can be termed the "extrinsic" grammatical properties of a constituent, following Sussman & Brown's terminology in the domain of electronics [Sussman & Brown 74]) nor to encode the position of deleted constituents. The primary use of features is to

summarize the important grammatical properties of a constituent's internal structure (its "intrinsic" grammatical properties) in easily accessible form. The idea is that a constituent's feature set should include markers for all those grammatical properties that later syntactic and semantic analysis routines may wish to have visible at a glance without the need to examine the constituent's internal structure. In a sense these features are *explicitly* included for the sake of efficiency; rather than have programs check sets of features, one could provide a set of predicates that actively compute whether a given feature is or is not true of any given grammatical structure. Including feature sets has the added advantage that those grammatical properties about which the system has knowledge are explicitly, rather than implicitly, present.

One reason that functional information is not encoded in a constituent's feature set is that the grammatical function of a constituent seems to be clearly indicated by its position in tree structure. (In this I follow [Chomsky 65].) Thus, the subject of a sentence is taken to be the NP directly under the S node; a direct object is taken to be the rightmost NP directly dominated by the VP under S. Similarly, an indirect object is an NP directly dominated by VP that is between the verb and an NP node. It would seem that Winograd chose to encode this information in the feature system because the tree structures that systemic grammar advocates are far shallower and bushier than the trees that are typically advocated by generative grammarians. The net result is that the position of a constituent in a tree provides far less information about its functional role, thus necessitating some auxiliary method for encoding functional relations.

Traces and Annotated Surface Structure

As an alternative to Winograd's use of features to indicate the position of constituents that have been "displaced" from their underlying positions, the grammar uses the device of *traces*, first argued for by Robert Fiengo [Fiengo 74]. In current linguistic theory, a trace is essentially a "phonologically null" NP in the surface structure representation of a sentence that has no daughters but is "bound" to the NP that filled that position at some level of "underlying structure". In a sense, a trace can be thought of as a "dummy" NP that serves as a placeholder for the NP that earlier filled that position; in the same sense, the trace's binding can be thought of as simply a pointer to that NP. It should be stressed at the outset, however, that a trace is indistinguishable from a normal NP in terms of normal grammatical processes; a trace *is* an NP, even though it is an NP that dominates no lexical material.

Some examples of the use of trace are given in Figure 5.2 immediately below.

(1a) What did John give to Sue?
(1b) What did John give *t* to Sue?
(1c) John gave *what* to Sue.

(2a) A book was given Sue.
(2b) A book was given Sue *t*.
(2c) ∇ gave Sue *a book*.

(3a) John was believed to be happy.
(3b) John was believed [$_S$ *t* to be happy].
(3c) ∇ believed *John* to be happy.

(4a) John was believed to have been shot.
(4b) John was believed [$_S$ t_1 to have been shot t_2].
(4c) ∇ believed ∇ to have shot John.
(4d) ∇ believed [$_S$ John to have been shot t_2].

Figure 5.2 – Some examples of the use of trace.

One use of trace is to indicate the underlying position of the wh-head of a question or relative clause. Thus, the structure built by the parser for 5.2.1a would include the trace shown in 5.2.1b, with the trace's binding shown by the line under the sentence. The position of the trace indicates that 5.2.1a has an underlying predicate/argument structure analogous to the overt surface structure of 5.2.1c. (The declarative form is used in 5.2.1c only because it most simply illustrates the underlying predicate/argument structure of the original sentence; the predicate/argument structure of a clause clearly has nothing whatever to do with whether the clause is a declarative or a question, or whatever.)

Another use of trace is to indicate the underlying position of the surface subject of a passivized clause (or, technically speaking, any np-

preposed clause). For example, 5.2.2a will be parsed into a structure that includes a trace as shown as 5.2.2b; this trace indicates that the subject of the passive has the underlying position shown in 5.2.2c. The symbol "∇" signifies the fact that the subject position of (2c) is *empty* in underlying structure; it is filled by an NP that dominates no lexical structure. (Again, following Chomsky, I assume that a passive sentence in fact has *no underlying subject*, that an agentive "by NP" prepositional phrase originates as such in underlying structure.) The trace in (3b) indicates that the phrase "to be happy", which the brackets show is really an embedded clause, has an underlying subject which is identical with the surface subject of the matrix S. (By *matrix S*, I mean a clause that dominates an embedded complement.) As (3c) shows, what is conceptually the underlying subject of the embedded clause has been passivized into subject position of the matrix S, a phenomenon commonly called "raising". The analysis of this phenomenon assumed here derives from [Chomsky 73]; it is an alternative to the classic analysis which involves "raising" the subject of the embedded clause into object position of the matrix S before passivization (for details of this later analysis see [Postal 74]). I will have much to say about this phenomenon later in this chapter.

As the example presented in 5.2.4a-d shows, traces can be bound to other traces. In this case, 5.2.4a has the annotated surface structure indicated by (4b), with trace t_2 bound to trace t_1, and t_1 bound to the NP "John". The underlying structure of this sentence is as shown in (4c). Adopting the viewpoint of generative grammar for the moment, looking at the transformational derivation of an annotated surface structure from an

underlying deep structure, these two traces arise in the following manner: First, the NP "John" is moved from its underlying position into the subject position of the embedded S by the passivization of the embedded clause, leaving behind t_2. This is shown in (4d). Then the matrix S is passivized, moving the NP "John" again, this time leaving behind trace t_1, and resulting in the structure indicated by (4b). (This explanation ignores much important detail for the sake of simplicity.) Note that I was speaking rather loosely above when I referred to a trace as pointing to the NP that filled that position in *underlying structure*. In point of fact, a trace may very well point to an NP that filled that position temporarily in the course of the transformational derivation.

It should be noted that the information that these traces encode is easily used by a semantic component to recover the underlying predicate/argument structure of an utterance. Once a parser has constructed the annotated surface structure of a sentence, it is not too large a jump to compute its predicate/argument structure. For example, all that needs to be done to recover the predicate/argument structures of the (b) sentences above (where the predicate/argument structure is taken here loosely to be analogous to the structure of the (c) sentences) is to repeatedly replace each trace with its binding until all traces have been eliminated, and to ignore the surface subjects of passivized clauses.

While the notion of a trace and its binding is discussed above in terms of the theory of generative grammar, later sections of this chapter will show a version of this notion to be extremely powerful within the theory of

parsing presented here. In the context of this theory of parsing, I will define a trace simply to be an NP which has no daughters and which has associated with it a *binding register* which can be set to a pointer to another NP. (Within the grammar presented below, a trace will also be flagged by the feature *TRACE*.) This register can be set by the PIDGIN code

Set the binding of <node> to <controlling node>.

The NP to which a trace's binding register points will be referred to as the *binding of the trace*. (Note: It should be noted that a trace is *not* a pointer in the current generative formulation of trace theory; why this is so is too complex a matter to discuss here. Suffice it to say that I have not seen a published coherent formalization of exactly what the binding of a trace is, and therefore feel free to take the position that I do here.)

Because the binding of a trace will often be another trace, it will be useful to define a notion of the *binding** of a trace. By *binding**, I mean the transitive closure of the binding relationship, defined recursively as follows: The binding* of an NP that is not a trace is the NP itself; the binding* of a trace is the binding* of its binding. Thus, in the example of 5.2.4b above, the binding and the binding* of t_1 is the NP "John", the binding of t_2 is t_1 while its binding* is "John", and the binding* of the NP "John" is itself, although it has no binding. The binding* of a node is returned as the value of the PIDGIN phrase

the binding of <node>.

There is no PIDGIN which returns the simple binding of a node because I have not found a need for such a primitive.

Before leaving this topic, I should indicate why traces were chosen over Winograd's use of features to encode the underlying position of displaced constituents. There are two primary reasons; each will be discussed in turn.

Consider the example of Winograd's use of traces to encode this information given in Figure 5.1, repeated below for convenience.

(a) ...which I wanted you to pick up

(b) (CLAUSE RSQ WHRS ... DOWNREL)
which I wanted you to pick up
(CLAUSE RSNG ... UPREL OBJ1UPREL)
to pick up

Figure 5.1 (repeated) - An example of Winograd's use of features.

The ultimate purpose of these features is to allow the underlying position of the displaced constituent to be computed. Let us examine how they must be used to do this computation. The feature *DOWNREL* tells a hypothetical semantic interpreter that it must find an embedded clause and look within it to find the underlying site of the WH-head. Finding an embedded clause and looking there, the semantic interpreter finds the feature *UPREL*, which confirms that this is the clause from which the WH-head has been displaced. It then finds the feature *OBJ1UPREL*, which indicates that the WH-head has been displaced from the first object position following the verb. Now it knows that the underlying object of this clause is in fact the WH-head. But this is exactly what a trace tells us immediately. The point is that the facts encoded by Winograd's features, that the head of this relative clause is used in a lower clause, or that a relative head used in the lower clause originated

in a higher clause are only of secondary importance; they are useful only to help deduce the primary fact - exactly where in the underlying tree structure the WH-head has been displaced from.

The central argument, however, for the use of traces in the context of PARSIFAL has to do not so much with the simplicity of the structure that results once the parser has completed its analysis, but rather with one central property of traces in terms of the parsing process itself: Once a trace has been attached to a constituent, or dropped into the buffer by a grammar rule, later grammatical processes will be unaware that an NP did not actually appear in this position in the input structure. Furthermore, by dropping a trace into the buffer, a grammar rule can assert, in effect, that an NP underlyingly appears at a given linear position in the input string without committing itself to the position of this NP in the underlying tree structure. While this may appear to be of dubious value out of context, this fact is crucial to the formulation of the grammar rules which will be presented in the remainder of this chapter. As will be demonstrated in what follows, this fact allows extremely simple grammar rules to capture complex generalizations about the structure of natural language.

The Grammar Elegantly Captures Linguistic Generalizations

Given the grammatical devices described above and the grammar formalism for rules described in earlier chapters, rules of grammar can be formulated that elegantly capture a wide range of linguistic phenomena. As will be demonstrated below, rules can be be written for the grammar

interpreter that explicitly capture, on nearly a one-to-one basis, the same generalizations that are typically captured by classical transformational rules. (By "classical", I mean the set of transformations commonly accepted in '68, '69 before the fragmentation of generative grammar into several widely divergent camps; such a set of transformations is presented in [Akmajian & Heny 75]). The central point I hope to make is that the availability of the buffer as a work space, in conjunction with a grammar written in the form of pattern-action rules, makes possible several techniques for writing simple, concise grammar rules that have the net effect of explicitly "undoing" the generative grammarian's transformations, thus capturing many of the same linguistic generalizations.

I should make it clear at the outset that not all grammatical processes which are typically expressed as single rules within the generative framework can be so captured within the grammar for this, or - I believe - any, parser. Such processes include the general phenomenon of "WH-movement", which accounts for the structure of WH-questions and relative clauses (at the least), and the problem of prepositional phrase attachment. Thus, while there is a wide range of grammatical generalizations that can be captured within a parsing grammar, it must be conceded that there are important generalizations that cannot be captured within this framework.

In the following sections we will explore a small range of the rules in the current grammar by focusing on the process of parsing the sentences shown in Figure 5.3 below:

(a) John has scheduled the meeting for Wednesday.
(b) Has John scheduled the meeting for Wednesday?
(c) Schedule the meeting for Wednesday.
(d) The meeting was scheduled for Wednesday.
(e) The meeting seems to have been scheduled for Wednesday.

Figure 5.3 - The key examples of this chapter.

The sentence given as 5.3.a above was discussed earlier in Chapter 4; it will serve as a benchmark against which the complexity of the parsing process for the remaining sentences will be judged. Once again, it will be assumed that NPs are built by the parser in a manner transparent to the processes that we consider here; we will investigate only the clause level parsing processes in this example. It will also be assumed that PPs as well as NPs are constructed before they come to the attention of the processes we consider here.

There are several techniques made possible by the buffer that will be used repeatedly to capture linguistic phenomena within fairly simple rule formulations. They are:

1) The ability to remove some constituent other than the first from the constituent buffer, using the Delete buffer primitive, compacting the buffer and thereby reuniting discontinuous constituents. In natural language, it is often the case that some third structure intervenes between two parts of what is intuitively one constituent. For example, in yes-no questions like 5.3.b above, the verb cluster is divided into two pieces with the subject intervening. In most parsers, special provisions must be made in the

grammar for handling such situations. As will be demonstrated below, the buffer mechanism makes this unnecessary.

2) The ability to place specific lexical items into the input stream using the Insert primitive, thereby allowing one set of rules to operate on only superficially different cases. Many grammatical constructions in natural language are best analyzed as slight variants of other constructions, differing only in the occurrence of an additional specific lexical item or two. Several examples are shown in Figure 5.4 below. Given the buffer mechanism, such constructions can be easily handled by doing a simple insertion of the appropriate lexical items into the shorter form of the construction, "transforming" the shorter form into the longer form, allowing both cases to then be handled by the same grammar rule.

1(a) all the boys
(b) all *of* the boys

2(a) I helped John pick it up.
(b) I helped John *to* pick it up.

3(a) John seems happy.
(b) John seems *to be* happy.

Figure 5.4

---

3) The ability to place a trace by inserting it into the buffer rather than by directly attaching it to a tree fragment. The explanation of exactly why this is a powerful and useful ability will be the central theme - at least implicitly - of the next two chapters as well as much of this chapter.

As the reader will soon see, this is perhaps the most significant single property of the grammar interpreter.

We will see examples of the use of these abilities in the sections that follow.

Parsing a Yes-No Question

In this section I will show that the syntactic difference between declaratives and yes-no questions can be captured briefly and perspicuously in the parsing grammar. As will be demonstrated below, the analysis of a yes-no question differs from the analysis of the related declarative only in the execution of two rules for each sentence type. While declaratives trigger the two rules shown in Figure 5.5.a below, yes-no questions trigger the two rules shown in Figure 5.5.b. These rules, which differ only in their patterns and in two lines of code (underlined in fig. 5.5), embody the only differences between the analyses of declaratives and yes-no questions.

```
{RULE MAJOR-DECL-S IN SS-START
[=np] [=verb] -->
Label c s, decl, major.
Deactivate ss-start.
Activate parse-subj.}

{RULE UNMARKED-ORDER
        IN PARSE-SUBJ
[=np] [=verb] -->
Attach 1st to c as np.
Deactivate parse-subj.
Activate parse-aux.}

DECLARATIVES
      (a)
```

```
{RULE YES-NO-Q IN SS-START
[=auxverb] [=np] -->
Label c s, quest, ynquest, major.
Deactivate ss-start.
Activate parse-subj.}

{RULE AUX-INVERSION
      IN PARSE-SUBJ
[=auxverb] [=np] -->
Attach 2nd to c as np.
Deactivate parse-subj.
Activate parse-aux.}

YES-NO QUESTIONS
      (b)
```

Figure 5.5 - Grammar rules for yes-no questions and declaratives.

The differences between the two sets of rules above are the following: 1) MAJOR-DECL-S will label a clause beginning with an NP followed by a verb as declarative; YES-NO-Q will label a clause beginning with an auxiliary verb followed by an NP as a yes-no question. 2) UNMARKED-ORDER will trigger if an NP is followed by a verb at the beginning of a clause, and will parse the NP as the subject of the clause; AUX-INVERSION triggers if an auxiliary verb is followed by an NP, and parses the NP as subject. It should be noted that if these rules were being written to handle *only* declaratives and yes-no questions, the two rules for each clause type could be conflated into one rule for each type. However, the two subject parsing rules also handle imperatives (as we will see in the following section) and wh-questions, and thus a generalization is captured by initiating the parsing of a major clause in two steps.

What is surprising about these rules is that they obviate the need

for the grammar to contain special provisions to handle the discontinuity of the verb cluster in yes-no questions. Consider the following sentence, previously given in Figure 5.3.b above:

5.3.b) Has John scheduled the meeting for Wednesday?

One of the auxiliary parsing rules that should be triggered during the course of the analysis of this sentence is the rule PERFECTIVE, repeated below in Figure 5.6. This rule will attach any form of "have" to the auxiliary and label the auxiliary perfective if the *following* word (implicitly a verb) has the feature *en*. It would seem that some provision must be made for the fact that in a yes-no question these two words might be separated by the subject NP, as in our example where the verb "scheduled" (which does carry the feature *en*, since *en* is morphologically realized with this verb as "-ed") does not follow "has", but rather follows an intervening NP. As we shall see immediately below, however, no special patch is needed to handle this discontinuity at all.

{RULE PERFECTIVE PRIORITY: 10. IN BUILD-AUX
[=*have] [=en] --> Attach 1st to c as perf. Label c perf.}

Figure 5.6 - The PERFECTIVE rule requires contiguous verbs.

Let us now trace through the initial steps of parsing 5.3.b and see why it is that no changes to the auxiliary parsing rules are required to parse yes-no questions. We will focus only on significant details of the analysis in this and the following examples. Again, I remind the reader that Appendix B discusses the details of PIDGIN, and Appendix C contains a compendium, organized by packet, of all the rules referenced in this chapter.

We begin with the parser in the state shown in Figure 5.7.a below, with the packet SS-START active. The rule YES-NO-Q matches and is executed, labelling S17 with features that indicate that it is a yes-no question, as shown in Figure 5.7.b below. (The buffer, not shown again, remains unchanged.) This step of the parsing process is quite analogous to the analysis process for declaratives.

The Active Node Stack (0. deep)

C: S17 (S) / (CPOOL SS-START)

The Buffer

1 : WORD134 (*HAVE VERB AUXVERB PRES V3S) : (Has)
2 : NP43 (NP NAME NS N3P) : (John)
3 : WORD136 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words: a meeting for Wednesday ?

(a) - Before YES-NO-Q has been executed.

The Active Node Stack (0. deep)

C: S17 (S QUEST YNQUEST MAJOR) / (PARSE-SUBJ CPOOL)

(b) - The Active Node Stack after YES-NO-Q is executed.

Figure 5.7

The packet PARSE-SUBJ is now active, and rule AUX-INVERSION matches and is executed; it attaches NP43 to S17. After AUX-INVERSION has been executed, the grammar interpreter notices that NP43 is attached and *it therefore removes NP43 from the buffer.* But now that the subject of the clause has been removed from the buffer, the pieces of the verb cluster "has scheduled" are no longer discontinuous, as the word "has" is now in the 1st buffer cell, and "scheduled" is in the 2nd cell. This is shown in Figure 5.8

below. In effect, the rule AUX-INVERSION, merely by picking out the subject of the clause, has "undone" the "inversion" of the subject of the clause and the first element of the auxiliary which signals the presence of a question. The configuration of the parser at this point is now indistinguishable from what its state would have been had the sentence been a declarative rather than a yes/no question, with the obvious exception of the question features with which the clause node S17 has been labelled. This can be seen by comparing Figure 5.8 below with Figure 4.21, a snapshot of the parsing of the declarative which "underlies" this yes-no question.

The Active Node Stack (0. deep)

C: S17 (S QUEST YNQUEST MAJOR) / (PARSE-AUX CPOOL)
NP : (John)

The Buffer

1 : WORD134 (*HAVE VERB AUXVERB PRES V3S) : (Has)
2 : WORD136 (*SCHEDULE COMP-OBJ VERB INF-OBJ V-3S ...) : (scheduled)

Yet unseen words: a meeting for Wednesday ?

Figure 5.8 - After AUX-INVERSION has been executed.

From this example, we see that the ability to delete constituents from other than the first place in the buffer, in conjunction with the fact that deletion always compacts the contents of the buffer, allows a key generalization to be captured. Most interestingly, the removal of the subject NP in this case was *not* specifically stipulated by the grammar rule, which merely specified that the second NP in the buffer was to be attached to the dominating S. The deletion followed instead from ***general principles of the***

grammar interpreter's operation. This latter point is crucial; given a simple statement of the structure of yes-no questions in English, the proper behavior follows from much more general principles. We will see more examples of generalizations captured via general properties of the parsing mechanism in the examples that follow.

Parsing an Imperative

In the last section it was demonstrated that the analysis process for declaratives and yes-no questions differed in the execution of exactly two rules. In this section I will demonstrate that the analysis process for declaratives and imperatives differs in exactly one rule of grammar. While declaratives trigger the rule MAJOR-DECL-S in the packet SS-START, imperatives will trigger the rule IMPERATIVE. After one or the other of these two rules has run, the remaining analysis process is essentially identical for both imperatives and declaratives.

As shown in Figure 5.9 below, the rule IMPERATIVE differs from MAJOR-DECL-S in the following ways: 1) MAJOR-DECL-S triggers on an NP followed by a verb; IMPERATIVE triggers on a tenseless verb. 2) The two rules assign different type features to the clause. 3) The rule IMPERATIVE includes the PIDGIN code

Insert the word 'you' into the buffer.

This code is the crucial difference between these two rules; we must examine exactly what its execution accomplishes.

{RULE MAJOR-DECL-S IN SS-START [=np] [=verb] -->` Label c s, decl, major. .... Deactivate ss-start. Activate parse-subj.}	{RULE IMPERATIVE IN SS-START [=tnsless] -->` Label c s, imper, major. Insert the word 'you' into the buffer. Deactivate ss-start. Activate parse-subj.}
DECLARATIVES (a)	IMPERATIVES (b)

Figure 5.9 - Differences between declarative and imperative rules.

In general, the PIDGIN construct

Insert the word '<word>' into the buffer (before <position>)

inserts <word> into the buffer before <position>, which may be either 1st, 2nd, or 3rd. If the position is not specified, as in the rule IMPERATIVE, the default is before 1st. The insert is done using the virtual buffer machine command insert (<word>, <position>). Thus, the previous contents of the buffer cell corresponding to <position>, as well as all constituents in the buffer to the right of that cell, are shifted one cell further to the right before <word> is put into the buffer. Note that this command allows only specific lexical items to be inserted into the buffer. Except for the initial entry of words into the buffer from the input list, all other constituent structure built up by the parser can only enter the buffer by being dropped from the stack into the buffer, where it is inserted, by definition, into the first buffer cell.

Once a word is inserted into the buffer by this command, it is indistinguishable from any other lexical item that has been inserted into the buffer from the input list. In the case of the word "you", this means that

the processes that build NPs in a manner transparent to clause level processes will immediately construct an NP node which dominates the pronoun "you". Thus, from the point of view of the processes that we are considering in this chapter, the net effect of this code will be to insert an NP dominating the pronoun "you" into the 1st position of the buffer.

Let us now trace through the initial steps of the following sentence, shown previously in Figure 5.3.c above, to see exactly what the full effect of the rule IMPERATIVE is:

5.3.c Schedule the meeting for Wednesday.

The parser's initial state is shown in Figure 5.10.a below; the packet SS-START, which contains IMPERATIVE, is active. The rule IMPERATIVE matches, and its action is executed, leaving the parser in the state shown in Figure 5.10.b. The node S20 has been labelled imperative, major, and a new NP node, NP46, which dominates the pronoun "you" is now in the first position in the buffer. The rule IMPERATIVE has deactivated the packet SS-START and activated the packet PARSE-SUBJ. *Note that the parser is in a state which is essentially equivalent to the state that it would be in, given a declarative clause as input*, following the execution of the rule MAJOR-DECL-S.

The Active Node Stack (0. deep)

C: S20 (S) / (CPOOL SS-START)

The Buffer

1 : WORD144 (*SCHEDULE COMP-OBJ VERB PRES TNSLESS ...) : (Schedule)

Yet unseen words: a meeting for Wednesday .

(a) - Before IMPERATIVE has been executed.

The Active Node Stack (0. deep)

C: S20 (S IMPER MAJOR) / (PARSE-SUBJ CPOOL)

The Buffer

1 : NP46 (NS NPL N2P NP PRON-NP) : (YOU)
2 : WORD144 (*SCHEDULE COMP-OBJ VERB PRES TNSLESS ...) : (Schedule)

Yet unseen words: a meeting for Wednesday .

(b) - After IMPERATIVE has been executed.

Figure 5.10 - The action of the rule IMPERATIVE.

Just as if the original sentence were a declarative, the rule UNMARKED-ORDER will trigger next, attaching NP46 to S20 as its subject. The rest of the analysis process will be exactly parallel to the process that would follow from a declarative sentence. Thus, the only difference in the analyses of imperatives and declaratives will be in the execution of one rule: declaratives will trigger MAJOR-DECL-S, while imperatives will trigger IMPERATIVE. (The reader might wonder how it is that the tenseless verb "schedule" will properly interact with the auxiliary parsing rules. As pointed out in Chapter 3, in a clause with a simple verb, i.e. a verb cluster consisting of only a main verb with no auxiliaries, the auxiliary node will

not dominate any lexical items, but will carry the tense and person/number codes of the verb. In the case of a tenseless verb, the auxiliary will thus be marked as tenseless, with no person/number codes.)

The crucial point of this example is that by merely inserting the word "you" into an appropriate position in the buffer, the rule IMPERATIVE has created a subject for the imperative clause, reestablishing the subject-verb order which typically initiates "unmarked" clauses like major declarative clauses and all subordinate clauses. As in the case of yes-no questions, it has thus been demonstrated that an important grammatical generalization can be easily captured within the framework of the grammar interpreter. And once again, it turns out to be the case that the relevant grammar rule need only briefly specify part of what the grammar interpreter should do; the bulk of what needs to be done (in this case the creation of an NP that dominates the pronoun) follows automatically from mechanisms not connected with this grammar rule, mechanisms that are independently motivated.

(The observant reader might wonder how the parser handles sentences like 5.11.a and b below, since "have" is potentially both tenseless and an auxiliary, triggering both YES-NO-Q and IMPERATIVE. The answer is the central focus of Chapter 9. In short, the problem is solved by introducing a "diagnostic" rule that decides (if it can) between potential conflicts by examining the number features of the following NP and the morphology of the verb in the 3rd buffer cell.

(a) Have the boys arrive today.
(b) Have the boys arrived today?

Figure 5.11 - Two sentences that trigger both YES-NO-Q and IMPERATIVE.

Parsing Passives - A Simple Example

Earlier in this chapter, the notion of a *trace* was introduced as a mechanism to encode the underlying predicate/argument structure of an utterance in a surface structure representation. As shown earlier in Figure 5.2, traces can be used to encode where in an utterance a wh-phrase can be said to belong in terms of predicate/argument relations, which NP in a higher clause "controls" the subject of an embedded S, as well as to indicate the fact that the surface subject of a passive clause is really its "underlying" object. In this section, we will investigate the question of how the grammar deals with simple passive constructions, and then in the following section investigate the phenomenon known in the transformational literature as "raising" [Postal 74], sentences in which what seems to be the subject of an embedded complement is passivized into the subject position of the higher clause. We will also see how the grammar handles verbs like "seem", which seem to allow the subject of an embedded clause to shift into subject position in the higher clause.

Let us begin by examining a simple case of passivization, as embodied in the following sentence, which was given above as 5.3.d:

5.3.d The meeting was scheduled for Wednesday.

We will begin our example with the parser in the state shown in Figure

5.12 below. The analysis process for the sentence prior to this point is essentially parallel to the analysis of any simple declarative with one exception: the rule PASSIVE-AUX in packet BUILD-AUX has decoded the passive morphology in the auxiliary and given the auxiliary the feature *passive* (although this feature is not visible in Figure 5.12). At the point we begin our example, the rule MVB has just been executed, and the packet SUBJ-VERB is active.

The Active Node Stack (1. deep)

S21 (S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (The meeting)
AUX : (has been)
VP : ↓
C: VP17 (VP) / (SUBJ-VERB CPOOL)
VERB : (scheduled)

The Buffer

1 : PP14 (PP) : (for Wednesday)
2 : WORD162 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)

Figure 5.12 - Analysis of a passive after the verb has been attached.

Besides the rule SUBJ-VERB, the packet SUBJ-VERB also contains the rule PASSIVE, shown in Figure 5.13 below. As I will show, this rule by itself is sufficient to account for many of the phenomena that accompany clause-level passivization (including, as I will show in the next section, the phenomenon of raising). The pattern of this rule is fulfilled if the auxiliary of the S node dominating the current active node (which will always be a VP node if packet SUBJ-VERB is active) has the feature *passive*, and the S node has not yet been labelled *np-preposed*. (The feature *np-preposed*,

which means that the subject NP has been conceptually "preposed" from an "underlying" position immediately after the verb, is used to label not only passives, but also clauses with main verbs like "seem", as in "John seems to be happy".) The action of the rule PASSIVE creates a trace, sets the binding of the trace to the subject of the dominating s node, and then drops the new trace into the buffer.

```
{RULE PASSIVE IN SUBJ-VERB
[** c; the aux of the s above * is passive;
        the s above * is not np-preposed] -->
Label the s above c np-preposed.
Create a new np node labelled trace.
Set the binding of c to the np of the s above c.
Drop c.}
```

Figure 5.13 - PASSIVE captures np-preposing in 4 lines of code.

Returning to our example, the packet SUBJ-VERB, containing both the rules SUBJ-VERB and PASSIVE, is now active. The patterns of both rules are fulfilled by the current parse tree: PASSIVE matches since the auxiliary of S21 is indeed labelled passive, and S21 itself is not labelled np-preposed; SUBJ-VERB will always match since it is a "default" rule with a dummy pattern which is always true. (Such a default rule always has a low priority so it matches if and only if no rules of higher priority match.) Since both rules can match, and PASSIVE has higher priority than SUBJ-VERB, the action of PASSIVE is run next.

PASSIVE first adds the feature *np-preposed* to S21; not only does this feature signal that the sentence has a preposed subject, but it also blocks the rule PASSIVE from applying repeatedly, since the pattern of the rule will

only match if this feature is absent. It next creates a trace, binds it to the subject of the sentence, in this case the NP "The meeting", and then drops the trace into the buffer. Since the verb of the clause has just been attached by the previous rule, this has the effect of inserting the trace into the surface string immediately after the verb. One will note that this is not at all equivalent to immediately attaching the trace as the object of the verb, as will become clear in the following section. The state of the parser after this rule has been executed is shown in Figure 5.14 below. S21 is now labelled with the feature np-preposed, and there is a trace, NP53, in the first buffer position. NP53, as a trace, has no daughters, but is bound to the subject of S21.

The Active Node Stack (1. deep)

S21 (NP-PREPOSED S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (The meeting)
AUX : (has been)
VP : ↓
C: VP17 (VP) / (SUBJ-VERB CPOOL)
VERB : (scheduled)

The Buffer

1 : NP53 (NP TRACE) : bound to: (The meeting)
2 : PP14 (PP) : (for Wednesday)
3 : WORD162 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)

Figure 5.14 - After PASSIVE has been executed.

After PASSIVE has been executed, the packet SUBJ-VERB is still active, but now the rule PASSIVE is blocked from matching by the presence of the feature *np-preposed*, so the rule SUBJ-VERB matches and is run next. After it has been executed the parser is in virtually the same state as in

Figure 5.14 above, with the exception that a different set of packets is now associated with VP17; SUBJ-VERB has deactivated the packet SUBJ-VERB and, in this case, activated the packets SS-VP and INF-COMP.

Sitting in packet SS-VP is the rule OBJECTS, which is now triggered by the trace NP53 in the first buffer cell. This rule attaches the trace to the VP, VP17, making the trace the object of the verb "scheduled". This flags the fact that the clause's apparent subject is really the underlying object of the verb, just as trace was used to mark the passive in Figure 5.2.2 above. In this sense, the passive has been "undone". The state of the parser at this point is shown in Figure 5.15 below.

The Active Node Stack (1. deep)

S21 (NP-PREPOSED S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (The meeting)
AUX : (has been)
VP : ↓

C: VP17 (VP) / (SS-VP INF-COMP CPOOL)
VERB : (scheduled)
NP : bound to: (The meeting)

The Buffer

1 : PP14 (PP) : (for Wednesday)
2 : WORD162 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)

Figure 5.15 - After OBJECTS has attached the trace to VP17.

Note that the rule OBJECTS is totally indifferent both to the fact that NP53 was not a regular NP, but rather a trace, and the fact that NP53 did not originate in the input string, but was placed into the buffer by grammatical processes. In the simple declarative example discussed previously

in Chapter 4, this rule was triggered in an identical manner by the NP "the meeting" which was in this case the actual surface object of the verb. Thus, whether or not this rule is executed is absolutely unaffected by differences between an active sentence and its passive form; the analysis process for either is identical as of this point in the parsing process. Furthermore, after this rule has been executed, the state of the parser as captured in Figure 5.15 above is essentially identical to its state after the object of the verb was attached in the earlier simple declarative example. Thus, the analysis process will be exactly parallel in both cases after the object has been attached. In point of fact, the analysis process for a passive sentence, outside of picking up the passive morphology in the auxiliary phrase, differs from the processing of an active clause only in the execution of the rule PASSIVE; the remainder of the analysis process is exactly parallel. (I remind the reader that the analysis of passive that is assumed above does *not* assume a process of "agent deletion", "subject postposing" or the like. As stated earlier in this chapter, I follow Chomsky is assuming that passive sentences have *no* underlying subject.)

Passives and Embedded Complements

In the previous section, we investigated how simple passives were handled by the parser. In this section we will investigate how the parser handles much more complicated passives, such as the sentence presented in an earlier section as 5.2.4a, repeated below for convenience. The desired analysis of this sentence, as shown in 5.2.4b, analyzes "to be happy" as an embedded S whose subject is a trace bound to the subject of the higher clause. Such an analysis views (4b) as deriving conceptually from an

underlying form similar to (4c), where the subject of the underlying clause has been passivized into subject position in the upper clause. In this section I will show how the parser builds such an analysis.

(4a) John was believed to be happy.
(4b) John was believed [$_S$ *t* to be happy].
|_____|
(4c) ∇ believed *John* to be happy.

Figure 5.2 (repeated) - The intended analysis for a complex passive

---

The reader may have wondered why the PASSIVE rule introduced in the previous section was formulated to drop the trace it creates into the buffer rather than immediately attaching the new trace to the VP node. As I will show in this section, such a formulation of PASSIVE also correctly analyzes passives like (4a) in Figure 5.2 above which involve "raising" with no additional complexity added to the grammar. Thus, a simple statement of passivization will have been shown to correctly capture a generalization about a range of grammatical phenomena. To fully show the range of the generalization, the example which we will investigate in this section, sentence (1) in Figure 5.16 below, is yet a level more complex than (4a) above; its analysis is shown schematically in 5.16.2. In this example there are two traces: the first, the subject of the embedded clause, is bound to the subject of the major clause, the second, the object of the embedded S, is bound to the first trace, and is thus ultimately bound to the subject of the higher S as well. Thus the underlying position of the NP "the meeting" can be viewed as being the object position of the embedded S, parallel to the overt surface structure of (3) in Figure 5.16. This analysis presupposes that a clause with main verb "seems" behaves as if the verb were passive; I will

show in the following example how this generalization can be easily captured.

(1) The meeting seems to have been scheduled for Wednesday.

(2) The meeting seems [$_S$ *t* to have been scheduled t for Wednesday].

(3) It seems [$_S$ that ∇ scheduled *the meeting* for Wednesday].

Figure 5.16 - The example which will be discussed in this section.

Before the analysis of (1) above can be discussed, it is necessary for the reader to understand how a simple embedded complement is initiated. Let us briefly look at how the complement in sentence 5.17 below is initiated.

5.17 We wanted John to schedule a meeting for Wednesday.

We will start this brief digression with the parser in the state depicted below in Figure 5.18. The rule SUBJ-VERB has just been executed and has activated the packets SS-VP and INF-COMP.

The Active Node Stack (1. deep)

S3 (S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (We)
AUX : NIL
VP : ↓
C: VP3 (VP) / (SS-VP INF-COMP CPOOL)
VERB : (wanted)

The Buffer

1 : NP6 (NP NAME NS N3P) : (John)
2 : WORD14 (*TO PREP AUXVERB) : (to)
3 : WORD15 (*SCHEDULE COMP-OBJ VERB PRES TNSLESS ...) : (schedule)

Yet unseen words: a meeting for Wednesday .

Figure 5.18 - Parsing a simple complement - After SUBJ-VERB has run.

One of the rules in the packet INF-COMP is the rule INF-START1, shown below in Figure 5.19. This rule will trigger if there is an NP in the buffer, followed by the word "to", followed by a tenseless verb. It will create a new S node, labelled as an infinitive, and attach the NP to the S node. This rule must be assigned a priority of 5 because this packet is often active at the same time that the packet SS-VP, which contains the rule OBJECTS, is active. OBJECTS is triggered by a single NP, so that the pattern of OBJECTS will match whenever the pattern of INF-S-START1 will. By giving INF-S-START1 a higher priority than OBJECTS, INF-S-START1, and not OBJECTS, will be executed by the grammar interpreter if both rules match, which is the desired behavior. (Note that it would be easy in this case for an algorithm to detect that the pattern of INF-S-START1 is a further specification of the pattern of OBJECTS, and thus must be assigned a higher priority, given the heuristic that a rule whose pattern is a further specification of another should receive higher priority than that other rule.)

(The reader might find it useful at this point to glance ahead to Figure 5.30 at the end of the penultimate section of this chapter. This figure shows the overall organization of the set of grammar rules presented through the end of the chapter. This figure clearly shows, for example, that the rules OBJECTS and INF-S-START1 are in packets which are closely related, although it doesn't indicate that they may be active at the same time.)

```
{RULE INF-S-START1 PRIORITY: 5. IN INF-COMP
[=np] [="to,auxverb] [=tnsless] -->
Label a new s node sec, inf-s.
Attach 1st to c as np.
Activate cpool, parse-aux.}
```

Figure 5.19 - One rule which starts infinitives.

With the parser in the state shown in Figure 5.18 above, the packet INF-COMP, containing the rule INF-S-START1, is active, and this rule now matches and is executed by the grammar interpreter. After this rule is run, the parser is in the state shown in Figure 5.20 below. Now the subordinate clause has been initiated and will be completed by grammatical processes similar to those that have been discussed in previous examples. After the embedded clause is completed, it will be dropped into the buffer where it will trigger another rule in the packet INF-COMP which will attach it to an NP attached to the dominating VP.

The Active Node Stack (2. deep)

S3 (S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (We)
AUX : NIL
VP : ↓
VP3 (VP) / (SS-VP INF-COMP CPOOL)
VERB : (wanted)
C: S4 (SEC INF-S S) / (CPOOL PARSE-AUX)
NP : (John)

The Buffer

1 : WORD14 (*TO PREP AUXVERB) : (to)
2 : WORD15 (*SCHEDULE COMP-OBJ VERB PRES TNSLESS ...) : (schedule)

Yet unseen words: a meeting for Wednesday .

Figure 5.20 - After INF-S-START1 has been executed.

With this preliminary out of the way, we can now begin our investigation of the analysis process for (a) in Figure 5.16 above.

We begin our example, once again, right after the rule MVB has been executed, attaching "seems" to VP20, the current active node, as shown in Figure 5.21 below.

The Active Node Stack (1. deep)

S22 (S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (The meeting)
AUX : NIL
VP : ↓
C: VP20 (VP) / (SUBJ-VERB CPOOL)
VERB : (seems)

The Buffer

1 : WORD166 (*TO PREP AUXVERB) : (to)
2 : WORD167 (*HAVE VERB TNSLESS AUXVERB PRES ...) : (have)

Yet unseen words: been scheduled for Wednesday .

Figure 5.21 - The example begins after MVB has been executed.

Following Chomsky (as referenced in [Fiengo 74]), I will analyze "seem" as coming from an underlying source like (1) in Figure 5.22 below, where the symbol ∇ indicates that the source has no underlying subject. If the complement is a that-clause, the subject position will be filled by the word "it" as in (2); if the complement is an infinitive, the subject of the complement will be preposed into the subject position of the higher clause, as in (3), leaving behind a trace. For the infinitive case, as in our example, this means that the parser must create a trace bound to the subject of the upper clause and place it into position in the lower clause.

(1) ∇ seems [$_S$ John be happy].
(2) It seems [$_S$ that John is happy].
(3) John seems [$_S$ *t* to be happy].

Figure 5.22 - The underlying form of clauses with verb "seem".

I will show that the rule SEEMS, shown below in Figure 5.23, is the only additional rule that is needed to handle this phenomenon. This rule simply says that if the contents of the buffer clearly indicate an infinitive, and if the main verb is a verb like "seems" which can be "subject-less", and if the NP preposing hasn't been undone yet (adding the feature *np-preposed* as a flag), then do the same thing that the PASSIVE rule does. The essence of this rule is that there is one simple action to handle the general phenomenon of NP preposing, but that two different situations flag the fact that the subject NP has been preposed. Thus our current example will demonstrate two points: 1) that the action of the PASSIVE rule is sufficient to handle several cases of NP preposing; and 2) that this rule correctly handles preposing even if the NP that has been preposed originated in a lower clause.

```
{RULE SEEMS IN SUBJ-VERB
[=*to] [=tnsless]
[** c; the verb of * is no-subj;
        the s above * is not np-preposed] -->
%This simple form won't handle "John seems happy."%
Run passive next.}
```

Figure 5.23

With the parser in the state indicated in Figure 5.21 above, the packet SUBJ-VERB, which contains SEEMS, is active, and this rule's pattern is fulfilled, so the rule is executed. SEEMS does nothing but signal the grammar interpreter to execute PASSIVE, so PASSIVE is now executed. This rule, as stated above, creates a trace, binds it to the subject of the current clause, and drops the trace into the first cell in the buffer. The resulting state is shown in Figure 5.24 below.

The Active Node Stack (1. deep)

S22 (NP-PREPOSED S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (The meeting)
AUX : NIL
VP : ↓
C: VP20 (VP) / (SUBJ-VERB CPOOL)
VERB : (seems)

The Buffer

1 : NP55 (NP TRACE) : bound to: (The meeting)
2 : WORD166 (*TO PREP AUXVERB) : (to)
3 : WORD167 (*HAVE VERB TNSLESS AUXVERB PRES ...) : (have)

Yet unseen words: been scheduled for Wednesday .

Figure 5.24 - After SEEMS has caused PASSIVE to be executed.

The rule SUBJ-VERB is now triggered, and deactivates the packet SUBJ-VERB and activates the packets SS-VP (which contains the rule OBJECTS) and INF-COMP (which contains INF-S-START1), among others. But now the patterns of OBJECTS and INF-S-START1 will both match, and INF-S-START1, will be executed by the interpreter since it has the higher priority. (Note once again that a trace is a perfectly normal NP from the point view of a rule's pattern.) Thus this rule now creates a new S node labeled infinitive and attaches the trace NP55 to the new infinitive as its subject. The resulting state is shown in Figure 5.25 below.

The Active Node Stack (2. deep)

S22 (NP-PREPOSED S DECL MAJOR) / (SS-FINAL CPOOL)
NP : (The meeting)
AUX : NIL
VP : ↓
VP20 (VP) / (SS-VP THAT-COMP INF-COMP CPOOL)
VERB : (seems)
C: S23 (SEC INF-S S) / (CPOOL PARSE-AUX)
NP : bound to: (The meeting)

The Buffer

1 : WORD166 (*TO PREP AUXVERB) : (to)
2 : WORD167 (*HAVE VERB TNSLESS AUXVERB PRES ...) : (have)

Yet unseen words: been scheduled for Wednesday .

Figure 5.25 - After INF-S-START1 has been executed.

We are now well on our way to the desired analysis. An embedded infinitive has been initiated, and a trace bound to the subject of the dominating S has been attached as its subject. Furthermore, the parser is now in a state analogous to the state depicted in Figure 5.20 after the parser has attached the subject of a simple infinitive complement. The key difference, of course, is that the NP that serves as the subject of the infinitive clause is in this case a trace.

The parser will now proceed, exactly as in earlier examples, to build the auxiliary, attach it, and attach the verb "scheduled" to a new VP node. After the rules that accomplish this have been executed, the parser is left in the state depicted in Figure 5.26 below. (Note that for the sake of brevity, only the 3 bottommost nodes in the active node stack will be shown in this and all successive diagrams.) The infinitive auxiliary has been parsed and

attached, and VP21 is now the current active node, with the verb "scheduled" as main verb of the clause. It should be noted that the auxiliary has been assigned the feature *passive* by the auxiliary parsing rules, although this is not shown in the figure below.

The Active Node Stack (3. deep)

.....
VP20 (VP) / (SS-VP THAT-COMP INF-COMP CPOOL)
VERB : (seems)
S23 (SEC INF-S S) / (EMB-S-FINAL CPOOL)
NP : bound to: (The meeting)
AUX : (to have been)
VP : ↓
C: VP21 (VP) / (SUBJ-VERB CPOOL)
VERB : (scheduled)

The Buffer

1 : PP15 (PP) : (for Wednesday)
2 : WORD174 (". FINALPUNC PUNC) : (.)

Yet unseen words:

Figure 5.26 - After the auxiliary and main verb have been parsed.

The packet SUBJ-VERB, containing the rules PASSIVE and SUBJ-VERB, is now active. Once again PASSIVE's pattern matches and this rule is executed, creating a trace, binding it to the subject of the clause, (which is in this case itself a trace), and dropping the new trace into the buffer. And again, it labels the dominating S node, in this case S23, with the feature *np-preposed*, blocking the PASSIVE rule from reapplying. This is shown in Figure 5.27 below. Note that in this figure, as in earlier figures, the *binding*" of each trace is indicated.

The Active Node Stack (3. deep)

.....
VP20 (VP) / (SS-VP THAT-COMP INF-COMP CPOOL)
VERB : (seems)
S23 (NP-PREPOSED SEC INF-S S) / (EMB-S-FINAL CPOOL)
NP : bound to: (The meeting)
AUX : (to have been)
VP : ↓
C: VP21 (VP) / (SUBJ-VERB CPOOL)
VERB : (scheduled)

The Buffer

1 : NP57 (NP TRACE) : bound to: (The meeting)
2 : PP15 (PP) : (for Wednesday)
3 : WORD174 (*. FINALPUNC PUNC) : (.)

Yet unseen words: (none)

Figure 5.27 - After PASSIVE has run on the lower clause.

Now the rule SUBJ-VERB will be executed, activating in this case the packets INF-COMP, containing INF-S-START1, and EMB-S-VP, which contains the rule OBJ-IN-EMBEDDED-S. The rule OBJ-IN-EMBEDDED-S is similar to the rule OBJECTS except that it does a semantic test before attaching its triggering NP to the VP, to assure that the NP is semantically acceptable with this verb. If the NP is not semantically acceptable with this verb, it advises the grammar interpreter to finish the VP, leaving the NP in the buffer to be attached later to some higher S. This rule is shown in Figure 5.28.a below. (Since INF-S-START1 has a higher priority than OBJ-IN-EMBEDDED-S, the grammar interpreter attempts to match the former rule before the latter, but INF-S-START1 fails to match, while OBJ-IN-EMBEDDED-S matches successfully.) OBJ-IN-EMBEDDED-S is now run, the semantic test is successful, we will assume, and the trace is attached to VP21. The

resulting state is shown in 5.28.b below.

```
{RULE OBJ-IN-EMBEDDED-S IN EMBEDDED-S-VP
[=np] -->
If 1st fits an obj slot of the cf of c
        then attach 1st to c as np
      else run embedded-vp-done next.}
```

(a) - The embedded clause version of the rule OBJECTS.

The Active Node Stack (3. deep)

```
                .....
        VP20 (VP) / (SS-VP THAT-COMP INF-COMP CPOOL)
                VERB : (seems)
        S23 (NP-PREPOSED SEC INF-S S) / (EMB-S-FINAL CPOOL)
                NP : bound to: (The meeting)
                AUX : (to have been)
                VP : ↓
C:      VP21 (VP) / (EMBEDDED-S-VP INF-COMP CPOOL)
                VERB : (scheduled)
                NP : bound to: (The meeting)
```

The Buffer

```
1 :     PP15 (PP) : (for Wednesday)
2 :     WORD174 (*. FINALPUNC PUNC) : (.)
```

Yet unseen words: (none)

(b) - After OBJ-IN-EMBEDDED-S has been executed.

Figure 5.28

The remainder of the parsing process proceeds in a fashion similar to the simple infinitive example discussed above. Once the infinitive is completed, it is dropped into the buffer, where it triggers a grammar rule which attachs it to an NP node attached to the VP of the major clause. The tree structure which results after the parse is complete is shown in Figure 5.29.a below. (For the sake of brevity, most features have been deleted from this tree and the following case frame structure.) A trace is indicated in

this tree by giving the terminal string of its ultimate binding in parentheses. To give the reader an idea of what the underlying structure of this sentence is taken to be, an condensed version of the output of the case-frame component that processes the output of this parser is shown in Figure 5.29.b. This case frame indicates that the toplevel predicate SEEM has one argument in the NEUTRAL case which is itself a proposition. The predicate of this embedded proposition is SCHEDULE and it has two arguments, a NEUTRAL which is "The meeting" and a TIME which is "Wednesday" with the marker "for". The predicate SCHEDULE also takes an AGENT, but the example does not specify who the AGENT is, so no AGENT can be given in the case frame analysis. The case frame component derives this case representation, in essence, by ignoring the surface subjects of all np-preposed clauses, and replacing all traces with their ultimate bindings.

```
(NP-PREPOSED S DECL MAJOR)
    NP: (MODIBLE NP DEF DET NP)
          DET: The
          NBAR: (NS NBAR)
                NOUN: meeting
    AUX: (PRES V3S AUX)
    VP: (VP)
          VERB: seems
          NP: (NP COMP)
                S: (NP-PREPOSED SEC INF-S S)
                      NP: (NP TRACE)
                                (bound* to: The meeting)
                      AUX: (PASSIVE PERF INF AUX)
                            TO: to
                            PERF: have
                            PASSIVE: been
                     VP: (VP)
                            VERB: scheduled
                            NP: (NP TRACE)
                                      (bound* to: The meeting)
                            PP: (PP)
                                  PREP: for
                                  NP: (NP TIME DOW)
                                        NOUN: Wednesday
    FINALPUNC: .
```

(a) - The tree structure resulting from parsing 5.3.d.

```
    PRED :   SEEM
    NEUT:     MARKER: OBJ
             (NP COMP)
          PRED :   SCHEDULE
          NEUT: MARKER: OBJ
                    (NP TRACE)
                      (bound* to:The meeting)
          TIME: MARKER: FOR
                   (NP TIME DOW)
                         (*WEDNESDAY NGSTART NOUN NS N3P DAY-OF-WEEK TIME)
                             Wednesday
```

(b) - The Case Frame resulting from a case analysis of (a) above.

Figure 5.29

The crucial point to be drawn from this example is that the simple formulation of the PASSIVE rule presented above, interacting with other simply formulated grammatical rules for parsing objects and initiating embedded infinitives, allows a trace to be attached either as the object of a verb or as the subject of an embedded infinitive, whichever is the appropriate analysis for a given grammatical situation. The PASSIVE rule is formulated in such a way that it drops the trace it creates into the buffer, rather than attaching the trace somewhere in particular in the tree. Because of this, later rules, already formulated to trigger on an NP in the buffer, will analyze sentences with NP-preposing exactly the same as those without a preposed subject. The PASSIVE rule, formulated as it is, elegantly captures the notion that the "underlying" location of a preposed NP is, in fact, immediately after the verb in the underlying *terminal string*, regardless of its position in the underlying *tree*. Once again, we see that the availability of the buffer mechanism is crucial to capturing this generalization; such a generalization can only be stated by a parser with a mechanism much like the buffer used here.

To summarize the grammatical processes discussed in this chapter, Figure 5.30 presents the full set of grammar rules presented in this and the previous chapter. The notation is the same as that of Figure 4.31, with the following extensions: Rules in simple lower case were first presented in Chapter 4; rules in italics were newly presented in this chapter.

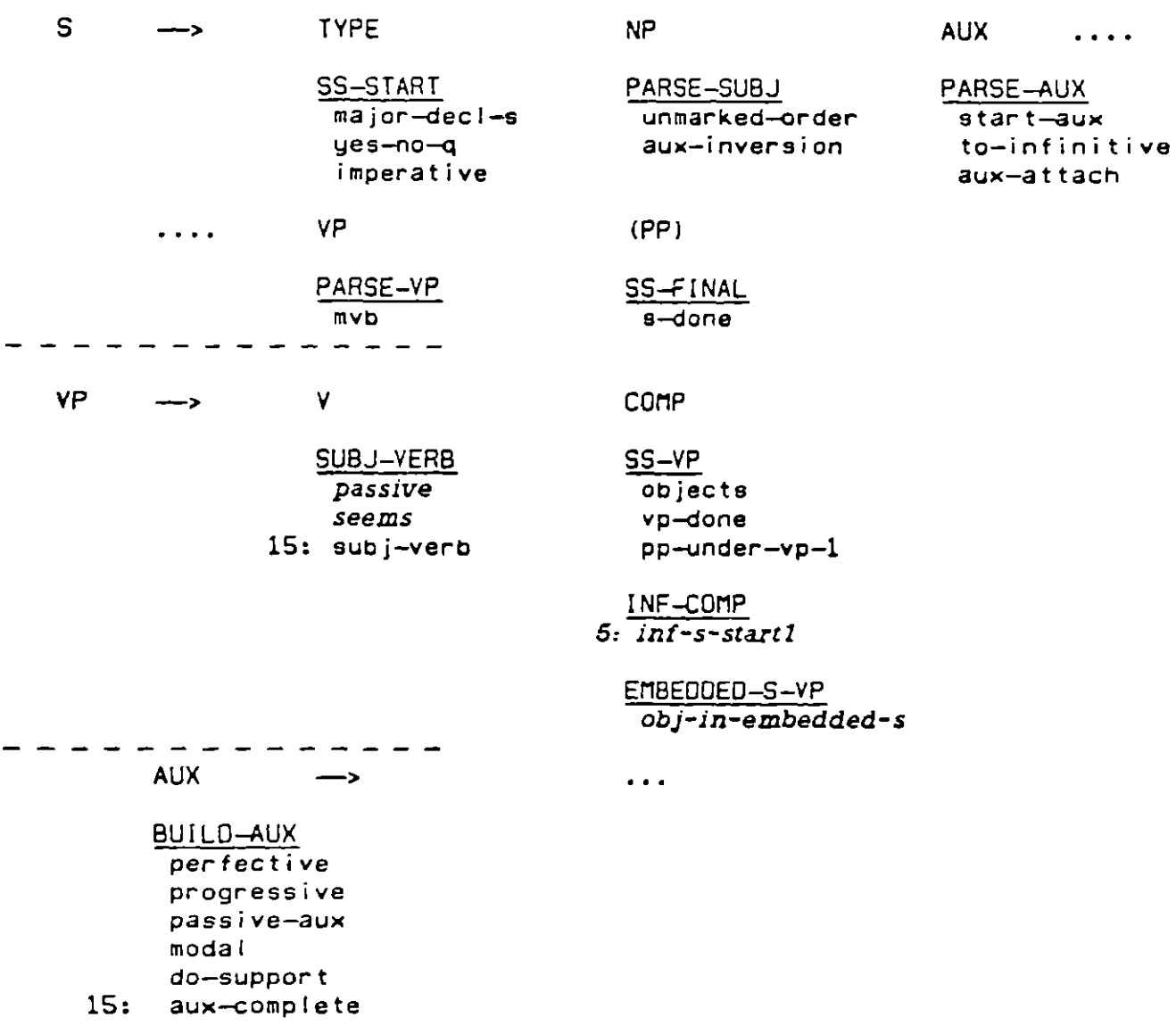


Figure 5.30 - The set of grammar rules as extended in this chapter.

On the Discovery of These Generalizations

As a historical note, I should mention that I was surprised to discover that grammar rules could be written so as to elegantly capture these generalizations. I must admit that this was not initially a foreseen result of this research. An earlier grammar of English, written in terms of a prior

implementation of a similar grammar interpreter, utilized rules that captured none of these generalizations, although (or perhaps because) these earlier rules were individually much more powerful computationally than the rules of the current grammar. It was only after severely restricting the grammar language that the current style of writing rules was discovered. After discarding the use of registers to store tree fragments, restricting the number of constituents that a single rule could access, and totally eliminating the use of variables in the grammar rules, it became necessary to reformulate much of the grammar to fit the tighter constraints of the current implementation. These constraints forced me to find methods of manipulating grammatical structures that soon led to a much simpler formulation of many grammatical rules by taking greater advantage of the buffer mechanism.

I restricted the richer computational structure of the earlier implementation because of a desire to approach the minimal computational mechanism necessary to parse English. Perhaps the seeming paradox that a more restricted mechanism led to a more elegant grammar should have come as no great surprise.

CHAPTER 6

THE GRAMMAR INTERPRETER AND CHOMSKY'S CONSTRAINTS

Introduction

In a series of papers over the last several years, Noam Chomsky has argued for several specific properties of language which he claims are universal to all human languages [Chomsky 73, 75, 76]. These properties, which form one of the cornerstones of his current linguistic theory, are embodied in a set of constraints on language, a set of restrictions on the operation of rules of grammar.

In this chapter and the next, I will show that important sub-cases of two of these constraints fall out naturally from the structure of the grammar interpreter. I will then show that many of the properties of the interpreter which are crucial to capturing these universals are motivated by the determinism hypothesis. Thus, I will demonstrate that significant sub-cases of Chomsky's universals follow naturally from the determinism hypothesis and in this sense are explained by it. Furthermore, if Chomsky's claims of universality are true, this demonstration shows that the mechanisms of the grammar interpreter capture a number of generalizations which are purportedly universal to all human languages. Such a demonstration provides further evidence in favor of the determinism hypothesis.

(It should be noted that these constraints are far from universally accepted. They are currently the source of much controversy; for various critiques of Chomsky's position see [Postal 74; Bresnan 76]. I will not argue for these constraints in this chapter but will accept them as more or less accurate, with several reservations which will be stated in the discussion that follows.)

As a preliminary to the following discussion, a few technical terms must be defined. The following are the standard linguistic definitions of *dominates* and *immediately dominates*: "If we can trace a path up the tree from a given node X to a given node Y, then we say *X is dominated by Y*, or equivalently, *Y dominates X*." If Y dominates X, and no other nodes intervene (i.e. X is a daughter of Y), then *Y immediately (or directly) dominates X*. [Akmajian & Heny 75]. One non-standard definition will prove useful: I will say that if Y dominates X, and Y is a cyclic node, i.e. an S or NP node, and there is no other cyclic node Z such that Y dominates Z and Z dominates X (i.e. there is no intervening cyclic node Z between Y and X) then *Y Dominates X*.

An Outline of Chomsky's Theory

Noam Chomsky's current theory of generative grammar differs in important respects from both the theory of generative grammar presented in *Aspects of the Theory of Syntax* [Chomsky 65], the so-called "standard theory" (ST), or the extension of that theory proposed in the papers collected in [Chomsky 72], otherwise known as the "extended standard theory" (EST). Because this theory is not yet widely known, the following section provides

a short summary of the aspects of this theory relevant to the focus of this chapter.

The standard theory proposes (in part) that there is a level of linguistic structure called "*deep structure*" which is mapped by an ordered set of grammatical transformations, the *transformational component*, onto a level of linguistic structure called "*surface structure*". The transformational component consists of such transformations as Passive, Dative Shift, Question Formation, Auxiliary Inversion, and the like. Both the level of deep structure and surface structure (as well as intermediate structures produced by the transformational component) have the form of *phrase markers*, which, loosely speaking, correspond to the tree structures of formal language theory, and each transformation maps the phrase marker which is its input onto another phrase marker. The level of surface structure is distinguished in that it is both the output of the transformational component, and the level of representation which is mapped by the *phonological component* onto a representation which can be mapped onto the acoustic waveform. The level of deep structure is distinguished in that it is both the level of structure which is the input to the transformational component and the level of linguistic structure which is appropriate for semantic interpretation.

The extended standard theory accepts the bulk of the standard theory as stated above. One of the crucial differences between the ST and EST is that the EST rejects the notion that the level of deep structure, by itself, is an appropriate level of representation for semantic interpretation. The EST takes into account the fact that some important semantic properties

of an utterance, such as scope of quantification and restrictions on pronominal reference, can only be deduced by examining the structure of the surface structure tree, including the left-to-right order of its terminals, as well as its deep structure. The EST proposes that the semantic interpretation of an utterance must derive from *both* its surface structure and its deep structure. (For a good tutorial comparison of these two theories, as well as the theories of such linguists as Lakoff and Fillmore, see [Maclay 71]. The volume in which this paper appears ([Steinberg & Jakobovitz 71]) also reprints the paper in which Chomsky first proposes the modification of the standard theory discussed here.)

Chomsky's current theory, as first presented in [Chomsky 73, 76] and currently being modified in such papers as [Chomsky to appear; Chomsky & Lasnik 77], dramatically alters the EST, extensively revising some of the cornerstones of the two earlier theories. (The version of Chomsky's theory presented here, which represents the current form of the theory to the best of my knowledge, derives not only from the papers cited, but also from lecture notes [Chomsky 75a; Freidin 77], and many conversations with several linguists (in particular C. Thiersch). For an early formulation of the theory intended for the non-technical public, see Chap. 3 of [Chomsky 75].)

As discussed in Chapter 5, the current theory, which I will call the Annotated Surface Structure Theory (ASST), entirely rejects the notion that the level of deep structure provides an input to the semantic component at all. As in the standard theory, semantic interpretation once again has but one level of linguistic representation as input, but in the ASST, this level is

the level of surface structure, annotated by the addition of *traces*. The key realization leading to this aspect of the ASST is that the only aspect of semantic information easily accessible from the level of deep structure, the predicate/argument structure of the utterance, can be encoded into surface structure by the addition of traces, "placemarkers" which show the underlying locations of the various NPs which have been shifted by the transformational component. This can be done quite naturally by having all the rules of the transformational component leave behind such a trace each time an NP is moved. As explained in Chapter 5, a trace consists merely of an NP which dominates no lexical material, but which is *bound* to the NP that originated in that location.

The ASST differs from Chomsky's two earlier theories in one other revolutionary manner. Outside of what can be called "housekeeping" transformations, transformations which handle various small details of grammatical form, the transformational component consists of exactly two transformations, which can be stated as the rules "MOVE NP" and "MOVE WH-phrase". Furthermore, in contrast to the transformations of the earlier theories, which consisted of fairly detailed patterns, called "structural descriptions" and actions, called "structural changes", these two rules are exceedingly general. In fact, the statement of the two rules given above, simply "Move Np" and "Move Wh-phrase", contains approximately the full level of detail that these rules provide. Stated informally, the rule MOVE NP takes an NP, any NP, and moves it anywhere. The rule MOVE WH-phrase takes a constituent which is labelled "+WH", i.e. an NP like "which boy" or "who" or a PP which has such an NP as its object, and moves it under a

node called "COMP" which also dominates complementizers like "that" and "for" (as in "for John to do it").

Clearly, these two rules unconstrained would allow all sorts of ungrammatical sentences to be derived, but the slack in the statement of the rules is taken up by a set of *constraints* which are claimed to be *grammatical universals*. Four of these constraints in particular are crucial to the theory; I will briefly discuss all of them here.

The Structure Preserving Hypothesis

Perhaps the least technical of these constraints is what is called the Structure Preserving Hypothesis (SPH), first proposed by Emonds [Emonds 70, 76]. The SPH states that a transformation can only move a constituent into a position where a constituent of that type could have been generated at the level of deep structure and which is currently unfilled by a constituent that dominates lexical material (i.e. constituents *can* be moved into a position that is filled by a trace). (This constraint does not apply to transformations such as the rule of Auxiliary Inversion which can only apply to what Emonds calls "root sentences", a category which includes major clauses and perhaps a few other environments.)

The SPH constrains the rule MOVE NP so that it can only move NPs into locations where an NP could have been generated at the level of deep structure. For example, if it is assumed that passive sentences are generated in underlying structure without subjects, as in Chapter 5, then the rule of MOVE NP can move one of the objects of the clause into the previously

unfilled subject position of that clause, but it cannot move an NP into the middle of the auxiliary where an NP cannot appear in underlying structure, thus eliminating such ungrammatical sentences as that given as 6.1.a immediately below, which would be derived from the structure that underlies 6.1.b:

6.1.a*John could have Mary given a book.

6.1.b John could have given Mary a book.

While I will have nothing further to say about the SPH below, it is worthy of mention in this context because the theory of deterministic parsing can provide a general sort of motivation for this constraint, although I have no specific proof of its necessity. The SPH, if true, greatly reduces the number of possibilities between which a deterministic parser must choose as it analyses syntactic structures. The SPH implies that there is a fixed order in which a parser can look for the subconstituents of any constituent except for a root S. Since a constituent can only be moved into a position where a constituent of that type could have been generated in underlying structure, the order in which a parser must search for constituents in surface structure is constrained to be simply the order in which they can be generated in the base. While the deep grammatical function of a constituent cannot be deduced merely from its surface position, the SPH serves to factor into two separate subtasks the problems of finding the structure of an utterance and deciding upon the deep grammatical function of its constituents once found.

From this same point of view, it is interesting to note that the

environment in which non-structure preserving operations *can* take place, i.e. the environment of root Ss, is the one environment in which a deterministic parser does not have to deal with the problem of where one constituent ends and a higher constituent begins. One cannot help but notice that the one environment in which the SPH does not "operate", thereby increasing the computational burden on a deterministic parser, is exactly the one environment in which another large computational burden is absent.

Subjacency

The three other constraints that Chomsky proposes are all fairly technical in nature. They are 1) the principle of Subjacency, 2) the Specified Subject Constraint, 3) the Tensed S Constraint (also called the Propositional Island Constraint). I will discuss each one briefly in turn.

Before doing so, however, I must note that these three constraints are not intended to merely constrain syntactic rules, i.e. transformations, but also the rules of superficial semantic interpretation that Chomsky calls "Semantic Interpretation I" (SI-1) rules, also called "rules of construal". SI-1 rules are rules of semantic interpretation that map annotated surface structures onto a level of structure that Chomsky calls "logical form". These rules differ from later semantic interpretation rules, called "Semantic Interpretation 2" (SI-2) rules, in that SI-2 rules can also take into account pragmatic and discourse information; the rules of SI-1, according to Chomsky's theory, have only the annotated surface structure as input. Since processes of semantic interpretation are outside the scope of this document, in what follows I will largely ignore the fact that these rules are intended to

apply to semantic processes as well as purely syntactic processes. For more on all of this, the non-linguist is again encouraged to read Chapter 3 of [Chomsky 75], which is an introductory presentation of the notions outlined here.

The principle of Subjacency, informally stated, says that no rule can involve constituents that are separated by more than one cyclic node. Let us say that a node X *is subjacent to* a node Y if there is at most one cyclic node, i.e. at most one NP or S node, between the cyclic node that Dominates Y and the node X. (To say the same thing in another way, X is subjacent to Y iff X and Y are either Dominated by the same cyclic node or if the cyclic node that Dominates X is itself Dominated by the cyclic node that Dominates Y.) Given this definition, the Subjacency principal says that no rule can involve constituents that are not subjacent.

Restricting our attention to the two transformations mentioned above, the Subjacency principal implies that the rules MOVE NP and MOVE WH-phrase are constrained so that they can move a constituent only into positions that the constituent was subjacent to. This means that if α, β, and ϵ in Figure 6.2 are cyclic nodes, no rule can move a constituent from position X to either of the positions Y, where $[_{\alpha}...X...]$ is distinct from $[_{\alpha}X]$.

$[_{\epsilon}...Y...[_{\beta}...[_{\alpha}...X...]...]...Y...]$

Figure 6.2 - Subjacency: no rule can involve X and Y in this structure.

Subjacency implies that if a constituent is to be "lifted" up more than one level in constituent structure, that this operation must be done by repeated operations. Thus, to use one of Chomsky's examples, the sentence given in Figure 6.3.a, with a deep structure analogous to 6.3.b, must be derived as follows (assuming that "is certain", like "seems", has no subject in underlying structure): The deep structure must first undergo a movement operation that results in a structure analogous to 6.3.c, and then another movement operation that results in 6.3.d, each of these movements leaving a trace as shown. That 6.3.c is in fact an intermediate structure is supported by the existence of sentences such as 6.3.e, which purportedly result when the ∇ in the matrix S is replaced by the lexical item "it", and the embedded S is tensed rather than infinitival. The structure given in 6.3.f is ruled out as a possible annotated surface structure, because the single trace could only be left if the NP "John" was moved in one fell swoop from its underlying position to its position in surface structure, which would violate subjacency.

(a) John seems to be certain to win.
(b) ∇ seems [$_S$ ∇ to be certain [$_S$ John to win]]
(c) ∇ seems [$_S$ John to be certain [$_S$ *t* to win]]
(d) John seems [$_S$ *t* to be certain [$_S$ *t* to win]]
(e) It seems that John is certain to win.
(f) John seems [$_S$ ∇ to be certain [$_S$ *t* to win]]

Figure 6.3 - An example demonstrating subjacency.

The Specified Subject Constraint

The Specified Subject Constraint (SSC), stated informally, says that no rule may involve two constituents that are Dominated by different cyclic nodes unless the lower of the two is the subject of its S or NP (taking the

head of an NP to be its subject). Thus, no rule may involve constituents X and Y in the structure shown in Figure 6.4 below, if α and β are cyclic nodes and Z is the subject of α, Z distinct from Y.

$[_{\beta}...Y...[_{\alpha}\ Z...X...]...]...Y...]$

Figure 6.4 - SSC: No rule can involve X and Y in this structure.

The SSC explains why the surface subject position of verbs like "seems" and "is certain" which have no underlying subject can be filled only by the subject and not the object of the embedded S: The rule MOVE NP is free to shift any NP into the empty subject position, but is constrained by the SSC so that the object of the embedded S cannot be moved out of that clause. This explains why (a) in Figure 6.5 below, but not 6.5.b, can be derived from 6.5.c; the derivation of 6.5.c would violate the SSC. (The SSC also implies that the derivations of sentences which exhibit what is known as Tough-movement must involve some rule other than MOVE NP; see [Chomsky to appear] for more on this.)

(a) John seems to like Mary.
(b)*Mary seems John to like.
(c) ∇ seems $[_S$ John to like Mary]

Figure 6.5 - Some examples illustrating the SSC.

The Tensed S Constraint

Finally, we come to the last of the four constraints, the Tensed S Constraint (TSC). The TSC, informally stated, says that no rule can involve two constituents X and Y Dominated by different cyclic nodes, if the "lower"

of them is Dominated by a tensed S, i.e. no rule can involve X and Y in the structure shown in (a) in Figure 6.6 below, if α is a tensed S. The TSC accounts for why 6.6.b can be derived from 6.6.c, while 6.6.d, which is derived from 6.6.e is ungrammatical. The structure depicted by 6.6.e is identical to that depicted by 6.6.c except for the fact that 6.6.e is tensed and 6.6.c is infinitival, but exactly this fact causes the TSC to block the derivation. (The structure depicted by 6.6.e can give rise to the quite acceptable sentence given as 6.6.f, where the ∇ has been replaced by the lexical item "it".)

(a) $[_{\beta}...Y...[_{\alpha}...X...]...]...Y...]$
(b) John seems to like Mary.
(c) ∇ seems $[_S$ John to like Mary]
(d)*John seems likes Mary.
(e) ∇ seems $[_S$ John likes Mary]
(f) It seems John likes Mary.

Figure 6.6 - Some examples illustrating the Tensed S Constraint.

The Constraints Imposed by the Grammar Interpreter

We now turn to the aspects of these constraints that are accounted for by the operation of the grammar interpreter.

It should be stated at the outset that the range of phenomena which are accounted for by the structure of the grammar interpreter is more limited than that accounted for by Chomsky's constraints. I will show only that the structure of the interpreter constrains the class of syntactic processes which Chomsky characterizes by the competence rules "MOVE-NP" (in this chapter) and "MOVE WH-phrase" (in the next). As stated above, Chomsky's

constraints are intended to apply to all "rules of grammar", both syntactic rules (i.e. "transformations") and those rules of semantic interpretation which Chomsky now calls "rules of construal", a set of "shallow semantic" rules which govern anaphoric processes [Chomsky to appear]. The discussion here will only touch on purely syntactic phenomena; the question of how rules of semantic interpretation can be meshed with the framework presented in this document is beyond the range of the present research. This would seem to be a fertile area for future investigation.

Another important limitation of this work is that the arguments to be presented below deal only with English, and in fact depend strongly upon several facts about English syntax (e.g. the fact that English is subject-initial). Whether these arguments can be successfully extended to other languages is an open question, and to this extent this work must be considered exploratory.

I will not show that these constraints must be true *without exception*; as we will see, there are various situations in which the constraints imposed by the grammar interpreter can be circumvented. Most of these situations, though, will be shown to demand much more complex grammar formulations than those typically needed in the grammar so far constructed. In this sense, the constraints discussed here will not be violated by "typical" grammar rules and the situations in which the constraints are violated will be quite atypical in that they demand grammar rules far more complex than the typical rule. This is quite in keeping with the suggestion made by Chomsky [Chomsky to appear] that the constraints are not

necessarily without exception, but rather that exceptions will be "highly marked" and therefore will count heavily against any grammar that includes them. If the degree of markedness is equated with rule complexity (as assigned by an appropriate metric), this falls out immediately from the discussion below. For the sake of convenience, I will often speak of these constraints in the following sections as if they were without exception; where I use "must" the reader should substitute "almost always will".

Finally, I should note that this chapter deals only with those grammatical processes characterized by the competence rule "MOVE NP"; the next chapter will deal with those processes characterized by the rule "MOVE WH-phrase". The reason for this is the following: As Chomsky notes, the phenomena he characterizes by the rule "MOVE WH-phrase" do not, in fact, appear to observe his constraints. Instead, Chomsky argues that his constraints account for the behavior stipulated by another more specific constraint, Ross's Complex NP Constraint [Ross 67]. What I will show in the next chapter is that the behavior characterized by the Complex NP Constraint itself follows directly from the structure of the grammar interpreter for rather different reasons than the behavior considered in this chapter. The claim then, is that the grammar interpreter accounts for the same range of behavior (given the caveats above) that Chomsky's two constraints account for, but in rather different ways for MOVE NP and MOVE Wh-phrase phenomena.

The Specified Subject Constraint and the Grammar Interpreter

In an earlier section, I pointed out that within Chomsky's framework, the Specified Subject Constraint constrains the rule "MOVE NP" in such a way that only the subject of a clause can be moved out of that clause into a position in a higher S. Thus, if a trace in an annotated surface structure is bound to an NP Dominated by a higher S, that trace must fill the subject position of the lower clause. In this section I will show that the grammar interpreter constrains grammatical processes in such a way that annotated surface structures constructed by the grammar interpreter will have this same property.

In terms of the parsing process, this means that if a trace is "lowered" from one clause to another during the parsing process, then it will be attached as the subject of the second clause, unless it is created by a WH-movement process. To be more precise, if a trace is attached so that it is Dominated by S1, and the trace is bound to an NP Dominated by some other S node S2, then that trace will necessarily be attached so that it fills the subject position of S1. This is depicted in Figure 6.7 below. This will be true presupposing the structural analyses for passivization and other constructions involving traces presented in previous chapters and assuming, again, that the grammar does not violate a small set of restrictions.

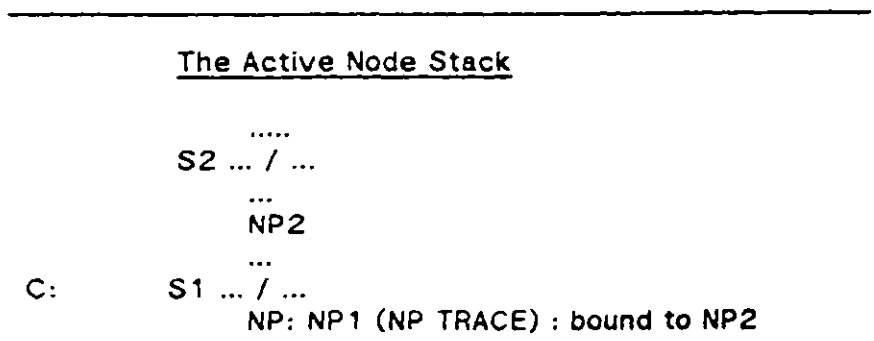


Figure 6.7 - NP1 must be attached as the subject of S1 since it is bound to an NP Dominated by a higher S.

Consider the process by which the trace *t1* in (1) below is created, bound to the NP "the meeting", and attached as the subject of the embedded clause.

(1) The meeting seems [$_S$ *t1* to have been scheduled for Wednesday].

|_____|

(The process of parsing this example, originally given in Figure 5.16, was thoroughly discussed in that chapter. The reader might find it helpful at this point to review that discussion, especially the discussion between Figures 5.21 and 5.25.) Immediately after the verb "seems" is attached to the VP of the major clause, the PASSIVE rule is executed on the advice of the rule SEEMS. This rule creates a trace, binds it to the subject of the dominant S, and then drops this new trace into the first position in the buffer. If this was a simple passive, the OBJECT rule would now attach the trace to the VP of the current clause. In this case, however, the rule INF-S-START1, with pattern

[=np] [=*to] [=tnsless]

will trigger, creating a new S node and then attaching the trace to that S node as its subject. The essence of the process is as follows: create and

properly bind a trace while the major S is the current S; drop the trace into the buffer; create a subordinate S; attach the trace to the newly created S.

The end result of this process is that a trace bound to an NP in a higher S has been attached as the subject of an embedded S without having been explicitly "lowered" from one S to the other. The original point of the "seems" example, of course, was that the rather simple PASSIVE rule handles both this case and the case of simple passives without the need for some mechanism to explicitly lower the NP. The PASSIVE rule captures this generalization by dropping the trace it creates into the buffer (after appropriately binding the trace), thus allowing other rules written to handle normal NPs to correctly place the trace.

This statement of PASSIVE does more, however, than simply capture a generalization about a specific construction. As I will argue in detail below, the behavior specified by both the Specified Subject Constraint and Subjacency follows almost immediately from this formulation. I will also argue that this formulation of PASSIVE is the only simple, non-*ad hoc*, formulation of this rule possible, and that all other rules characterized by the competence rule "MOVE NP" must operate similarly; I will attempt to show that all other possible formulations are blocked by the nature of the grammar interpreter. If this is true, then the behavior characterized by these constraints in enforced, in essence, by the grammar interpreter.

While it is easy to show that the behavior characterized by the SSC follows from the PASSIVE rule as stated here, the demonstration that other

possible formulations of PASSIVE are blocked by the structure of the grammar interpreter is quite tedious because other possible formulations of this rule must be ruled out one at a time. It will also be necessary to impose a subsidiary constraint on the operation of the grammar interpreter, which I will call the *Left-to-Right Constraint*, which says, in essence, that constituents must be removed from the buffer in left-to-right order. Because of this, I will present the basis of the argument assuming these two points to be true, and will then go back and discuss the two "lemmas". The two lemmas, stated informally, are the following:

> The "Lowering" Lemma: the only reasonable method for "lowering" a trace bound to an NP in one clause into a lower clause is to do so implicitly by dropping the trace into the buffer.
>
> The Left-to-Right Constraint: the constituents in the buffer are (almost always) attached to higher level constituents in left-to-right order, i.e. the first constituent in the buffer is (almost always) attached before the second constituent.

Given these lemmas, it is easy to show that a trace bound to an NP in one clause can only serve as the subject of a clause dominated by that first clause.

By the Lowering Lemma, a trace can be "lowered" into one clause from another only by the indirect route of dropping it into the buffer before the subordinate clause node is created, i.e. a trace can be "lowered" only by the method that is embodied in the formulation of the Passive rule. This means that the ordering of the operations is crucially: 1) create a trace and drop it into the buffer, 2) create a subordinate S node, 3) attach the trace to the newly created S node. The key point here is that at the time that the subordinate clause node is created and becomes the current active node, the

trace must be sitting in the buffer, filling one of the three buffer positions. Thus, the parser must be in one of the three states shown in Figure 6.8 below. The configuration shown in Figure 6.8.a is the most likely state that the parser will be in at this time, since the states shown in (b) will occur only if other nodes have been created and dropped into the buffer after the trace was. (The reader is reminded that a node dropped from the active node stack into the buffer always goes into the first buffer position.)

The Active Node Stack

C: S123 (S SEC ...) / ...

The Buffer

1 : NP123 (NP TRACE) : bound to NP in S above S123
2 : ...
3 : ...

(a) Most likely parser state

The Buffer

1 : ...
2 : NP123 (NP TRACE) : bound to NP in S above S123
3 : ...

The Buffer

1 : ...
2 : ...
3 : NP123 (NP TRACE) : bound to NP in S above S123

(b) Other possible configurations of the buffer

Figure 6.8 - Possible parser states after embedded S created.

Now, given the L-to-R Constraint, a trace which is in the buffer at the time that an embedded S node is first created must be one of the first several constituents attached to the S node or its daughter nodes. From the structure of English, we know that the leftmost three constituents of an embedded S node, ignoring topicalized constituents, must either be

COMP NP AUX or NP AUX [$_{VP}$ VERB ...].

(The COMP node will dominate flags like "that" or "for" that mark the beginning of a complement clause.) But then, if a trace, itself an NP, is one of the first several constituents attached to an embedded clause, the only position it can fill will be the subject of the clause.

This is all that must be demonstrated to capture the behavior characterized by the SSC. I have shown that any trace bound to an NP in a higher clause which is attached to a lower clause can only be attached as the subject of that clause. This is exactly the empirical consequence of Chomsky's Specified Subject Constraint in such cases as explained above.

The L-to-R Constraint and the Lowering Lemma

I will me now discuss the two lemmas assumed in the previous argument one at a time. (On a first reading, the reader may wish to skip this section and continue with the discussion of Subjacency.)

The crux of the Lowering Lemma is that the only way to place a trace bound to an NP in a clause S_i into a second clause dominated by S_i is by dropping it into the buffer. I will first point out that if a trace is to be lowered into an embedded S at all, then it must be dropped into the buffer.

I will then show that there are no other possible alternatives to lowering such a trace.

First of all, the buffer is the only mechanism in the grammar interpreter in which a constituent can be stored; this model includes nothing similar to the registers of an ATN. (The registers included in PIDGIN (see Appendix B) are a programming convenience for the sake of semantics; they are unused by grammatical processes. The reader can check that the contents of the registers in the current grammar have no effect at all upon the operation of the parser.) Since the interpreter mechanism includes nothing like the registers of an ATN, there can be no analog of the "SENDR" action of an ATN by which the contents of a register at one level of the ATN network can be lowered into a lower level of the ATN. Thus, there is no mechanism which would make it possible to lower a trace which is created and bound while one S is the current S into a subordinate clause and then to attach the trace to a node Dominated by the lower S.

I will now go through the range of alternatives to "lowering" a bound trace one at a time, showing each to be infeasible in turn. An exhaustive listing of the range of alternatives is shown in Figure 6.9 below in terms of when the trace is created and bound with respect to the processing of the subordinate clause to which the trace is eventually attached. As Figure 6.9 shows, there are only six possible combinations. Each of these will be considered in turn.

	Create time	Binding time
1)	t1	t1
2)	t1	t2
3)	t1	t3
4)	t2	t2
5)	t2	t3
6)	t3	t3

t1: Before the embedded S is created
t2: After the embedded S is created, while it is in the stack.
t3: After the embedded S is created, after it is popped from the stack.

Figure 6.9 - All possible creation and binding times for an embedded trace.

Possibility (1) in Figure 6.9, that the trace is both created and bound before the embedded S node is created, is exactly the combination embodied in the passive rule. If a trace is created before the S node which will become its father, then it must be dropped into the buffer before that S is created. Otherwise, it will be dropped into the buffer after the S has been popped from the active node stack, and some grammar rule will be forced to attach it to a node which is sitting in the buffer. This possibility is ruled out by the following constraint imposed by the grammar interpreter: the only nodes to which nodes can be attached are the the current active node and the current S. This constraint reflects the functions of the buffer and the stack as discussed in Chapter 3, viz. the presence of a node in the stack signals that the parser is attempting to determine its daughters, while the presence of a node in the buffer signals that the parser is attempting to to determine its father. Note also that this amounts to the following restriction: the grammar interpreter does not provide a general two place command "Attach *x* to *y*", but rather two one place commands "Attach-to-C *x*" and "Attach-to-the-current-S *x*". This alternative formulation is quite in

keeping with the goal of restricting as much as possible the formalism for writing grammar rules, restricting as much as possible the class of possible grammar rules.

Possibilities (2) and (3) differ from (1) only in that the trace is to be bound at some later time. Since both of these possibilities share with possibility (1) the fact that the trace is created before the embedded S, and therefore must be dropped into the buffer if the trace is to be properly attached, they are both consistent with the lowering lemma, and need not be shown infeasible. (Actually, both *are* infeasible, but this can be left as an exercise for the reader.)

One possible alternative to lowering a trace might be to create a trace while the embedded S node is in the stack, consistent with possibilities (4) and (5) in Figure 6.9. This alternative is blocked by the fact that a trace should only be created if the higher clause is passive or has some other specific property that calls for the creation of a trace. Because the only nodes in the active node stack that are specifically accessible to the grammar interpreter are the current active node and the current S node, no grammar rule can possibly access the higher node to check whether a higher S node is passive or not. For the same reasons, even if a grammar rule could somehow create a trace only when appropriate, it could not access the subject of the higher clause to properly bind the newly created trace.

Note that this limitation of the grammar interpreter cannot be considered a constraint in the usage of this word within current linguistic

theory in that this limitation does not follow from any *stipulated* restriction upon the operation of the interpreter. Instead, this limitation stems from the fact that there is no tree-climbing operator in PIDGIN, i.e. that there is no operator which accesses the father of a given node. If the absence of a particular operation in a mechanism is to be considered a constraint, then the weaker the mechanism, the more implicit constraints it involves. This position is clearly nonsensical.

The last alternative is to defer creating the trace until the subordinate clause has been parsed and the subordinate S node has been popped from the active node stack, consistent with possibility (6) in Figure 6.9. A trace could then be created and attached to the subordinate clause, now presumably attached to the VP of the current S. This possibility is ruled out by the constraint stated above which allows attachment only to the current active node and the current S.

Since all possibilities inconsistent with the Lowering Lemma have been shown to be impossible, and since the PASSIVE rule clearly does allow traces to be lowered, the Lowering Lemma follows.

I will now turn to the Left-to-Right Constraint. I will not attempt to prove that this constraint must be true, but merely to show why it is plausible.

For the grammar of English included in Appendix D, and, it would seem, for any grammar of English that attempts to capture the same range of

generalizations as this grammar, the constituents in the buffer are attached to the current active node (and thus are removed from the buffer by the grammar interpreter) in left-to-right order, with a small range of exceptions. This usage is clearly not enforced by the grammar interpreter as presently implemented; it is quite possible to write a set of grammar rules that specifically ignores a constituent in the buffer until some arbitrary point in the clause, though such a set of rules would be highly ad-hoc. As we will soon see, however, the assumption that constituents are removed from the buffer in left-to-right order will turn out to be crucial not only for the argument presented here, but also for the demonstration in the next section that the grammar interpreter enforces Subjacency as well as the SSC.

For this reason, I will make the following assumption in the discussion that follows: the constituents in the buffer are almost always attached to higher level constituents in left-to-right order, i.e. the first constituent in the buffer is almost always attached before the second constituent. The one exception to this seems to be that a constituent C_i may be attached before the constituent to its left, C_{i-1}, if C_i does not appear in surface structure in its underlying position (or, if one prefers, in its unmarked position) and if its removal from the buffer reestablishes the unmarked order of the remaining constituents. The primary exceptions to left-to-right attachment in the current grammar are the rule AUX-INVERSION, discussed in Chapter 5, and the various rules for adverb attachment that will be discussed in Chapter 8. These rules all reestablish the unmarked order of the clause. In the discussion that follows, I will refer to the assumption stated here as the Left-to-Right (L-to-R) Constraint.

(A possible reformulation of the L-to-R Constraint which may include the exceptions mentioned above is the following: All constituents must be attached to higher level constituents according to the left-to-right order of constituents in the unmarked case of that constituent's structure. While I have no strong evidence for this restatement, it does seem to handle the above exceptions, if it is assumed that adverbs attach at the beginning of the constituents that they modify.

This reformulation is interesting in that it is a natural consequence of the operation of the grammar interpreter if the association of packets with phrase structure rules suggested in Chapter 4 were implemented. This scheme would add a "base component" of phrase structure rules to the grammar. A packet of grammar rules would then be explicitly associated with each symbol on the right hand side of each phrase structure rule. A structure of a given type would then be constructed by activating the packets associated with each node type of the appropriate phrase structure rule in left-to-right order. Since these base rules would reflect the unmarked l-to-r order of constituents, the constraint suggested here would then simply fall out of the interpreter mechanism.)

Subjacency and the Grammar Interpreter

Chomsky's Subjacency constraint, stated informally, says that no rule can affect constituents in two distinct clauses (or more generally, two cyclic constituents) S1 and S2 unless S1 Dominates S2 (i.e. S1 dominates S2 and there is no third clause S3 that "comes between" S1 and S2). In terms of

the competence rule "MOVE-NP", this means that an NP can be moved only within the clause in which it originates, or into the clause that Dominates that clause. In this section I will show that the parsing correlate of this constraint follows from the structure of the grammar interpreter. Specifically, I will show that there are only limited cases in which a trace can be "lowered" more than one clause, i.e. that a trace created and bound while any given S is current must almost always be attached either to that S or to an S which is Dominated by that S.

Let us begin by examining what it would mean to lower a trace more than one clause. Given that a trace can only be "lowered" by dropping it into the buffer and then creating a subordinate S node, as discussed above, lowering a trace more than one clause necessarily implies the following sequence of events, depicted in Figure 6.10 below: First, a trace NP1 must be created with some S node, S1, as the current S, bound to some NP Dominated by that S and then dropped into the buffer. By definition, it will be inserted into the first cell in the buffer. (This is shown in Figure 6.10.a). Note that throughout this figure the current S is shown as the current active node for the sake of clarity.) Then a second S, S2, must be created, supplanting S1 as the current S (fig. 6.10.b). (NP1 is no longer necessarily in the first buffer position.) Next, a third S, S3, must be created, becoming the current S (fig. 6.10.c). During all these steps, the trace NP1 remains sitting in the buffer. Finally, NP1 is attached under S3 (fig. 6.10.d). By the Specified Subject Constraint, NP1 must then attach to S3 as its subject.

The Active Node Stack

.....

C: S1 ... / ...

The Buffer

1st: NP1 (NP TRACE) : bound to NP Dominated by S1

...

(a) - NP1 is dropped into the buffer while S1 is the current S.

The Active Node Stack

.....

S1 ... / ...

C: S2 ... / ...

The Buffer

...

NP1 (NP TRACE) : bound to NP Dominated by S1

...

(b) - S2 is created and becomes the current S.

The Active Node Stack

.....

S1 ... / ...

S2 ... / ...

C: S3 ... / ...

The Buffer

...

NP1 (NP TRACE) : bound to NP Dominated by S1

...

(c) - S3 is created, with NP1 still in the buffer.

The Active Node Stack

.....

S1 ... / ...

S2 ... / ...

C: S3 ... / ...

NP: NP1 (NP TRACE) : bound to NP Dominated by S1

The Buffer

...

(d) - NP1 is attached to S3 as its subject (by the SSC).

Figure 6.10 - Lowering a trace more than 1 clause

I will show in what follows that this sequence of events is impossible. The essence of the argument is this: Nothing in the buffer can change between the time that S2 is created and S3 is created if NP1 remains in the buffer. This follows from the fact that NP1, like any other node that is dropped from the active node stack into the buffer, is inserted into the first buffer position, coupled with the fact that by the L-to-R Constraint, nothing to the right of NP1 can be attached to a higher level constituent until NP1 is attached. But if nothing in the buffer changes between the time that S2 is created and S3 is created, then there is no motivation for creating both of these clauses from the same input data. Therefore, there is no motivation for generating S3, and NP1 will thus be attached to S2.

I now turn to the details of the argument. (The reader may wish to skip to the concluding paragraph of this section on the first reading.)

To make the argument rigorous, one assumption not stated above is needed: that no grammatical processes will insert any constituents into the buffer between the time that NP1 is dropped into the buffer and the creation of S2. While I cannot prove that this must be the case, there are a number of reasons which make this assumption plausible. If this assumption were not true, then either 1) some constituent was constructed between the time that the trace was dropped into the buffer and S2 was created or else 2) a previously complete constituent was kept on the active node stack until after the trace was created and dropped. Case (1) is ruled out by the L-to-R Constraint; the newly constructed constituent must necessarily be built up

of constituents to the right of NP1, which is forbidden by the L-to-R constraint. Case (2) is in fact possible, but it assumes a purely gratuitous action: retaining a complete constituent on the active node stack until the trace was created instead of simply popping the completed constituent from the active node stack when first completed. Thus, outside of this one rather implausible possibility, this assumption seems valid.

Given this assumption, the desired result follows rather quickly.

First of all, the contents of the buffer will remain unchanged between the time that S2 is created and the time that S3 is created if NP1 remains in the buffer. As was shown in the immediately preceding discussion, it is most unlikely that constituents will enter the buffer after NP1 is inserted in the buffer, and thus NP1 will remain in the first buffer position. But if NP1 is not removed from the buffer, then no constituents to its right can be removed, by the L-to-R constraint. Thus, the contents of the buffer cannot change.

But if the contents of the buffer do not change between the creation of S2 and S3, then what can possibly motivate the creation of both S2 and S3? The contents of the buffer must necessarily provide clear evidence that both of these clauses are present, since, by the determinism hypothesis, the parser must be correct if it initiates a constituent. Thus, the same three constituents in the buffer must provide convincing evidence not only for the creation of S2 but also for S3. Furthermore, if NP1 is to become the subject of S3, and if S2 Dominates S3, then it would seem that the

constituents that follow NP1 in the buffer must also be constituents of S3, since S3 must be completed before it is dropped from the active node stack and constituents can then be attached to S2. But then S2 must be created *entirely* on the basis of evidence provided by the constituents of another clause (unless S3 has less than three constituents). Thus, it would seem that the contents of the buffer cannot provide evidence for the presence of both clauses unless the presence of S3, by itself, is enough to provide confirming evidence for the presence of S2. This would be the case only if there were, say, a clausal construction that could only appear (perhaps in a particular environment) as the initial constituent of a higher clause. In this case, if there are such constructions, a violation of subjacency should be possible.

With the one exception just mentioned, there is no motivation for creating two clauses in such a situation, and thus the initiation of only one such clause can be motivated. But if only one clause is initiated before NP1 is attached, then NP1 must be attached to this clause, and this clause is necessarily subjacent to the clause which Dominates the NP to which it is bound. Thus, the grammar interpreter enforces the Subjacency Constraint.

There is, in fact, one situation in which structural subjacency can be violated without a trace being lowered more than one clause. Assume that the above argument is correct, and that NP1 must attach to S2, i.e. that it must attach to an S subjacent to the S under which it was created. But even if this is true, after S2 is constructed, it might then be dropped back into the buffer. (This, of course, will only happen if S2 is not attached to some higher level constituent at the time of its creation.) At this point, a

rule might trigger on the contents of the buffer, which now may include not only S2, but two succeeding constituents as well. This hypothetical rule might then create another S node, S4, and then attach S2 to S4. Since NP1 is not bound to an NP Dominated by S4, subjacency has been violated. The point is that subjacent in terms of the grammar interpreter mechanism means subjacent in terms of processing, which is not necessarily the same as subjacency in terms of final attachment.

This situation, which seems not unlikely, is made less likely by the following observation: Note that when NP1 is dropped into the buffer, it will necessarily fill the first buffer position. But this means that it must be the leftmost constituent of the S to which it is attached, S2, if that S is created after NP1 is dropped into the buffer. In particular, this means that S2 cannot have an explicit complementizer. Now, assume that S2 is now dropped into the buffer, filling the first buffer position. Then, by the same argument, if the contents of the buffer now trigger the creation of another S, S4, then S4 cannot have an explicit complementizer either. Thus, the resulting structure will have an embedded S as the first constituent of another S neither of which begin with explicit complementizers. Thus, neither sentence will begin with any explicit flag to indicate that they are subordinate clauses.

This situation is impossible for English; a clause which is the initial constituent of another clause *must* be marked by a complementizer. One widely known explanation for this is simply that the parser must know at the time the clause is initiated whether or not it is a subordinate clause.

If the embedded clause follows the verb of the matrix S, then the parser can expect the embedded clause and an explicit flag is unnecessary. But if the embedded clause is an initial constituent of the matrix S, then the embedded S must contain an explicit flag to signal the fact that it is a subordinate S. This seems to indicate that the creation of an S that is not attached at the time of creation is a "highly marked" action, at least for English. If this were not true, a parser could always wait until the end of the clause to decide whether a particular clause was embedded or not.

As a concluding point, it is worthy of note that while the grammar interpreter appears to behave exactly as if it were constrained by the subjacency principle, it is in fact constrained by a version of the Clausemate Constraint! (The Clausemate Constraint, long tacitly assumed by linguists but first explicitly stated, I believe, by Postal [Postal 64], states that a transformation can only involve constituents that are Dominated by the same cyclic node. This constraint is at the heart of Postal's attack on the constraints that are discussed above and his argument for a "raising" analysis.) The grammar interpreter, as was repeatedly stated above, limits grammar rules from examining any node in the active node stack higher than the current cyclic node, which is to say that it can only examine clausemates. This parsing version of the clausemate constraint, in fact, is crucial to the argument presented above showing that subjacency is a natural consequence of the grammar interpreter. The trick is that a trace is created and bound while it is a "clausemate" of the NP to which it is bound in that the current cyclic node at that time is the node to which that NP is attached. The trace is then dropped into the buffer and another S node is

created, thereby destroying the clausemate relationship. The trace is then attached to this new S node. Thus, in a sense, the trace *is* lowered from one clause to another. The crucial point is that while this lowering goes on as a result of the operation of the grammar interpreter, it is only implicitly lowered in that 1) the trace was never *attached* to the higher S and 2) it is *not* dropped into the buffer because of any realization that it must be "lowered"; in fact it may end up attached as a clausemate of the NP to which it is bound - as the passive examples of Chapter 4 made clear. The trace is simply dropped into the buffer because its grammatical function is not clear, and the creation of the second S follows from other independently motivated grammatical processes. From the point of view of this performance theory, we can thus have our cake and eat it too; to the extent that it makes sense to map results from the realm of performance into the realm of competence, in a sense *both* the clausemate/"raising" and the subjacency positions are correct.

These Arguments as Evidence in Support of the Determinism Hypothesis

In closing, I would like to show that the properties of the grammar interpreter crucial to capturing the behavior of Chomsky's constraints were originally motivated by the determinism hypothesis, and thus, to some extent, the determinism hypothesis explains Chomsky's constraints. The extent to which the determinism hypothesis provides an explanation for Chomsky's constraints itself provides support for this hypothesis; to the extent that his constraints follow from a single more general principle, they no longer need to be stipulated, and thus linguistic theory (meaning here the amalgam of

performance and competence theories) is simplified.

The strongest form of such an argument, of course, would be to show that (a) either (i) the grammar interpreter accounts for *all* of Chomsky's constraints in a manner which is conclusively universal or (ii) the constraints that it will not account for are wrong and that (b) the properties of the grammar interpreter which were crucial for this proof were *forced* by the determinism hypothesis. (To show necessity, of course, it must be demonstrated either that no other mechanism would serve to implement a deterministic parser or that any such mechanism would lead to fundamentally the same results.) If such an argument could be made, it would show that those of Chomsky's constraints which are valid follow from the determinism hypothesis, giving strong confirmation to the determinism hypothesis.

I have shown none of the above, and thus my claims must be proportionately more modest. I have argued only that important sub-cases of Chomsky's constraints follow from the grammar interpreter, and while I can show that the determinism hypothesis strongly *motivates* the mechanisms from which these arguments follow, I cannot show necessity. The extent to which this argument provides evidence for the determinism hypothesis must thus be left to the reader; no objective measure exists for such matters.

The ability to drop a trace into the buffer is at the heart of the arguments presented above for subjacency and the SSC as consequences of the functioning of the grammar interpreter; this is the central operation upon which the above arguments are based. Also crucial to the arguments are

various restrictions upon the operation of the grammar interpreter, such as the fact that there is no way to access any nodes in the active node stack except for the current active node and the current cyclic node.

As discussed at the beginning of Chapter 3, the buffer itself, and the fact that a constituent can be dropped into the buffer if its grammatical function is uncertain, are directly motivated by the determinism hypothesis. It was argued in that chapter that a parser must necessarily provide some sort of lookahead mechanism if it is to function deterministically. It was also argued that this lookahead must be constrained in some manner if the determinism claim was to have any content, but that this constraint must be based on some number of constituents, rather than some fixed number of lexical items. It is easy to see that such a requirement also necessarily implies an ability to construct constituents and - in some sense or other - buffer them; the example given in that earlier discussion demonstrated the necessity of holding on to the first three constituents of a clause before the grammatical role of the first could be determined. Given these attributes which a deterministic parser *must* fulfill, the buffer mechanism intuitively seems to be one of the simplest computational devices that fulfills these specifications. In this sense, it seems clear that the buffer mechanism and the ability to drop constituents from the active node stack into the buffer follow naturally and directly from the general principles which a deterministic parser must fulfill.

Again, as for the limitations on the grammar interpreter which prevent it from examining any node on the active node stack above the

current cyclic node, there is nothing here to motivate or explain. What demands explanation and motivation is why a given facility *is* included in the model; the goal of a computational theory of linguistic performance should be to account for a given range of phenomena using the weakest computational mechanisms possible. Thus, there is no need to explain why a mechanism of only limited power has been implemented if it can be shown that the mechanism is powerful enough to do the job that is required.

It is thus the case that the aspects of the grammar interpreter which are crucial to the arguments presented above, namely the buffer mechanism and its ability to buffer a constituent until its grammatical function is clear, are directly motivated by the determinism hypothesis. Given this, it is fair to claim that if Chomsky's constraints follow from the operation of the grammar interpreter, then they are strongly linked to the determinism hypothesis. If Chomsky's constraints are in fact true, then the arguments presented in this chapter provide solid evidence in support of the determinism hypothesis.

CHAPTER 7

PARSING RELATIVE CLAUSES - ROSS'S COMPLEX NP CONSTRAINT

Introduction

In the preceding chapter, I argued that the grammar interpreter behaves as if it obeys two of Chomsky's constraints for the class of syntactic processes which Chomsky characterizes by the competence rule "MOVE-NP". In this chapter, I will turn to those processes characterized within his theory by the competence rule "MOVE-WH-phrase" and show that, for these constructions, the grammar interpreter enforces the behavior stipulated by Ross's Complex NP Constraint (CNPC).

If this is true, it is in some sense irrelevant whether or not the grammar interpreter parallels Chomsky's constraints for "MOVE-WH-phrase" phenomena at all. While Chomsky argues that his constraints apply both to MOVE-NP and MOVE-WH phenomena, he concedes that the MOVE-WH phenomena do not appear to obey his constraints at all. Instead, he shows that his constraints account for the behavior stipulated by the CNPC. If it can be demonstrated that this behavior follows from mechanisms of the grammar interpreter that were motivated by the determinism hypothesis, then much the same range of phenomena for both "MOVE-NP" and "MOVE-WH-phrase" cases has been shown to follow directly or indirectly from the determinism hypothesis.

An Analysis of "Wh-clauses"

Before discussing this issue, it is necessary to present the general outline of an analysis of those constructions which are characterized by the rule "MOVE-WH-phrase", i.e. relative clauses, wh-questions and other constructions that have "displaced" wh-phrases. Something like this analysis must be assumed if any grammar rules are to be written at all for the grammar interpreter which will parse these "wh-clauses", as I will show below. This analysis derives from Chomsky's analysis [Chomsky to appear], and assumes much the same structure as that assumed by his analysis, as we will see.

Wh-clauses cause difficulty because the "gap" in a wh-clause, i.e. the site from which the wh-phrase has been shifted and into which the parser must place a trace, can be arbitrarily far away from its surface location. This is problematic because there may be several intervening S nodes between the wh-phrase and the clause in which the gap appears. Thus, in the sentence shown in Figure 7.1.a below, there are three intervening S nodes between the wh-phrase and the gap, as the analysis shown in 7.1.b reveals; the location of the gap in 7.1.a is marked by a trace which is bound to "Who".

(a) Who did Joe say that Bill had claimed that Bob had told to go jump off a bridge?

(b) [$_{S1}$ [$_{COMP}$ Who] did Joe say [$_{S2}$ that Bill had claimed [$_{S3}$ that Bob had told [$_{S4}$ *t* to go jump off a bridge]]]?]

Figure 7.1 - The gap in wh-constructions can be deeply embedded.

These intervening S nodes are problematic in the context of PARSIFAL because once the S node which Dominates the wh-phrase is no longer the current cyclic node, grammar rules can access neither this S node nor the wh-phrase it Dominates. As the reader should now remember, PARSIFAL does not allow grammar rules to access any nodes on the active node stack except for the current active node and the current cyclic node. Thus, once there is an S node intervening between the S node which Dominates the wh-phrase and the bottom of the active node stack, the intervening S node becomes the current cyclic node and both the wh-phrase and the S that Dominates it become inaccessible.

One solution to this problem, and the solution which I will adopt here, is to simply assign wh-clauses an internal structure similar to that assigned by Chomsky's current analysis of wh-movement, the so-called *cyclic* analysis of wh-movement. In the non-cyclic analysis given in 7.1.b above, the wh-phrase has been attached to the S node via a node called COMP which will also dominate complementizers like "for", "that", etc. I will call such a wh-phrase attached to COMP a Wh-comp. The cyclic analysis assigns to each S node intervening between the S node Dominating the Wh-comp and the site of the gap a structure similar to that assigned to the top-level S of a

wh-clause, with one major difference: each intervening S node has as its Wh-comp not the Wh-phrase itself, but a trace which is bound to the Wh-comp of the next higher S. Thus, the structure of 7.1.a above is something like that shown in 7.2 below, with two COMP nodes dominating both a Wh-comp and the complementizer "that". Each trace in this structure is bound to the Wh-comp that is immediately to its left. The gap-filling trace is underlined for clarity.

[S [COMP Who] did Joe say [S [COMP *t* that] Bill had claimed [S [COMP *t* that] Bob had told [S [COMP *t*] t to go jump off a bridge]]]?]

Figure 7.2 - A cyclic analysis of wh-movement.

Because the binding* of each Wh-comp is the wh-phrase itself (where "binding*" is the transitive closure of the binding relation, as defined in Chapter 5), and because grammar rules can access the binding* of nodes that are themselves accessible, this "cyclic" approach makes the wh-phrase accessible in each S intervening between the wh-phrase and the site of the gap. Thus this approach provides a first step in circumventing the inaccessibility problem.

(I should note here that Chomsky's own analysis assumes the existence of an additional node called S-bar which dominates the COMP node and the S node itself. This node is motivated, at least in part, by considerations that are not applicable in the context of this parser, as I will discuss below, and therefore the existence of this node is not assumed in the analysis presented above for the sake of simplicity. If other motivations for this S-bar node are taken as compelling (e.g. Bresnan's original motivation for

the S-bar node (see [Bresnan 73])), the structures presented here and elsewhere can be modified appropriately; the difference in structure is irrelevant to any of the arguments presented in this document.)

Note that any grammar which uses this analysis must create and appropriately bind a trace that is to serve as a Wh-comp for some S *before that S node is itself created*, a fact which will be crucial in what follows below. After an S node is created, that S node itself is the "nearest" S node, and therefore the Wh-comp of the higher S will be inaccessible. A convenient scheme to circumvent this problem is to create the COMP node, attach the complementizer and the Wh-comp (creating and binding a trace, if necessary), drop the newly parsed COMP into the buffer, and *then* create the S node.

One nice feature of this analysis, as developed so far is that no additional mechanisms have been added to the grammar interpreter to make the Wh-phrase accessible in embedded Ss. Unfortunately, there is one important class of wh-clauses which the grammar interpreter cannot handle without extension.

The problematic case occurs when an NP node intervenes between the S node nearest the bottom of the stack and the location of the gap, as in the following sentence:

7.3 $[_{S1}$ $[_{COMP}$ What] is that $[_{NP1}$ a copy of *t*]?]

In this case, the gap must be detected by the parser while the NP is the current cyclic node, and therefore the Wh-comp of the major S must be

accessible at that time. This means that it will be necessary to access a daughter of a node (i.e. the Wh-comp of S1 in the example above) which is higher in the active node stack than the current cyclic node, in this case the NP node (here, NP1).

One way out of this bind would be to consider NPs to be *non*-cyclic nodes, and for various reasons such a solution looks rather attractive. Such a decision, however, would have far reaching linguistic implications. For example, this analysis would predict that passivized NPs are not "unpassivized" by the parser, but rather would be handled by later semantic processing. (This is equivalent to saying that such passives would be *base generated* within the context of the theory of generative grammar.) While such implications seem quite reasonable, such an analysis cannot be assumed without a careful and detailed linguistic analysis of the full range of implications, a task which is beyond the scope of this document.

Another possible solution would be to allow access to the current cyclic node *and* the cyclic node above that; i.e. to accept Chomsky's subjacency analysis. If such an analysis is unnecessary elsewhere, however, it would seem that allowing access to two cyclic nodes rather than one is conceding a substantial increase in the power of the parser for the sake of solving only this one problem.

Having stated these options, I will now note that there is an unsolved problem here, and provide a special mechanism to circumvent the problem for the time being. This mechanism, in essence, considers NPs to be

non-cyclic nodes for the sole purpose of accessing the Wh-comp of a node. This mechanism is implemented by providing a special phrase in PIDGIN which returns as its value the node which is the "nearest" Wh-comp, the PIDGIN phrase

(the) Wh-comp

where the "the" is optional, as indicated by the parentheses. More precisely, this phrase returns the node which is the Wh-comp of the S node closest to the bottom of the active node stack. If intervening NPs are ignored, this is exactly equivalent to returning the Wh-comp of the "current S node".

Ross's Complex NP Constraint

In the next section, I will demonstrate that the determinism hypothesis forces behavior which accounts for much of the phenomena accounted for by Ross's Complex NP Constraint (CNPC) [Ross 67], given the structural analysis of wh-clauses developed above. Before doing so, let me briefly review Ross's formulation of this constraint.

Consider the sentences shown in Figure 7.4 below. Note that 1(b) and 2(b), each a relativized form of the corresponding (a) sentence, differ in that the wh-phrase in 1(b), but not 2(b), has been extracted from a relative clause. Sentence 2(b) is very acceptable, while 1(b) is extremely bad. Similarly, 3(b) and 4(b), each a relativized form of the corresponding (a) sentence, differ in that the wh-phrase in 3(b), but not 4(b), has been extracted from a "noun complement". And here sentence 4(b) is acceptable while 3(b) is not. (By "noun complement", I mean the S which often follows such nouns as "report", "claim", "rumor", etc.)

1(a) I read a statement which was about that man.
1(b)*The man who I read a statement which was about is sick.

2(a) I read a statement about that man.
2(b) The man who I read a statement about is sick.

3(a) I believed the claim that Otto was wearing this hat.
3(b)*The hat which I believed the claim that Otto was wearing is red.

4(a) I believed that Otto was wearing this hat.
4(b) The hat which I believed that Otto was wearing is red.

Figure 7.4 - Some sentences which motivate the Complex NP Constraint.

Within the framework of generative grammar, one could attempt to explain this data by stipulating that nothing can be moved from within a relative clause, and that nothing can be moved from within a noun complement. These constraints can be combined into one constraint, however, if it is noticed that both relative clauses and noun complements share one important structural property: Both relative clauses and noun complements are clauses which are Dominated by an NP node. Thus, the structures of the relevant NPs in 7.4.1a and 7.5.3a above can be taken to be as shown in Figure 7.6.a and 7.6.b below, respectively. As these analyses show, the crucial difference between relative clauses and noun complements are that 1) a relative clause dominates a trace which is bound to the Wh-comp, and 2) a relative clause is a daughter of an S node, while a noun complement is the daughter of a node called NBAR which is itself dominated by NP. In either case, however, the embedded S is Dominated by the node NP.

(a) I read [$_{NP}$ John's [$_{NBAR}$ [$_{N}$ statement]] [$_{S}$ [$_{COMP}$ which] *t* was about that man]].

(b) I believed [$_{NP}$ John's [$_{NBAR}$ [$_{N}$ claim] [$_{S}$ that Otto was wearing this hat]]].

(c) I believed [$_{S}$ that John [$_{VP}$ [$_{V}$ claims] [$_{S}$ that Otto was wearing this hat]]]

Figure 7.5 - NP Dominates both relative clauses and noun complements.

(The reason for this NBAR node, by the way, is as follows: if we look at an NP as being analogous to an S node at some level, and a noun as similarly analogous to a verb, then we need a node that serves as a "verb phrase" node to complete the analogy. The NBAR node serves this function. Thus, the NP whose structure is shown in 7.5.b is very much analogous in structure to the S whose structure is shown in 7.5.c, although their semantic functions are quite different. This follows from the so-called "X-bar convention", first suggested by Chomsky [Chomsky 70], and well explained in [Jackendoff 74].)

Given this generalization about the structures of relative clauses and noun complements, a combined constraint can be stated which is far more general than either of the "special purpose" constraints stated above. This constraint, stated below in Figure 7.7, is the Complex NP Constraint, as stated by Ross.

No element contained in a sentence dominated by an NP with a lexical head noun may be moved out of that NP by a transformation.

Figure 7.7 - The Complex NP Constraint

It must be noted that the CNPC, as stated by Ross, is gratuitously broad. Ross's data, as presented in his thesis, only supports a weaker form of this constraint, and, as far as I know, no broader argument for the CNPC has been presented. Briefly stated, Ross's data shows only that an *NP* cannot be moved out of a sentence dominated by an NP with a lexical head by *whatever grammatical rules form questions and relative clauses.* It is this weaker form of the CNPC that I will deal with below.

The CNPC and the Determinism Hypothesis

Given the analysis of wh-clauses presented in the previous section, it is now easy to show that the behavior described by the (weakened) CNPC follows in part from the structure of the grammar interpreter and in part directly from the determinism hypothesis.

The CNPC for relative clauses can be dispensed with quite easily. (The form of the following argument is very similar to that presented by Chomsky in [Chomsky 73].)

Consider Figure 7.4.1b, repeated below as Figure 7.8.1. It would seem that the intended analysis of this sentence is as given in 7.8.2, where t_1 to be bound to "which" and t_2 to be bound to "who". To parse this sentence, however, the parser must somehow have access to the COMP of S_1 while in S_2, or else the binding of t_2 could not be accomplished, as discussed at length in the last chapter. But this is impossible, since neither S_1 nor its COMP is accessible from within S_2. Thus, as long as the COMP node can dominate at most one Wh-comp, the only possible NP to which a gap-filling

trace can be bound is the Wh-comp of the current S node. This means that only one NP in a relative clause can be replaced by the grammar interpreter.

---

(1)*The man who I read a statement which was about is sick.

(2)*The man [$_{S1}$ [$_{COMP}$ who] I read a statement [$_{S2}$ [$_{COMP}$ which] t_1 was about t_2]] is sick.

Figure 7.8

I will now turn to the case of noun complements and show that they must behave in accordance with the CNPC.

I will assume below that the grammar must "recover" the "deleted" Wh-comp of a reduced relative clause by inserting some lexical item which serves as the Wh-comp. It is necessary to replace the Wh-comp because the gap-filling trace must be bound to the Wh-comp of the clause, which, of course, makes it necessary that there be one. I will rather arbitrarily use the lexical item "which" for this purpose, simply because it is the only wh-word that does not make an animate/inanimate distinction.

The essence of the argument to be presented below is the following: Because the initial segments of relative clauses and noun complements can be identical, the parser can only be certain that a potential noun complement is not in fact a relative clause when it completes the analysis of such a clause without discovering a gap to fill. This means that a Wh-comp must be attached to the COMP node of the clause if and when it turns out that the clause is a relative clause. Because the clause must be treated as a reduced

relative (the word "that" is always parsed as a complementizer), this Wh-comp must be the reduced relative marker "which". But this means that the Wh-comp *cannot* be set to a trace which is bound to the Wh-comp of the next higher S, because the Wh-comp must be reserved for the reduced relative marker. Thus, any "higher level" Wh-comps are inaccessible within the clause, whether or not the clause turns out to be a noun complement, resulting in exactly the observed behavior stated by the CNPC. In short, since the parser can't tell a noun complement from a relative clause until it is finished with the clause, it must leave open the option that the clause *is* a reduced relative clause, with the reduced relative marker as Wh-comp, and thus cannot initially fill the Wh-comp with a trace that would make a "higher level" Wh-comp accessible within the clause.

The remainder of this section will develop this argument is full detail.

Consider the two sentences 1(a) and 2(a) shown in Figure 7.9 below. The sentence shown in 7.9.1a contains a relative clause, while that shown in 7.9.2a contains a noun complement. These two sentences are identical through the last word of the embedded S, yet the two Ss have different structures, as do the two dominating NPs; the structure of 1(a) is as shown in 1(b) below, while the structure of 2(a) is shown in 2(b). (Note that in 1(b) the COMP node dominates both the complementizer "that" and an inserted "which" as a Wh-comp. Here I follow Chomsky in assuming that "that" is not the Wh-head, but rather a complementizer, and that a relative clause flagged by "that" is in fact a reduced relative clause.)

1(a) I showed John the report that a stranger had taken.
1(b) I showed John [$_{NP}$ the [$_{NBAR}$ report] [$_{S}$ [$_{COMP}$ *which* that] a stranger had taken *t*]].

2(a) I showed John the report that a stranger had taken the plans.
2(b) I showed John [$_{NP}$ the [$_{NBAR}$ report [$_{S}$ [$_{COMP}$ that] a stranger had taken the plans.]]].

Figure 7.9 - Relative clauses and noun complements can be almost identical.

Given this similarity, it would seem that a deterministic parser confronted with a potential noun complement can only determine the internal structure of the clause by attempting to find a gap into which a trace can be placed, just in case the clause is a relative clause. (Exactly what is involved in detecting such a gap deterministically is the focus of Chapter 10; for now I will simply assume that this can be done somehow-or-other.) If no gap is found, the clause is a noun complement; if a gap is detected, the clause is a reduced relative.

This implies that the reduced relative marker must be attached as the Wh-comp of the potential noun complement when and if a gap is discovered. (Note that the internal structure of the reduced relative clause of 7.9.1b above differs from that of the noun complement of 7.9.2b only in that the relative clause has a gap-filling trace and the reduced relative marker serving as a Wh-comp. Otherwise, the two internal structures are identical.) Attaching a Wh-comp after a gap is detected might seem to violate the left-to-right constraint stated in the previous chapter, but a careful examination of the constraint shows that this is not the case. As formulated in Chapter 6, the L-to-R constraint states, roughly, that

constituents *must be removed from the buffer* in left-to-right order. Since the Wh-comp must be attached long after a COMP node dominating the complementizer "that" has been attached to the S, it would be pointless to drop the "which" into the buffer, expecting normal grammatical processes to do the attachment. Instead, the reduced relative marker must be directly attached to the COMP node by a special purpose grammar rule. While this is clearly *ad hoc*, it seems unavoidable, and violates no constraints imposed by the grammar interpreter.

But now note that a gap within this potential noun complement might not indicate that the clause is a reduced relative at all, if the clause is itself dominated by a wh-clause. Ignoring the CNPC, the clause might be a noun complement even if it contains a gap, with this gap "belonging" to a dominating wh-clause. For example, were it not for the CNPC (the violation of which makes this sentence very bad, of course), the sentence shown in Figure 7.10.1 below might be interpreted to have the structure shown in 7.10.2. Under this interpretation, S_1 is a noun complement and the binding* of t_1 is the Wh-comp of the major S. Note, by the way, that the structure here is exactly parallel to the structure shown in Figure 7.2 except for the violation of the CNPC.

(1) Who did John tell us the report that Herbert had met?

(2) [$_S$ [$_{COMP}$ Who] did John tell us [$_{NP}$ the [$_{NBAR}$ report [$_{S'}$ [$_{COMP}$ t_1 that] Herbert had met t_2]]]]

Figure 7.10 - A gap in a possible noun complement could result from a dominating wh-clause (ignoring the CNPC).

Thus, it would seem that a gap in a potential noun complement which is dominated by a wh-clause can indicate one of two possibilities, which I will refer to as possibility RELCL and possibility NCOMP. They are:

Possibility RELCL - that the potential noun complement is actually a relative clause.

Possibility NCOMP - that the clause *is* a noun complement with the gap resulting from the dominating wh-clause.

As I will now show, however, the parser must necessarily commit itself to one possibility or the other before beginning to analyze the potential noun complement; one of these possibilities must be rejected out of hand.

The problem is simply that if possibility NCOMP above is to be left open, then a trace bound to the Wh-comp of the next higher S *must* be created before the node which is the potential noun complement is created. Once the noun complement node is created, the next higher S is no longer accessible to the grammar interpreter, and a trace which is to serve as the Wh-comp cannot be appropriately bound. Thus, either a trace bound to the Wh-comp of the next higher S must be created and bound before the potential noun complement is created or possibility NCOMP is permanently blocked.

But consider what follows if such a trace *is* created and bound.

If this trace is created, then, by the determinism hypothesis, it must be used in the analysis of the sentence. But if this trace must serve as the Wh-comp of the potential noun complement (what else could one do with it?), then possibility RELCL above is necessarily blocked. If the potential noun complement is to be a reduced relative, then it must have the reduced relative marker as its Wh-comp, not the trace. Thus, the creation of a trace to serve as Wh-comp if possibility NCOMP is not to be discarded out of hand eliminates possibility RELCL, since once the trace is created, it *must* become the Wh-comp.

It is thus clear that the grammar interpreter cannot leave open both possibility RELCL and possibility NCOMP simultaneously. If possibility NCOMP is to remain open, a trace must be created before the potential noun complement is initiated to serve as Wh-comp. But creating this trace commits the parser to its use, thus blocking possibility RELCL. Clearly, then, one of these possibilities must be arbitrarily eliminated from consideration by the grammar interpreter. The question to be answered now, of course, is which.

There seems to be a simple criteria which yields a very plausible answer to this question. If possibility RELCL is ruled out, then a trace must always be created to serve as Wh-comp whenever a noun complement dominated by a wh-clause is initiated, just in case it should turn out that the complement contains the gap. However, if it turns out that the

complement *does not* contain the gap, then the trace is gratuitous. If possibility NCOMP is ruled out, on the other hand, then the lexical item "which" can be attached as Wh-comp if and only if it turns out that a potential noun complement is, in fact, a reduced relative. Thus, ruling out possibility RELCL will lead on occasion to the production of a gratuitous parse node, contrary to the determinism hypothesis, while ruling out possibility NCOMP leads to no such result. Therefore, ruling out possibility RELCL is clearly preferable.

This result leads to exactly the behavior stipulated by the CNPC. If possibility RELCL is ruled out, then no NPs can every be moved out of a noun complement by wh-movement processes. But this means that exactly the behavior stipulated in this case by the CNPC is forced by the determinism hypothesis, as claimed.

It has thus been demonstrated that the behavior stipulated by the CNPC for both relative clauses and noun complements follows in part from the structure of the grammar interpreter and in part directly from the determinism hypothesis, given the analysis of relative clauses and noun complements stated in this chapter.

Chomsky's Analysis - a Comparison

It should be noted that the analysis given above differs in one key respect from Chomsky's analysis, which, in essence, derives the CNPC from Subjacency. There is one key case which Chomsky's analysis rules out which is eliminated by neither the CNPC nor the analysis presented above.

As I will show, it is at least questionable whether this case should be ruled out by the same analysis as those cases which fall under Ross's CNPC.

Chomsky shows that within the generative framework, the behavior stipulated by the CNPC falls out of his Subjacency principal, by and large. The rule of MOVE-WH-phrase itself, in one formulation, moves any NP into the COMP of any S. Since this rule is constrained to obey Subjacency, however, MOVE-WH-phrase can only move a given NP into a COMP that the NP is subjacent to, i.e. into the COMP of the S that Dominates that NP or into the COMP of the next higher S. But this blocks the movement of a WH-phrase from inside either a relative clause or a noun complement into the COMP of the next higher S, as Figure 7.11 below shows. The structure shown in 7.11.1, for instance, would be derived by moving the NP "who" first into the COMP of S_2 (leaving trace t_2) and then into the COMP of S_1. But this final step violates Subjacency, since NP_1 intervenes between S_1 and S_2, and thus the COMP of S_2 is not subjacent to the COMP of S_1. Similarly, the structure shown in 7.11.2 would be derived by moving the NP "who" into the COMP of S_2 and then into the COMP of S_1. Again, NP_1 intervenes between S_1 and S_2, and thus the COMP of S_2 is not subjacent to the COMP of S_1.

(1) *The man [$_{S1}$ [$_{COMP}$ who] I read [$_{NP1}$ a statement [$_{S2}$ [$_{COMP}$ which] t_1 was about t_2]]] is sick

(2) *[$_{S1}$ [$_{COMP}$ Who] did John tell us [$_{NP1}$ the [$_{NBAR}$ report [$_{S2}$ [$_{COMP}$ that] Herbert had met t_2]]]]

Figure 7.11 - Subjacency subsumes the CNPC for relative clauses and noun complements.

But now consider the sentences shown in Figure 7.12 below. (Sentences 7.12.1a and 7.12.2 are taken from [Chomsky 73]; 7.12.4 is from [Ross 67].) Chomsky argues that the ungrammaticality of 7.12.1a can be accounted for exactly in the same way as the sentences shown in Figure 7.11 above, since the structure for this sentence, shown in 7.12.b, would be derived by moving "who" from inside NP_2 into the COMP of S_1 in one fell swoop, violating Subjacency. This explanation accounts for the ungrammaticallity of 7.12.2 in exactly the same way. Thus, Chomsky claims that the CNPC is in fact subsumed by Subjacency, with the additional bonus that Subjacency accounts for the ungrammaticality of sentences like these, which do not fall under the CNPC as stated by Ross.

(1)a*Who did you hear stories about a picture of?
(1)b [$_{S1}$ [$_{COMP}$ Who] did you hear [$_{NP1}$ stories about [$_{NP2}$ a picture of t]]]

(2)*What do you receive requests for articles about?

(3) What did Pres. Wiesner forbid the printing of articles about?

(4) What books does the government prescribe the height of the lettering on?

Figure 7.12 - Subjacency predicts that all of these sentences should be bad.

As Chomsky himself notes, however, there is a problem with this analysis. While Subjacency accurately predicts that 7.12.1a and 7.12.2 should be unacceptable, it also predicts that 7.12.3 and 7.12.4 should be equally bad, for exactly the same reasons. Unfortunately, this is not the case. While both (3) and (4) are unacceptable for some speakers, most of these speakers judge them to much better than 7.11.1 and 7.11.2 above. Furthermore, these sentences are quite acceptable for many speakers. While it is not clear why 7.12.1 and 7.11.2 are unacceptable, the Subjacency theory does not seem to adequately characterize what is going on here.

The analysis presented earlier in this chapter does not run into this difficulty; it accounts for the same range of data as the CNPC and says nothing at all about why 7.12.1 and 7.11.2 should be unacceptable. Thus, in this sense, the analysis presented earlier in this chapter is more descriptively adequate than Chomsky's analysis.

(I must note that the success of my analysis hinges crucially upon the stipulation embodied in the grammar interpreter that, in essence, considers only Ss to be cyclic nodes for the purposes of parsing Wh-clauses, and that, therefore, the result is far less strong than it would otherwise be. However, it should be noted that there is no possibility of patching the Subjacency analysis in the same way. If NPs were to be considered non-cyclic within Chomsky's framework, then Subjacency would no longer capture *any* of the phenomena captured by the CNPC.)

Woods' Hold-list Mechanism - A Comparison

To conclude this discussion, I will compare the approach taken here with the hold-list mechanism of Woods' ATN [Woods 72; Kaplan 72]. The hold-list is a push-down stack appended to the basic ATN interpreter that holds wh-phrases for which matching gaps have not yet been found. The ATN interpreter uses the bottom of the hold-list (thinking of the stack as growing downwards) to fill a hypothesized gap whenever it traverses a special sort of arc called a "virtual NP arc" in the grammar. A virtual NP arc returns the NP at the bottom of the hold-list just as if such this NP had just been constructed by the ATN interpreter from the lexical items after the input pointer. Thus, a grammar designer for an ATN can place a virtual NP arc into the grammar wherever there is the possibility of a gap in a wh-clause, and the ATN mechanism will non-deterministically explore the implications of each choice for a given wh-clause. After an NP at the bottom of the hold-list is utilized in this way, it is popped from the hold-list.

Something like this mechanism could be grafted onto PARSIFAL, perhaps pushing pointers to Wh-comps onto this hold-list which could then be used to appropriately bind a trace whenever the corresponding gap was discovered by the parser. Since the bottom of the hold-list is accessible anywhere in the grammar, Woods' formulation also solves the inaccessibility problem described above. Because the parser is deterministic, of course, something less hypothesis driven would necessarily replace the "virtual NP arc" notion, but this is irrelevant to the current discussion. (This issue is discussed at length in Chapter 9.)

The principal objection to such an approach is simply that the addition of such a powerful additional mechanism is unnecessary within the context of PARSIFAL. As noted above, the approach taken here requires very little in the way of additional mechanisms, and certainly nothing as complex as an additional push-down stack. The hold-list/virtual-NP-arc mechanism represents a fairly major extension of the ATN framework, and it is a point in favor of the framework adopted here that the same phenomena can be handled with only an extension to allow access to the Wh-comp of the next higher S. It should also be noted that the ATN mechanism, as implemented by Woods, freely allows violations of the CNPC, and thus has no theoretical implications in so far as this phenomena is concerned.

CHAPTER 8

PARSING NOUN PHRASES - EXTENDING THE GRAMMAR INTERPRETER

Introduction

In the preceding chapters, it was assumed that noun phrases came into the buffer parsed by some mechanism which was transparent to the "clause-level" rules which were the focus of those chapters. In this section that assumption will be revoked, and extensions to the basic grammar interpreter will be introduced that allow for the parsing of noun phrases. I will also make clear why these extensions are necessary.

The grammar interpreter will be extended in this chapter by introducing a new class of grammar rules which I will call *Attention Shifting rules* (AS rules). (Rules which are not AS rules, i.e. all the rules introduced in previous chapters, I will henceforth call *Normal rules*.) Attention shifting rules, as the name implies, allow the grammar interpreter to "shift its attention" from the first constituent in the buffer to some later constituent, if there is evidence that the later constituent initiates a higher-level constituent of a given sort. Most typically in the current grammar, this mechanism is used to initiate NPs, although other sorts of constituents can be initiated using AS rules, as we shall see. The parser can then immediately construct the detected constituent, and then "shift its attention" back to the beginning of the buffer, with the newly parsed constituent now

in the buffer, providing contextual clues as to how the constituents in preceding buffer cells should be utilized.

Attention Shifting Rules - Motivation

In the grammar for English discussed in previous chapters, the two rules shown in Figure 8.1 below were introduced to recognize and flag simple declarative sentences and yes-no questions. The rule MAJOR-DECL-S will label the current active node a declarative if an NP followed by a verb appears in the buffer, while the rule YES-NO-Q will label the current active node a yes-no question if an auxiliary verb followed by an NP appears in the buffer. It seems plausible that two rules much like these rules must be included in any grammar of English.

```
{RULE MAJOR-DECL-S IN SS-START
[=np] [=verb] -->
Label c s, decl, major.
Deactivate ss-start. Activate parse-subj.}

{RULE YES-NO-Q IN SS-START
[=auxverb] [=np] -->
Label c s, quest, ynquest, major.
Deactivate ss-start. Activate parse-subj.}
```

Figure 8.1 - These two rules treat NPs as primitive.

Note that the patterns of both of these rules will trigger only if there is an NP somewhere in the buffer and thus, in a sense, both patterns treat NPs as "primitive" constituents. Yet while the other constituents in these patterns are lexical items, and are in this sense truly primitive (morphology being done before the parser is called), NPs are much larger constituents that must be constructed before they can be used by other processes.

As will now be demonstrated, it is impossible to formulate a grammar which constructs these NPs at the proper time in other than an *ad hoc* manner.

In the case of MAJOR-DECL-S, which triggers on an NP filling the first buffer position, formulating a set of grammar rules which constructs the NP is quite easy. A rule like START-NP' in Figure 8.2 below can be added to the packet CPOOL, the packet which is (nearly) always active. This rule will trigger if any word which can start an NP appears in the first buffer position. (The feature *npstart* is given to all determiners, adjectives, nouns, etc., i.e. to members of all word classes that can start NPs.) This rule, once activated, creates an NP node and activates the appropriate packet to initiate building the NP. When the NP is finished, it will be dropped into the buffer, at which point the rule MAJOR-DECL-S will trigger if the NP is followed by a verb.

```
{RULE START-NP' IN CPOOL
[=npstart] -->
Create a new np node.
If 1st is det then activate parse-det
        else activate parse-qp-1.
Activate npool.}
```

Figure 8.2 - A proposed rule for initiating NPs.

Note that this formulation depends crucially on the fact that it is possible to detect the "leading edge" of an NP, and thus initiate its construction, with no prior expectation of the presence of that NP. (This is also true of all the refinements of this formulation which will be discussed

below.) To say the same thing a little differently, the construction of an NP can be initiated in a totally data-directed manner (totally data-directed in that the packet containing this rule is always active) because the leading edge of an NP is so clearly marked syntactically. Thus, not only is it extremely useful to treat NPs as "primitive" constituents at the clause level, it is also quite possible, given that these constituents can be initiated in a data-driven manner.

While the rule START-NP' works fine for NPs whose first constituent appears in the first buffer cell, this rule will not work at all in the case of an NP which is to serve as the subject of a yes-no question or, more generally, in the case of any NP which must be constructed before its "leading edge" reaches the first buffer position. The problem is simply that the rule START-NP only examines the constituent in the first buffer position, and thus will not trigger if the NP's leading edge appears in the second or third buffer cell.

One possible solution to this problem might be to trigger a rule like YES-NO-Q on any word in the second buffer position that can initiate an NP, i.e. to re-write the rule YES-NO-Q with a pattern like that shown in Figure 8.3:

{RULE YES-NO-Q' IN SS-START
[*auxverb] *[*npstart]* --> .....}

Figure 8.3 - A possible alternative to YES-NO-Q.

The problem with this patch, as well as with the first formulation, is simply that many words that *can* initiate NPs often don't and that many words that rarely initiate NPs sometimes do, as was discussed in Chapter 1. For example, the word "five" usually initiates an NP, as in the example shown in Figure 8.4.a below, but sometimes it simply doesn't, as in 8.4.b. Similarly, while the word "as" can start an NP, as shown in 8.4.c, it can also start a subordinate clause equally well, as in 8.4.d.

(a) There are *five good reasons* for not even trying.
(b) Theses get written *five times more slowly than I thought possible.*
(c) *As many as ten* different explanations were proffered.
(d) *As these examples show*, "As" initiates various constituents.

Figure 8.4 - One word doesn't determine constituent type.

Another possible solution to this problem might be to have a second rule much like START-NP' except that it would trigger explicitly on the constituent in the *second* buffer cell. (Such a rule would have a pattern like

[t] [=npstart]

which would simply accept anything at all in the first buffer cell.) This rule (call it START-NP") would then activate packets of rules identical to the rules activated by START-NP' except that these alternate rules would carefully ignore the contents of the first buffer position. In a sense, these rules would treat the buffer as if it began with the second buffer cell. While this solution is obviously *ad hoc*, it could in fact be made to work. Furthermore, in many ways it seems to be on the right track.

The solution to this problem which I will adopt is in a sense a

variant of this last suggestion. The problem will be solved by introducing a rule called START-NP, shown in Figure 8.5 below, which is identical to START-NP' except that is flagged (by writing it as an "AS RULE") as a special sort of rule which I will call an *Attention Shifting rule*. Attention shifting rules are rules that can cause the grammar interpreter to shift the parser's "attention" from the actual start of the buffer to some later buffer cell. When such a rule is envoked, the grammar interpreter shifts the effective start of the buffer by creating a "virtual buffer start" that later rules will see as the beginning of the buffer. Once the constituent initiated by the attention shift is completed, it can be dropped into the buffer and the virtual buffer start can be discarded by the grammar interpreter. The parsing process will then continue as if that constituent had simply appeared in the buffer fully formed, justifying the assumption made in previous chapters. (The problems raised by the sentences in Figure 8.4 above can in fact also be solved within this framework. I will outline a common solution to this set of problems later in the chapter.)

```
{AS RULE START-NP IN CPOOL
[=npstart] -->
Create a new np node.
If 1st is det then activate parse-det
          else activate parse-qp-1.
Activate npool.}
```

Figure 8.5 - The final version of the NP-initiating rule.

While this mechanism may solve the problem of initiating NPs, it raises several new questions. Among them are:

-How is this new sort of rule to be matched against the buffer?

-What sort of mechanism can be used to "shift" the buffer start, and

how can the buffer later be shifted back to its original state?

-Can the length of the buffer still be limited to only three buffer cells if the effective start of the buffer might conceivably shift to the third buffer position?

These questions will be answered in the following several sections.

Attention Shifting - Implementation

In this section I will outline the implementation of the attention shifting mechanism in terms of the virtual buffer mechanism developed in chapter 3. This implementation can be easily accomplished by using the extended buffer mechanism with offset which was introduced in that chapter, using the offset mechanism to implement the "attention shift".

To refresh the reader's memory, repeated below is the key portion of the description of the offset mechanism:

> [The offset mechanism causes] the index given to the Read, Insert, and Delete commands to refer to a cell whose location is offset from the beginning of the buffer by some fixed amount. This offset is specified by a new command Offset(j), which causes the offset parameter *m* to be set to the sum of j and the previous value of m. (The offset parameter m is initially stipulated to be 0.) After a command of this form has been executed, the command Read(i), for example, will return the item in cell x[i+m] rather than the item in cell x[i].
>
> The offset mechanism is in fact coupled to a *push down stack* of such offsets, so that offsets may be pushed and popped from the stack. This stack, the third data structure of the full parser model, is referred to as the *buffer pointer stack*, or just as the *pointer stack*. The current offset parameter m is thus the offset at the bottom of the buffer pointer stack, again thinking of the stack as growing downwards. This means that two new commands are needed for this extended buffer mechanism:
>
> Offset(j): Push the sum of j and m_{old}, the current value of the bottom of the pointer stack, onto the pointer stack, making

m_{new} the new offset.

Pop-offset: Pop the current value off the pointer stack, making the previous value pushed onto the stack the current offset. If the top of the stack has already been reached, executing this command causes an error.

By stipulating that the stack is initialized with an offset parameter of 0, this extended mechanism will behave exactly like the unextended buffer mechanism as long as no Offset commands are executed.

Note that the Read, Insert, and Delete commands are constrained even in this extended buffer model to take only positive buffer indices. Thus, if the current offset is m, there is no way to read, insert or delete any item in cells x[i], for $i \leq m$. Furthermore, since no mechanism is provided to examine the pointer stack itself, there is no way to even note whether there is any offset at all, e.g. whether the command Read(1) will return the item in x[1] or some other cell. Because it will often be useful to refer to the effective beginning of the buffer, i.e. the cell whose contents are returned by the command Read(1), I will often use the terms *the effective buffer start* to refer to this cell. When discussing the extended buffer model, this cell will be referred to as "the first buffer cell", rather than the absolute first cell in the buffer, and in general, "the *i*th buffer cell" refers to the cell whose contents are returned by the buffer command Read(i) rather than the cell x[i].

Given this offset mechanism, attention shifting is conceptually implemented in the grammar interpreter as follows (the actual implementation is less straightforward and considerably more efficient, although equivalent in behavior):

Each attention shifting rule, flagged in PIDGIN by an initial "(AS RULE .....}", has a pattern which consists of exactly one node description. Each time the grammar interpreter attempts to match the pattern of any normal rule, before it attempts to match each node description in the pattern against the relevant constituent in the buffer, it checks to see if the pattern of any AS rule in any active packet matches against that node. (AS rules are

only matched against constituents in the buffer, not against either of the accessible nodes in the active node stack.) If at least one AS rule does match against a node, then the grammar interpreter does the following: If the constituent that triggered the AS rule was in the *n*th buffer cell, the grammar interpreter first executes the buffer command Offset(n). This shifts the effective buffer start to the cell that was previously the nth cell in the buffer, i.e. the constituent that was previously Nth is now 1st. The grammar interpreter then aborts the normal rule matching process and runs the AS rule with the highest priority of those AS rules that matched.

(The grammar interpreter checks each pattern against the constituents in the buffer in left-to-right order, so that if two constituents in the buffer will potentially trigger active AS rules, it will be the leftmost constituent of these two that will first trigger an AS rule.)

As mentioned above, after the buffer start has been shifted, those constituents before the effective buffer start will be invisible to succeeding normal rules. These rules will view the constituent in the buffer cell pointed to by the new bottom of the buffer pointer stack as 1st, the constituent in the cell after it as 2nd, and the constituent in the cell after 2nd as 3rd. There is absolutely no way in which succeeding rules can access the constituents in cells to the left of the buffer pointer as long as that buffer pointer is on the bottom of the stack, or, for that matter, in the stack, since (as the reader may have observed) the Offset mechanism can only push increasingly large offsets onto the stack. Initially, of course, the effective buffer start is identical with the contents of buffer cell x[1].

After a constituent that was being built during an attention shift is completed and has been dropped from the stack into the buffer by some grammar rule, that rule should "dismiss" the attention shift by popping the current buffer pointer from the bottom of the buffer pointer stack. This is done by the PIDGIN code

Restore the buffer.

Executing this command will restore the effective buffer start to its previous location by popping the buffer stack, returning the previous buffer pointer to the bottom of the stack. After the attention shift has been dismissed, the grammar interpreter will again match rules against the previous effective buffer start and the constituents that follow it. Note that it is imperative that the newly constructed node be dropped into the buffer before the buffer pointer stack is popped. Since a node dropped from the active node stack into the buffer is inserted into the buffer by executing the buffer command Insert (c,1), the node will be inserted into the cell which is the current effective buffer start. If the node is dropped into the buffer after the buffer pointer stack is popped, it will not be inserted into the buffer at the proper location.

One might think that rather than attempting to match AS rules against buffer constituents each time that the grammar interpreter attempts to do "normal rule" pattern matching, that AS rules could be matched only once against each constituent in the buffer, say, when it first enters the buffer. This would be sufficient except for the fact that grammar rules are continually activating and deactivating packets of rules. Thus, when a given

constituent first enters the buffer, no AS rule in the packets that are active at that time might match. The next grammar rule that is executed just might activate a packet which does contain an AS rule that will match against that constituent, however. Thus, it is necessary to constantly check to see whether AS rules in any active packets match against each constituent.

The reader might also wonder why it is necessary to allow for a stack of buffer pointers, rather than a single shifting pointer. As it turns out, it is often necessary to allow attention shifts within attention shifts, thus making it necessary to allow for a stack of buffer pointers. For example, NPs must be constructed by an attention shifting process, as was discussed in the first section of this chapter. Within NPs, however, it is necessary to build number phrases (e.g. "four hundred thirty-five") by an attention shifting process, as will be demonstrated in a later section. Thus, there is need for an attention shift to build the number phrase within the attention shift to build the NP.

It should be pointed out that often the process of attention shifting results in only a logical shift of the effective buffer start. This will occur whenever an attention shifting rule triggers on a constituent that fills the current first buffer position. In this case, a new pointer will be pushed onto the buffer pointer stack, but the new pointer will point to the same buffer position as the pointer now immediately below it on the stack.

The ATN PUSH Arc - A Comparison

Before turning to some examples that demonstrate the operation of the Attention Shifting mechanism, let me digress to compare the operation of the AS mechanism with the PUSH arc mechanism of Woods' ATN. This comparison will make clear exactly where the AS mechanism differs from this standard method for forming subordinate constituents. (As Winograd points out [Winograd 71], his PARSE function is essentially equivalent to the PUSH arc, so here, as in Chapter 2, I will use the ATN as an exemplar of a larger class of parsers.)

As a first approximation, the ATN PUSH arc can be thought of as a simple recursive function call. A PUSH arc in an ATN grammar tells the parser to push to a specific subgrammar, which is to produce a constituent of a given type. If the subgrammar succeeds, it will return such a constituent, which can then be utilized by the higher level grammar. In essence, this amounts to calling a subroutine which returns as its value a constituent of a given sort. Since a network A can contain PUSHes to a network B that in turn pushes to A, the whole mechanism is recursive. (For example, the NP network pushes to the S network, which contains PUSHes to the NP network.)

This first approximation must be slightly modified - at least in focus - in that the ATN operates by simulating a nondeterministic mechanism. Thus, the execution of a given PUSH arc may or may not succeed in that a failure from the lower network may propagate back through the PUSH arc and cause the arc to fail. In light of this, and as stressed in Chapter 2, a

PUSH arc must be viewed as an encoding of a hypothesis that a constituent of a given sort can be found at a particular point in the input string. For example, an ATN grammar will include a PUSH to the NP network, say, at a given point in the grammar if it is reasonable, under some circumstance or other, to hypothesize the existence of an NP at this point in the input string. It should be stressed that this hypothesis is dependent only on that part of the input string which the parser has fully analyzed; it is not preconditioned upon any properties of what follows in the input string; the PUSH arc is an entirely top-down mechanism.

On the other hand, an AS rule is entirely *data-driven*, i.e. entirely bottom-up. Almost all AS rules are in packets that are always active while a constituent of a given sort is being parsed (typically, either an S or an NP); the rule START-NP of the previous section, for example, is always active when an S node or a VP node is the current active node. Thus, there is little top-down control of the AS rule mechanism. Instead, an AS rule is triggered on the basis of features of the next word (or, more generally,constituent) in the input string which will mark (if the AS rule is a correct rule of grammar) the "leading edge" of a given sort of constituent. To say the same thing in a different way, an AS rule is triggered almost entirely on the basis of bottom-up evidence for the presence of a constituent of a given sort.

To make this point clear, it is worth noting that the activation of an AS rule has much more in common with the computer science notion of an *interrupt* than it does with the notion of a *function call*. The triggering

of an AS rule is like an interrupt in that both the triggering of the rule and all processing that is done before the buffer pointer is restored is transparent to "normal" processing. Also like an interrupt, a special mechanism is required to shift the processor's "attention" from the previous task and then, when appropriate, shift the processor's "attention" back, all without the knowledge of the background task. While the fact that attention shifts can be recursively nested might lead one to believe that something similar to recursive function calls is going on, the notion of *nested interrupts* is a better model.

An Example of Attention Shifting

An example should help to make this attention shifting mechanism a little more clear. Let us see how the initial segment of the sentence shown as 8.6 below is constructed by the parser, focusing on the processes by which the NP "a meeting" is constructed:

8.6 Is a meeting scheduled for Monday?

The example will begin after the rule INITIAL-RULE has been executed, leaving the parser in the state depicted in Figure 8.7 below, with the packet SS-START active. This state is exactly equivalent to that depicted in Figure 4.17, so there is no need here for further comment on the action of the rule INITIAL-RULE. Several changes to the format of parser state snapshots in this and all following figures should be noted, however. First of all, the pointer stack is depicted as a list of absolute buffer indices, with the bottom of the list at the extreme left. The effective buffer start, the buffer cell whose absolute index is the bottom of the pointer list, is also

flagged by the symbol "=====" in the line immediately before it. To say the same thing in another way, the constituent shown in the line following the symbol "=====" is 1st.

The Active Node Stack (0. deep)

C: S22 (S) / (CPOOL SS-START)

The Buffer (Pointer stack: (1))
=====

Yet unseen words: is a meeting scheduled for monday ?

Figure 8.7 - The example begins after INITIAL-RULE has been run.

After the rule INITIAL-RULE has been executed, the grammar interpreter begins the pattern matching process. The grammar interpreter checks to see whether the pattern of any normal rule in an active packet matches against the constituents in the buffer and the active node stack, starting with the rules of highest priority. Among the rules in the packet SS-START, which is currently active, is the rule YES-NO-Q, with the pattern

[=auxverb] [=np].

Let us assume that the grammar interpreter begins the pattern matching process by attempting to match this rule against the contents of the buffer. Because there is no constituent in the first buffer cell, the grammar interpreter inserts the first word from the input list, the word "is", into the buffer. Before attempting to match the first node description of the pattern of YES-NO-Q against 1st, the grammar interpreter checks to see if this constituent fulfills the pattern of any active AS rule, which, let us assume, is not the case. The grammar interpreter now discovers that the first description in YES-NO-Q's pattern is fulfilled by 1st, so the interpreter moves

on to the next node description in the pattern of YES-NO-Q. The second buffer cell is also empty, so another word from the input list is inserted into this cell. The state of the parser at this point is depicted in Figure 8.8. Note that the numbers to the far left of each buffer entry is the *absolute* index of the corresponding buffer cell.

The Active Node Stack (0. deep)

C: S22 (S) / (CPOOL SS-START)

The Buffer (Pointer stack: (1))

=====

1 : WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
2 : WORD156 (*A NPSTART DET NS N3P INDEF) : (a)

Yet unseen words: meeting scheduled for monday ?

Figure 8.8 - During the process of matching the pattern of YES-NO-Q.

Again, before the interpreter checks to see if the next node description in the pattern of YES-NO-Q is fulfilled by the corresponding constituent in the buffer, in this case *2nd*, it checks to see if any AS rule is triggered by that constituent. This time the interpreter discovers that an AS rule is triggered, the word "a" fulfilling the pattern of the rule START-NP presented above. Because this constituent triggers an AS rule, the buffer interpreter shifts its attention to the cell it occupies by executing the buffer command Offset (2), since the constituent that triggered the AS rule was in the 2nd buffer position relative to the effective buffer start. This state is depicted in Figure 8.9.

```
About to run: STARTNP

          The Active Node Stack ( 0. deep)

C:        S22 (S) / (CPOOL SS-START)

          The Buffer  (Pointer stack: (2 1) )

1 :       WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
          =====
2 :       WORD156 (*A NPSTART DET NS N3P INDEF) : (a)

Yet unseen words:  meeting scheduled for monday ?
```

Figure 8.9 - After the rule STARTNP has been triggered, the grammar interpreter shifts the effective buffer start.

After the effective buffer start is shifted, the action of rule STARTNP is run. Besides creating an NP node, NP40 in Figure 8.10 below, this rule activates the packets PARSE-DET, a packet of rules that initiate determiner structures, and NPOOL (Noun phrase POOL), which contains rules that are always to be active whenever an NP is being parsed. The state of the parser after this rule is executed is shown in Figure 8.10.

```
          The Active Node Stack ( 1. deep)

            S22 (S) / (CPOOL SS-START)
C:          NP40 (NP) / (NPOOL PARSE-DET)

          The Buffer  (Pointer stack: (2 1) )

1 :       WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
          =====
2 :       WORD156 (*A NPSTART DET NS N3P INDEF) : (a)

Yet unseen words:  meeting scheduled for monday ?
```

Figure 8.10 - After rule START-NP has been executed.

The interpreter now continues matching rules against the contents of

the buffer, beginning with the effective buffer start, which is now the second absolute buffer position. The pattern of the rule DETERMINER in the active packet PARSE-DET now matches against 1st and the action of this rule is run. DETERMINER is shown in Figure 8.11 below. This rule attaches the determiner to the NP, adds some features to the NP, and switches on the packet PARSE-NOUN after deactivating PARSE-DET. The interpreter now notices that the determiner "a" has been attached, and removes it from the buffer.

```
{RULE DETERMINER IN PARSE-DET
[=det] -->
Attach 1st to c as det.
Label c det.
Transfer the features indef, def, wh from 1st to c.
Deactivate parse-det.
Activate %for the sake of the example% parse-noun.}
```

Figure 8.11 - The rule DETERMINER in the packet PARSE-DET.

The rule NOUN matches next, after WORD157 "meeting", the next word in the input list, enters the buffer. The state of the parser after NOUN has matched, but before its action is run, is depicted in Figure 8.12 below. Note that WORD155, "is", remains invisible before the effective buffer start, totally unaffected by all of this.

```
About to run: NOUN

          The Active Node Stack ( 1. deep)

            S22 (S) / (CPOOL SS-START)
C:        NP40 (INDEF DET NP) / (PARSE-NOUN NPOOL)
                    DET : (a)

          The Buffer   (Pointer stack: (2 1) )

1 :       WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
          =====
2 :       WORD157 (*MEETING NPSTART NOUN NS) : (meeting)

Yet unseen words:  scheduled for monday ?
```

```
{RULE NOUN IN PARSE-NOUN
[=noun] -->
%Simplified for the sake of this example.%
Attach a new nbar node to c as nbar.
%Note that c is now the nbar node.%
Attach 1st to c as noun.
Transfer the features massn, time, ns, npl from 1st to c.
Drop c %i.e. the nbar node%.
Drop c %i.e. the np node%.
Restore the buffer.}
```

Figure 8.12 - Before the rule NOUN is executed.

The rule NOUN first attaches an NBAR node to the NP, then attaches the noun "meeting" to the NBAR. (The reason for this NBAR node is as follows: if we look at an NP as being analogous to an S node at some level, and a noun as similarly analogous to a verb, then we need a node that serves as a "verb phrase" node to complete the analogy. The NBAR node serves this function. This follows from the so-called "X-bar convention", first suggested by Chomsky [Chomsky 70], and well explained in [Jackendoff 74].) The NBAR node is then popped from the active node stack, and then the NP node is popped. Since the NP node is not attached to any larger node, it is

dropped into the buffer, where it is inserted, by definition, before 1st, i.e. it is inserted at the *effective* beginning of the buffer. The state of the parser at this point is shown in Figure 8.13.a. Finally, the buffer pointer is restored to its earlier state with the execution of the PIDGIN code "Restore the buffer", which pops the buffer pointer stack. The state of the buffer after the NP is dropped and the pointer stack is popped is shown in Figure 8.13.b.

```
          The Active Node Stack ( 0. deep)

C:        S22 (S) / (CPOOL SS-START)

          The Buffer  (Pointer stack: (2 1) )

1 :       WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
          =====
2 :       NP40 (MODIBLE NS INDEF DET NP) : (a meeting)

Yet unseen words:  scheduled for monday ?
```

(a) - After the finished NP is dropped into the buffer.

```
          The Buffer  (Pointer stack: (1) )
          =====
1 :       WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
2 :       NP40 (MODIBLE NS INDEF DET NP) : (a meeting)
```

(b) - The buffer after the pointer stack has been popped.

Figure 8.13 - Dismissing the attention shift.

Now NP40, the new NP, appears in the buffer as a complete constituent. From the point of view of normal rules, the NP will seem to have simply appeared in the buffer as if it were in the input list already parsed. The interpreter will now once again match normal rules against the absolute beginning of the buffer, since the absolute beginning is now also the effective buffer start.

The interpreter now begins to match the rule YES-NO-Q against the constituent buffer once again. Once again, the grammar interpreter checks to see whether 1st triggers any AS rules before checking to see that it fulfills the first node description in the pattern of YES-NO-Q. After no AS rules are triggered by 1st and the corresponding node description is fulfilled, the interpreter checks to see that 2nd, now the NP NP40, does not trigger an AS rule before checking to see that the NP fulfills the second node description in the pattern of the rule YES-NO-Q.

The pattern of this rule will now match, since the constituents which are 1st and 2nd now fulfill the two descriptions of the pattern of YES-NO-Q. The action of this rule will then be run, labelling C with features that indicate that it is a yes/no question, resulting in the state shown in Figure 8.14. Since normal rules will now continue the parse for a while, the example will be terminated at this point.

The Active Node Stack (0. deep)

C: S22 (S QUEST YNQUEST MAJOR) / (CPOOL PARSE-SUBJ)

The Buffer (Pointer stack: (1))

=====

1 : WORD155 (*BE VERB AUXVERB PRES V3S) : (is)
2 : NP40 (MODIBLE NS INDEF DET NP) : (a meeting)

Yet unseen words: scheduled for monday ?

Figure 8.14 - After the rule YES-NO-Q has been run.

Another Example of Attention Shifting

As another example of attention shifting, this section will investigate the question of how a deterministic grammar can distinguish between the wide range of constituents that start with the word "as". This problem has been used in several previous sections as evidence against any scheme that attempts to use one word lookahead to indicate what constituent begins next. Up to this time, however, the grammar mechanism was not sufficiently developed to allow discussion as to how this problem can be solved within the context of PARSIFAL. As we shall see, the solution is quite simple and natural given the attention shifting mechanism. This section shall show that once again, a seemingly non-deterministic process has a totally deterministic solution.

First of all, note that the type of a constituent that begins with "as" can be determined, by and large, by examining the two constituents after that word. Consider the sentences in Figure 8.15 below (which appeared earlier as Figure 2.2):

(a) *As these examples show*, "As" initiates various constituents.
(b) *As many as ten* different explanations were proffered.
(c) No one could ever be *as big as big bad John.*
(d) He left *as quickly as he could.*
(e) Bill offered his advice *as an expert in such matters.*
(f) Who could believe he caught *as big a fish as that?*

Figure 8.15 - "As" can initiate many different constituent types.

If the constituent after "as" is a quantifier, an adjective, or an adverb, each followed by "as", as in 8.15 b, c, and d, then the entire phrase will be a phrase of that type. (The relevant diagnostic for each of the examples given is summarized schematically in Figure 8.16 below; the lettering of the diagnostics corresponds to the lettering of the examples above.) If the constituent after "as" is an NP, as in 8.15 a and e, then the entire entity will be a prepositional phrase, with "as" serving as a preposition. (Note that I treat most subordinate clauses initiated by "subordinators" such as "but", "when", "after" as prepositional phrases with clauses as objects. Thus, although these two sentences differ in that the NP following "as" serves as prepositional object in 8.15.e, but as the subject of a clause which is itself the prepositional object in 8.15.a, they have the same higher-level structure. Other rules can then distinguish between these two subcases.) Finally, if "as" is followed by an adjective which is followed by an indefinite singular NP, as in 8.15.f, then the entire construction will be an NP. The NP will also be followed by the word "as", but there is no need to see this fourth constituent to diagnose the correct structure.

```
(a,e)  [=*as]  [=np]                              —> PP
(b)    [=*as]  [=quant]  [=*as]                   —> QUANT
(c)    [=*as]  [=adj]    [=*as]                   —> ADJ
(d)    [=*as]  [=adv]    [=*as]                   —> ADV
(f)    [=*as]  [=adj]    [=*np, indef, ns]        —> NP
```

Figure 8.16 - diagnosing phrases which begin with "as".

Now note that if a phrase initiated by "as" is part of an NP, then it either initiates the NP or else it immediately follows the head noun of the NP. The examples of Figure 8.17 below are illustrative:

(a) three boys as big as Herbert
(b)*three as big as Herbert boys
(c) as many as three hundred boys
(d)*the as many as three hundred boys

Figure 8.17 - A phrase beginning with "as" can only appear in NPs initially or immediately after the head noun.

These facts lead to the following solution to the problem of "as": An AS rule should be added to the grammar that triggers on "as" and placed in the packet CPOOL which is always active during "clause-level" processing. After this rule has triggered, it should activate a packet of rules each of which is sensitive to one of the types of constituents that can begin with "as", as shown above. The correct structures can be initiated by these rules, completed by packets which they in turn activate, and dropped into the buffer. (I ignore here - and have no solution for - the problem of how to parse the sorts of highly ellided structures that often follow "as"; e.g. "Sam had written as many words this year as John books last year." (see [Bresnan 73]).) If the completed structure itself initiates an NP, for instance, the AS rule that begins NPs will then be triggered by this new constituent. Thus, the grammar can construct phrases beginning with "as" in an entirely data-driven (i.e. bottom-up) manner, and these completed phrases can then trigger still higher level constituents in a data-driven manner if (and only if) that is appropriate.

Why There Must be a Stack of Buffer Pointers

A good test for whether a constituent should be initiated by an attention shifting rule seems to be whether or not that constituent must be constructed before it reaches the front of the constituent buffer; i.e. whether or not there is a grammar rule that depends on the existence of that constituent at some position in the buffer other than 1st. Thus, the original motivation for attention shifting rules was the fact that the subject of a yes-no question must be constructed before it can reach the front of the buffer, i.e. that it must be initiated while its "leading edge" is still in the second buffer cell.

In this section I will use this test to show that attention shifts must be allowed within attention shifts.

Consider the following two noun phrases:

8.18.a a hundred boys
8.18.b a hundred pound rock

In 8.18.a, note that "a" is not the determiner of the NP, but rather part of the number phrase (NUMP) "a hundred", while in 8.18.b "a" serves as the determiner of the NP and "hundred" is part of the adjective phrase (ADJP) "hundred pound". The structures of these phrases are shown in Figure 8.19 below (suppressing unimportant detail).

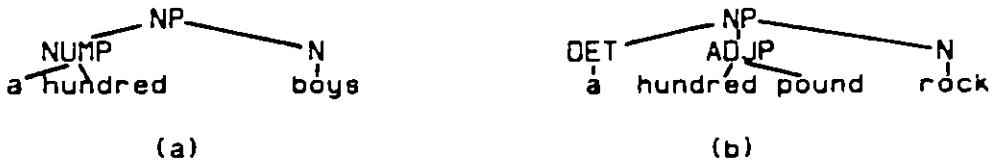


Figure 8.19 - The word "a" can serve in two very different roles.

Now note that the following test will decide whether or not the determiner "a", if followed by a number like "hundred" or "thousand" (which I will call "big number words" or "bignums"), serves as a determiner or as part of a NUMP: If the word following the number is a singular "measure" word like "pound", "ton", "foot", etc., as in 8.19.b above, then the number serves as part of the ADJP, and the determiner serves as the determiner of the NP. (The rule pattern which would recognize this structure is

[="a] [=bignum] [=noun, ns, measure].)

If the word following the number is not a singular measure word, as in 8.19.a, the determiner and the number form a number phrase. The crucial point here is that it is necessary to examine the third constituent in the buffer, assuming that "a" is 1st, to determine the role of 1st. (I will come back to this example in Chapter 9 in which the general subject of diagnostics of this type will be discussed.)

Now note that this diagnostic must be applied not only to single number words, but also to entire number phrases that begin with "big number words". Thus, 8.20.a and 8.20.b below are parallel to 8.18.a and 8.18.b above:

8.20.a a hundred and thirty-seven boys
8.20.b a hundred and thirty-seven pound rock

This provides evidence that the NUMP "hundred and thirty-seven" must be constructed before the role of the preceding determiner can be established. Since the NUMP must be constructed before its leading constituents reach the front of the buffer, such number phrases must be initiated by an attention shifting rule. However, the grammar interpreter will only notice the presence of the "leading edge" of this structure after it has initiated the NP by which it is dominated. Since this NP itself will have been initiated by an attention shifting rule, the grammar interpreter must allow for nested attention shifts.

AS Rules and the Length of the Buffer

As the reader will have realized, the buffer shifting mechanism forces some increase in the total length of the constituent buffer beyond the three buffered constituents I have previously assumed. If all normal rules can access up to three constituents, and if the buffer pointer may be shifted to the second or third absolute location in the buffer, then the buffer must necessarily be longer than three constituents. This leaves the question: To what length must the buffer be allowed to grow if a grammar utilizes the attention shifting mechanism?

While the buffer could be allowed to grow to arbitrary lengths, it seems empirically that a buffer of only five (or so) constituents is quite adequate for parsing almost all of English. (While I will not pursue the matter here, it seems to be the case that those sentences which *do* require more than five simultaneously buffered constituents fall into the class of

"center embedded sentences" which are well known to cause serious perceptual difficulties for speakers of English. (An example of such a sentence is "The rat the cat the dog chased bit ate the cheese.".) A promising area for future research would be to investigate whether this model can account for the full range of performance phenomena that fall under the rubric of center embedding problems.)

Below is a brief plausibility argument that an absolute buffer length of five constituents is sufficient for English, given that no grammar rule can access more than three constituents. It is intended only to give the reader a feel for the empirical considerations that lead me to believe that a total buffer length of five is sufficient.

The purpose of attention shifting, fundamentally, is to allow "lower level" constituents to be constructed in a manner which is transparent to whatever "higher level" constituent is presently under construction. Thus, given that the pattern of a grammar rule may be made up of three node descriptions, each of which may match against a constituent of a type that must be initiated by an attention shifting rule, the grammar interpreter must provide enough space in the buffer to allow up to three consecutive constituents to be constructed by attention shifting processes.

If it were the case that there was no need for nested attention shifts, then a buffer of exactly five constituents would be adequate: The worst case will be when the constituent starting in the third buffer position must be initiated by an attention shifting rule; after the attention shift, the

third buffer cell from the absolute beginning of the buffer is 1st, and the fifth buffer cell is 3rd. This is shown below in Figure 8.21.

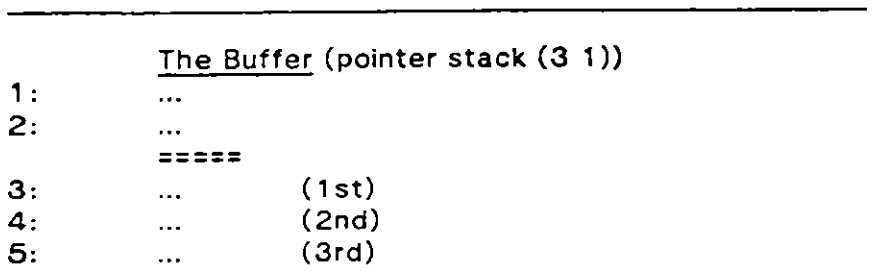


Figure 8.21 - A buffer of 5 cells is sufficient if only one attention shift at a time is possible

As we have seen, however, there is call for nested attention shifts. Thus, it might seem that a five cell buffer is simply not adequate. Two empirical observations, however, taken together, imply that a buffer of length five is in fact adequate. These two observations are that: I have yet to find either 1) a grammar rule whose pattern matches against a constituent in the 3rd buffer cell that must be initiated by an attention shifting rule, or 2) a triply-nested attention shift where all three attention shifts actually do shift the buffer pointer. (Sometimes an attention shift will be triggered by the constituent already in the 1st buffer cell. Such an attention shift will not physically change the effective buffer start.) If there are at most two attention shifts that actually move the effective buffer start, and if both of these attention shifts move the effective buffer start to the constituent that was in the 2nd (relative) buffer position before the shift, then a buffer length of five is adequate. After the first shift to the second position, the absolute second position is 1st and the absolute third position is 2nd, as shown in Figure 8.22.a below. Shifting again to the relative second position shifts the effective buffer start to the absolute third position, and again the

fifth absolute position is 3rd, as shown in 8.22.b. While situations that might require larger buffers than this could arise in principle, the situation discussed here seems to be the worst situation that arises in practice.

The Buffer (pointer stack (2 1))

```
1:      ...
        =====
2:      ...     (1st)
3:      ...     (2nd)
```

(a) - The buffer after one shift to 2nd.

The Buffer (pointer stack (3 2 1))

```
1:      ...
2:      ...
        =====
3:      ...     (1st)
4:      ...     (2nd)
5:      ...     (3rd)
```

(b) - The buffer after two consecutive shifts to 2nd.

Figure 8.22 - A buffer length of 5 suffices for two shifts to the relative 2nd buffer position.

The reader may wonder why I bother to argue for limits on the length of the buffer at all. Clearly, the main constraint on the ability of a grammar to deterministically build proper analyses of input sentences is the limitation that no single grammar rule is allowed to examine more than three constituents. (I discount here the possibility that through clever rule encodings, using intentional attention shifts, that arbitrary lookahead could be effected.) Any limitation on the length of the buffer can thus be considered a secondary limitation on the power of the parser.

My primary reason for trying to constrain the length of the buffer

is primarily for the sake of possible future investigations into the psychological plausibility of this model. As I mentioned above, I believe that an account of center embedding phenomena which is at least initially plausible may derive from limiting the buffer to three high level constituents, and five constituents total. Furthermore, five happens to fall nicely within the range of seven plus or minus two, thus leaving open the possibility that the buffer might be equated with the psychologist's notion of short term memory. In terms of the present state of the model, however, the limitations which I impose here must be considered to be largely arbitrary, although quite adequate, it seems, for all but the most contorted English sentences.

In Conclusion

To close the discussion of attention shifting rules, let me emphasize the data-driven nature of these rules. The constituents that are initiated by AS processes, such as NPs and the various sorts of "as" phrases discussed above, can appear almost anywhere within a clause, and thus a grammar cannot be constructed that efficiently incorporates strong expectations about where such constituents may or may not appear. In both of these cases, however, we have seen that the presence of such constituents is strongly marked by the constituents that serve as their "leading edges". Thus, both of these sorts of constituents can be initiated with high accuracy in a purely data-driven manner. I have also demonstrated that by initiating these constituents bottom-up, a class of apparent exceptions to the data-driven initiation of NPs, namely the problem of "as" as an inconclusive trigger, simply disappears. Given the approach discussed above, a grammar can build

a quantifier phrase initiated by "as", for example, in a data-driven manner, and then let the completed QP trigger the NP that it certainly initiates. While "as" may initiate many different sorts of higher level structures, it is fairly certain that the resulting quantifier phrase does in fact initiate an NP.

I believe that the trade-off exhibited here between the ability to accurately predict the presence of a constituent on the basis of prior expectation and the ability to easily detect the presence of a constituent by the clear marking of its leading edge is a general feature of the grammar of English and perhaps of natural language in general. It is easy to see that this property necessarily follows from the claim that the parsing of syntactic structure must be a deterministic process. I therefore claim that the observation of clear examples of this tradeoff can be taken as evidence in support of the determinism hypothesis.

The existence of this phenomenon provides strong evidence that syntactic processing should be done by a combination of expectation- and data-driven methods. There is no good method by which a purely hypothesis-driven parser can take advantage of the grammatical properties that have been investigated in this chapter. Whether or not the "leading edge" of an NP is clearly marked, such a parser must necessarily hypothesize the existence of an NP at every point at which such a constituent could possibly occur. Given the model of syntactic processing presented in this document, the existence of such constructions with well marked "leading edges" allows analysis to proceed in an efficient data-driven manner whenever clearly indicated. Such constructions, however, are necessarily a

cause of inefficiency in a purely hypothesis-driven parsing model. In short, it is impossible for a purely hypothesis-driven parser to take advantage of these clear-cut guides to syntactic processing which are encoded in the input string itself.

CHAPTER 9

DIFFERENTIAL DIAGNOSIS AND GARDEN PATH SENTENCES

Introduction

As the observant reader has probably noted, there are sentences in English which seem to indicate that *people* don't parse English deterministically, quite independently of any claim that English *can* be parsed that way. While such sentences are not *a priori* counterexamples to the determinism hypothesis as stated in Chapter 1, they seem to call into question the validity of the determinism hypothesis as a basis for any sort of psychological modelling. In this chapter I will present some anecdotal evidence which supports speculation that such sentences are in fact the exception that proves the rule, given one small modification of the determinism hypothesis. The revised determinism hypothesis, in essence, states that any English sentence *which a native speaker analyzes without conscious effort* can be parsed deterministically.

This chapter will present a potential explanation for the difficulty caused by so-called *"garden path" sentences*, sentences which have perfectly acceptable syntactic structures, yet which many readers initially attempt to analyze as some other sort of construction, i.e. sentences which lead the reader "down the garden path". (Some examples of garden path (GP) sentences are given in Figure 9.1 below.) This theory will derive from an investigation of several situations in which a deterministic parser is presented

with an input which contains a *local structural ambiguity*. (An ambiguity is *local* if it can be resolved by examining the grammatical context provided by the entire sentence; an ambiguity is *structural* if two different structures can be built up out of a string of smaller constituents each of a given type.) To handle each of these ambiguities, a special purpose grammar rule will be presented which can *differentially diagnose* between the two structural possibilities. I will show that some ambiguities can be diagnosed with assurance by PARSIFAL's grammar rules while it seems that diagnosing other ambiguities requires a buffer "window" larger than the three constituent maximum which each rule is allowed to examine. These latter ambiguities, I suggest, are exactly those which cause garden paths.

(a) The grocery store always orders a hundred pound bags of sugar.
(b) I told the boy the dog bit Sue would help him.
(c) The horse raced past the barn fell.

Figure 9.1 - Some examples of garden path sentences.

It must be stressed that the theory which will be presented here is highly speculative, and that the evidence presented for it is either anecdotal or the result of informal experiment. Hopefully, such "armchair psycho-linguistics" will provide incentive for future research in this area which will yield more substantial evidence for or against the speculations presented below.

A Theory of Re-analyzing Garden Path Sentences

Before approaching the question of differential diagnosis and its relation to garden path sentences, it is necessary to sketch the overall

framework of a system which can "recover" from garden path sentences. The fundamental assumption of the theory which will be presented later in this chapter is that the grammar interpreter, by itself, should not be able to recover from a garden path, i.e. that the grammar interpreter, given a garden path sentence as input, should take the garden path and then become "stuck". Because this immediately raises the crucial question of how such sentences *can* be handled, I will digress to sketch a speculative theory of how this can be done in the context of a natural language understanding system which uses the parser as a subsystem. After this general theory has been presented, I will then return to the specific question of the nature of diagnostic rules and their relation to the phenomena of garden path sentences.

A deterministic parser should be viewed, I believe, as a fast, "stupid" black box. According to this view, the grammar of English is really quite simple (which doesn't imply that *finding* the correct grammar is simple), and it costs very little computationally to syntactically analyze a sentence. Because of this simplicity, such a parser is much too weak a mechanism to be able to recover by itself if it runs into difficulty and cannot fully analyze a sentence.

When functioning as part of a larger scale system, given a garden path sentence as input, a deterministic parser ideally should take the garden path and become "stuck" at the point at which people become conscious that they have been misled, for reasons which will be made clear below. (By "stuck", I mean that no grammar rule in any active packet applies.) Clearly, if this is true, then there are grammatical English sentences which *cannot* be

parsed deterministically, so the determinism hypothesis must be modified appropriately. The following modification of the determinism hypothesis is in keeping with this suggestion:

> There is enough information in the structure of natural language in general, and in English in particular, to allow left-to-right deterministic parsing of those sentences *which a native speaker can analyze without conscious effort.*

The following is a highly speculative theory of how a natural language understanding system should reanalyze those sentences which have led the parser itself down a garden path, assuming that the behavior of the parser conforms to the modified determinism hypothesis:

When the parser cannot analyze a given input, it will fail with some portion of the input analyzed in some fashion, correct or incorrect. (Note that a *non*-deterministic parser typically fails after it has searched the entire space of utterances defined by its grammar, and has arrived at the root node of the search tree with no alternatives left. When such a parser fails, it can give no partial analysis at all, only an indication that its search space has been exhausted, unless it has stored all of its attempted analyses.)

At this point, some higher level "conscious" grammatical problem solving component comes into play. This component restarts the parser on the remainder of the input after activating a special packet of rules which allow the initiation of grammatical fragments. (This packet will contain rules which encode information like "If a tensed verb which is not an auxiliary appears at the beginning of the buffer without a prior subject, then begin a sentence fragment that lacks a subject".) The parser may be

restarted once or twice, say, producing a string of two or three grammatical fragments which correspond to the original utterance. Thus, even though the parser cannot successfully analyze the input into a single coherent grammatical structure, it can be used to reduce the input into several coherent pieces which can be given to a higher level analysis process.

Once the original input has been "chunked" into fragments, the higher level problem solver uses a set of grammatical heuristics to attempt to discover where the parser initially went astray. An example of such a heuristic is given in Figure 9.2 below; for a discussion of parsing using "clever" backtracking which I believe can easily be reinterpreted to yield a set of such heuristics, see Hill [Hill 72]. The problem solver manipulates the fragments using these heuristics until either some solution is found, or the set of available heuristics is exhausted. I believe that a speaker of a language is aware that this latter process is taking place (although certainly not aware of exactly what the process entails).

Given two consecutive sentence fragments, if the first fragment is a sentence that is complete through some part of the verb phrase and the second fragment is a clause that lacks a subject, then see if the bulk of the first fragment can be turned into a relative clause of some type (using another set of heuristics), thereby converting the entire fragment into a single NP which will become the subject of the second fragment.

Figure 9.2 - A heuristic for recovering from a garden path.

To try to add some plausibility to this theory, let me suggest that the reader perform the following introspective experiment: After reading this paragraph, look at the garden path sentence given below in Figure 9.3 and try to find the correct analysis while attempting to introspect about your

thought processes while doing so:

The cotton clothing is made of grows in Mississippi.

Figure 9.3 - Try to introspect while reading this sentence.

When the author first looked at this sentence, it seemed as if the following happened (all of which, not surprisingly, is consistent with the above theory): The sentence read easily until the word "grows" was encountered. At that point it became clear that I had been led down the garden path, and had misanalyzed the sentence to have the structure shown in (a) of Figure 9.4 below. I completed reading the sentence, and the rest of the sentence took on the structure shown in 9.4.b. At this point, a fair amount of time passed while I studied these two grammatical fragments. I was conscious of exerting quite a bit of mental energy, but unaware of the exact nature of the internal process. Suddenly, I was aware that the phrase "clothing is made of" was a relative clause, and reanalyzed the first fragment as shown in 9.4.c. Given this structure, it immediately became clear that the overall structure was as shown in 9.4.d. (The heuristic used to solve this problem was that stated in Figure 9.2 above, I believe.)

(a) [$_S$ [$_{NP}$ The cotton clothing] [$_{VP}$ is made [$_{PP}$ of [$_{NP}$????]]]]
(b) [$_S$ [$_{NP}$????] [$_{VP}$ grows in Mississippi]]
(c) [$_{NP}$ The cotton [$_S$ clothing is made of t]]
(d) [$_S$ [$_{NP}$ The cotton [$_S$ clothing is made of t]] [$_{VP}$ grows in Mississippi]]

Figure 9.4 - Introspective stages of analyzing 9.3 above.

The key points of this account are: 1) The initial analysis of the sentence proceeded some distance into the sentence and then stopped quite suddenly with an initial fragment of the sentence (mis)analyzed; 2) The processing seemed to continue until the sentence was analyzed into two fragments; 3) The process of reconstructing the correct analysis was (i) done with conscious effort, and (ii) was based on the fragmentary initial analysis, not on a new analysis of the input sentence. All of this, if at all accurate, lends some credence to the theory stated above.

Diagnosing Between Imperatives and Yes/No Questions

With the question of how a complete natural language understanding system can recover from garden path sentences behind us, I will now turn back to the central issue of this chapter: a theory of what causes a sentence to cause the parser to take a garden path in the first place. This will be done in the context of a specific structural ambiguity and the diagnostic rule which attempts to resolve it. A consideration of where and why this rule fails will lead directly to a theory of garden paths.

Consider the following sentences, first presented in Chapter 1:

9.5.a Have the students who missed the exam take the exam today.
9.5.b Have the students who missed the exam taken the exam today?

Note that even though 9.5.a is an imperative sentence and 9.5.b is a yes/no question, the initial segments of these two sentences are exactly identical. The difference in sentence type is further reflected in the differing structural analyses assigned to these identical initial segments, as was illustrated in Figure 1.3. That figure shows that in the imperative

construction the word "have" serves as the main verb of the matrix S, while the NP "the students who missed the exam" serves as the subject of an embedded complement "the students...take the exam today". In the yes/no question, "have" serves as an auxiliary verb, while the NP "the students who missed the exam" is the subject of the major clause; there is no embedded complement in the analysis of the yes/no question. Because these two constituents serve in different grammatical roles in the two different constructions, neither of these two constituents can be utilized by a deterministic parser until it is determined whether the sentence is an imperative or a yes/no question.

Within the framework of the PARSIFAL grammar interpreter, we can say that an input sentence is ambiguous if at some point in the parsing process there are at least two rules of identical priority whose patterns are fulfilled by the state of the parser, and there are no rules of higher priority whose patterns match. Thus, given the set of rules presented in Chapter 5, the initial segment of these sentences is ambiguous in that it fulfills the patterns of both IMPERATIVE and YES/NO-QUESTION. This is because the word "have" (and, it may be noted, only the word "have") can be both an auxiliary verb, fulfilling the first node description in the pattern of YES/NO-QUESTION, or a tenseless verb, fulfilling the first node description in the pattern of IMPERATIVE. To resolve this ambiguity, either an additional diagnostic rule must be added to the grammar or else one of these two rules must be changed to eliminate the ambiguity altogether. In some sense, the two choices are equivalent, but the first is conceptually the cleaner of the two solutions, and it is this option which will be developed in the discussion

that follows.

Rather than merely presenting a diagnostic rule "sprung full grown from the head of Zeus", as it were, I will try to motivate the structure of the diagnostic one step at a time, trying to give the reader a feeling for the range of information that is used to distinguish between the two alternative analyses.

To begin with, it is clear that this diagnostic rule must apply whenever both IMPERATIVE and YES/NO-QUESTION are applicable, i.e. exactly when the word "have" appears in the first buffer position and an NP appears in the second position, as discussed above. Furthermore, the diagnostic must have access to the constituent in the third buffer location, since the pair of sentences given in Figure 9.5 above differ only in this third position. Given this, the pattern of the diagnostic rule, which I will call HAVE-DIAGNOSTIC, must be:

[=*have, tnsless] [=np] [t]

(The form of the word "have" that is tenseless is the word "have", so the feature *auxverb* need not be specified in this pattern.)

Second, it is clear that a sentence which fulfills this pattern can be a yes/no question only if the NP, which will be the subject of the sentence, agrees with the initial verb in person and number. Since the verb "have" will agree with an NP of any person and number except 3rd person singular, if the NP *is* 3rd person singular, the sentence must be an imperative. For example, in Figure 9.6 below, any clause whose initial segment is as shown

in (a) must be an imperative, while any clause whose initial segment is as shown in 9.6.b may be either an imperative or a yes/no-question. (It should be noted that this is the first mention of person/number agreement in this document. Usually, it is unnecessary to use agreement information to decide upon the proper interpretation of a sentence, and so agreement can usually be safely ignored. Where it seems to be of value is in resolving various ambiguous situations such as this. I believe that agreement information will become highly important when an attempt is made to resolve lexical ambiguity deterministically, but that issue is outside the scope of this document.)

(a) Have the student who missed the exam .....
(b) Have the students who missed the exam ....

Figure 9.6 - Sentence (a) must be an imperative.

If the NP is not 3rd person singular, then HAVE-DIAGNOSTIC must examine the constituent in the 3rd buffer position and try to determine the correct analysis for the input sentence on the basis of its features. If 3rd is an NP, then the sentence is a yes/no-question, as in Figure 9.7.a below. If 3rd is *tnsless* (i.e. tenseless), then the sentence is an imperative, as in 9.7.b.

(a) Have the boys *a dollar between them*?
(b) Have the students who missed the exam *take* it today.

Figure 9.7 - Diagnosis on the basis of the 3rd constituent.

One might suppose that if 3rd is *en*, as in 9.5.b above, that the sentence must be a yes/no-question, but this is not the case, as the following sentence shows:

9.8 Have the students who booed taken to the principal!

If this feature occurs, the sentence is either a yes/no-question, or else an imperative with a passive subordinate clause, but one cannot tell which without examining more of the sentence. Indeed, it is possible for such a sentence to be globally ambiguous (ignoring final punctuation), as the sentences 9.9.a and 9.9.b show:

9.9.a Have all of the eggs broken?
9.9.b Have all of the eggs broken!

In terms of the grammar interpreter, it is clear that more than three constituents must be examined to diagnose such pairs of sentences (using final punctuation to resolve the above pair). This seems to call into question the constraint which allows a grammar rules to examine only the first three constituents in the grammar interpreter's buffer.

For reasons which will become clear below, I will ignore this newly revealed problem, and simply stipulate, for the moment, that if HAVE-DIAGNOSTIC cannot determine what the structure of a given input sentence is on the basis of the three constituents that it can examine, then it will guess that the sentence is a yes/no-question. Assuming that the diagnostic tests discussed above reveal as much information as can be gleaned from an examination of only the first three buffer constituents, the code for the rule HAVE-DIAGNOSTIC can now be given; this diagnostic is shown in Figure 9.10 below. Since this rule must be run instead of either YES/NO-QUESTION

or IMPERATIVE if all three are applicable, it must be in the same packet as they, and it must have higher priority than either of them.

```
{RULE HAVE-DIAG PRIORITY: 5 IN SS-START
[=*have, tnsless] [=np] [t] -->
If 2nd is ns, n3p or 3rd is tnsless
        then run imperative next else
If 3rd is not verb then run yes-no-q next else
%If not sure, assume it's a y/n-q and%
        run yes-no-q next.}
```

Figure 9.10 - HAVE-DIAGNOSTIC summarizes the diagnostics discussed above.

Garden Paths and a Three Constituent Window

Let us now turn to the stipulation that each grammar rule can only examine the first three constituents in the buffer.

In this section and the next, I will present some evidence which supports the contention that this stipulation is empirically justified. This evidence results from a series of experiments conducted by the author during the course of this research. These experiments were highly informal in nature, and therefore the conclusions presented below can be taken as highly suggestive at best, although the results were typically quite clear.

In each of these experiments, some number of people were asked to read a sentence or a series of sentences typed on a sheet of paper. After reading each sentence, each person was asked if he encountered any difficulty while reading the sentence, or, if he was familiar with the notion, if the sentence was a garden path. At least 15 people were asked to look at

each of these sentences, unless the first four or five people who were asked to examine a particular sentence either all took a garden path or none took a garden path on that sentence. Somewhere over twenty such experiments were conducted, at frequent intervals, over a three year period, as I discovered more and more possible garden paths while developing grammar rules. About two thirds of the sentences people were asked to read led to garden paths for at least half the people asked. Approximately 50 or 60 people total participated in at least one of these experiments; most were students or staff at the MIT Artificial Intelligence Laboratory.

Let me now claim, on the basis of several of these experiments, that the diagnostic rule HAVE-DIAGNOSTIC resolves the relevant ambiguity approximately as well as many people, i.e. that many people, given a sentence which would trigger this diagnostic if input to the parser, will arbitrarily analyze it as a yes/no-question if they cannot diagnose its structure on the basis of the first three constituents. If the sentence is actually an imperative, these people will eventually realize that they were led down the garden path, and will consciously reanalyze the sentence.

In the most surprising of the relevant experiments, approximately 40 people were asked to read the following sentence, and then asked if they had any difficulty the first time they read the sentence, i.e. whether they noticed taking a garden path on the sentence:

9.11 Have the packages delivered tomorrow.

Of the 40 people asked, 20 reported misanalyzing the sentence as a question at least for an instant, and more than a few were quite startled when they

first realized that this simple sentence did not have the structure they had supposed. (It should be noted, by the way, that the sentence is entirely anomalous if interpreted as a yes/no question, yet still fully half the people shown this sentence misanalyzed it initially. This provides some evidence that high level semantic analysis is *not* employed to diagnose such ambiguities. Several people, when asked about this, mentioned imagining an image of a package giving birth to little packages in an attempt to semantically reconcile the false yes/no-question interpretation!)

This result immediately leads one to ask why half the people shown sentence 9.11 *do* analyze the sentence correctly. One possible explanation is that people simply choose one of the two possibilities at random when they are faced with such an undiagnosable ambiguity. If this is true, those people who did not notice taking a garden path while analyzing the sentence 9.11 simply happened to guess the correct structure initially, which seems quite plausible.

Unfortunately, this simple explanation seems to be wrong, as the following experiment shows. About fifteen people were asked to read the sentence 9.12 below, and see if they had any difficulty in understanding what it meant:

9.12 Have the soldiers given their medals by their sweethearts.

Only two of the fifteen had no difficulty with this sentence, and four or five had to be shown the correct analysis. (The sentence can be paraphrased "Have their sweethearts give the soldiers their medals.".) Almost without exception, everyone shown the sentence tried to interpret it as a yes/no

question. This result is consistent with other experiments involving this ambiguity as well; people almost never seem to mistakenly interpret an imperative as a yes/no question. Thus, almost without exception, if a person cannot resolve this ambiguity, he assumes the yes/no-question interpretation. This result disproves the simple "random-guess" theory suggested above.

One explanation consistent with both the above experiments seems to be the following, although I have no evidence to offer in its favor: The number of buffer cells that can be examined by a grammar rule, and perhaps the total number of cells in the buffer as a whole, is not a universal of language, but in fact varies from individual to individual, and perhaps varies consistently from speakers of one language to speakers of another. Thus, a given language, say English, might require, for reasonably full comprehension of spoken language, that grammar rules must be allowed to examine up to three buffer cells. Some speakers of English, however, for unknown reasons, may develop the ability to examine four, or perhaps even five cells at a time. These individuals can thus diagnose any local structural ambiguity which can be resolved within a "window" of that size.

Such an explanation nicely accounts for why half of those asked to read the sentence 9.11 above could correctly diagnose its structure, while only some small percentage of those asked to read 9.12 could do so. As the reader can verify for himself, 9.11 can be accurately diagnosed by a grammar rule that can examine *four* buffer cells simultaneously, while resolving the structure of 9.12 requires a rule that examines *five* buffer cells. The results presented above imply that half of those who participated in these

experiments can access only three buffer cells at a time, most of the remainder can access four cells, and a few individuals can access up to five cells. (This, of course, implies that it should be the same few people who correctly analyze potential garden paths in each of these experiments. This seems to be exactly the case.)

As a final note, I should mention that the sentences which I typically asked people to examine were examples which specifically embodied structural ambiguities for which I could not find a fool-proof diagnostic rule. Almost without exception, if I could not find a diagnostic rule which decided between structural analyses A and B for sentences embodying a given ambiguity, at least half the people who were shown example sentences would attempt to analyze all such sentences as if they were examples of structure A, i.e. they took the garden path on all sentences which were examples of structure B. Thus, this theory of garden paths seems to have fairly good predictive value, as least as measured by informal means such as this.

John Lifted A Hundred Pound Bags.

To show that the theory presented above has at least a degree of generality, this section presents another structural ambiguity which can lead to garden path sentences, and shows once again that garden paths occur in exactly those sentences for which full diagnosis would require examination of more than three buffer constituents within the framework of the grammar interpreter.

In an experiment similar to those discussed above, about twenty people were asked to read the sentence presented as the section heading immediately above, repeated below as 9.13. Approximately half of these people were led down the garden path by this sentence. As I will demonstrate in the remainder of this section, no rule that can only examine three buffer locations can correctly diagnose the structure of this sentence, which provides an explanation for the garden path in terms of the theory presented in the previous section.

9.13 John lifted a hundred pound bags.

The problematic segment of this sentence is the NP "a hundred pound bags"; this construction was discussed in general terms in Chapter 8, so I will briefly review the discussion presented in that chapter.

As Chapter 8 points out, any NP which begins with an indefinite pronoun (like "a") or indefinite quantifier (like "any"), followed by a word like "hundred" or "thousand" is locally ambiguous. The ambiguity can be diagnosed in general, I claimed in that chapter, by noting whether or not the following word is a singular "measure" noun, i.e. a word like "pound", "ton", and the like. If the following word is not a singular measure noun, then the determiner or quantifier is part of a number phrase; thus the phrase shown in Figure 9.14.a below is assigned the structure shown in 9.14.a'. If the following word *is* a singular measure noun, then the determiner is the determiner of the NP and the number word and measure word form an adjective phrase; this test will assign the structure 9.14.b' to the phrase 9.14.b.

(a) a hundred boys
(a') [$_{NP}$ [$_{NUMP}$ a hundred] [$_{N}$ boys]]

(b) a hundred pound rock
(b') [$_{NP}$ [$_{DET}$ a] [$_{ADJP}$ hundred pound] [$_{N}$ rock]]

Figure 9.14 - Another ambiguity requiring a diagnostic rule.

Actually, the situation is slightly more complicated than this. As discussed in Chapter 8, the diagnostic rule which embodies the test discussed above must apply not only to inputs containing just the word "hundred" but also to number phrases (NUMPs) which begin with this word or "bignums" like "thousand", "million", etc. (For reasons not relevant to this discussion, it is necessary to distinguish the word "hundred" from words like "thousand" or "million", which are labelled with the feature *bignum*.) Because of this, number phrases must be initiated by an attention shifting rule, and thus even the word "hundred" in isolation will be attached to a dominating number phrase node before the diagnostic rule will be triggered. Ignoring the details of the structure that the grammar assigns to number phrases, it is sufficient to say only that before the diagnostic, which I will call A-HUNDRED-DIAG, is executed, the word "hundred" will have been attached as the num1 of the num1 of the number phrase. Thus the pattern of this rule must test to see whether the num1 of the num1 of 2nd is either the word "hundred" or a bignum. If this rule matches and the third constituent in the buffer is a singular measure noun, then this rule causes the determiner to be attached to the NP, resulting eventually in the structure shown in 9.14.b'; if not, then it attaches the determiner to the NUMP, resulting in the structure shown in 9.14.a'. The code for A-HUNDRED-DIAG is given

immediately below in Figure 9.15.

```
{RULE A-HUNDRED-DIAG PRIORITY: 6. IN PARSE-DET
[=*a]
[=num, bignumg; the num1 of the num1 of * is any of *hundred, bignum] [t] -->
If 3rd is noun, ns, measure
     then run determiner next
     else attach 1st to the num1 of 2nd as num0;
          deactivate parse-det;
          activate parse-qp-1.}
```

Figure 9.15 - The diagnostic rule A-HUNDRED-DIAG.

With this diagnostic as a starting point, let us turn our attention back to sentence 9.13, and try to explain why this sentence is a garden path.

While the diagnostic presented above correctly resolves the local ambiguity in the phrases shown in Figure 9.14 above, it is not in fact generally correct. For example, the diagnostic will incorrectly analyze the phrase shown in Figure 9.16.a below, assigning it the incorrect structure shown in 9.16.b; the correct analysis is shown is 9.16.c. (Let me note parenthetically that this misanalysis is identical to that assigned the final NP of 9.13 by those who took the garden path, according to introspective reports.) The question to be answered, of course, is whether the diagnostic can be fixed, and if so how.

(a) a hundred pound rocks
(b) [$_{NP}$ [$_{DET}$ a] [$_{ADJP}$ hundred pound] [$_{N}$ rocks]]
(c) [$_{NP}$ [$_{NUMP}$ a hundred] [$_{ADJP}$ pound] [$_{N}$ rocks]]

Figure 9.16 - The diagnostic above analyzes (a) as (b), not (c).

Compare for the moment the phrase given in Figure 9.14.b above, whose structure is shown in 9.14.b', and the phrase given in 9.16.a above, with the structure shown in 9.16.c. These two phrases begin with the same three words "a hundred pound", yet the grammatical function of each of the two words "a" and "hundred" is totally different in 9.14.a' and 9.14.b' above. Since both of these words have different grammatical functions in these two phrases, neither of these two words can be utilized and removed from the buffer until the ambiguity between the two structures 9.14.b' and 9.16.c is somehow resolved. But since the following constituent of both of these phrases is identical, any diagnostic which will resolve these two structures must examine a fourth constituent. This of course means that 9.14.a and 9.16.a cannot be differentiated on the basis of any diagnostic rule that can only examine three buffer positions.

Given the theory presented in the previous section, this last fact predicts that one of the two phrases 9.14.b and 9.16.a must necessarily lead to a garden path for those people who can be modelled as having a buffer window of length three. This prediction is borne out by the results of the experiment presented at the beginning of this section. And this, of course, provides another bit of evidence in favor of the "buffer window of length three" theory of garden path sentences.

Most Garden Paths Cause No Difficulty in Spoken Language

One potential objection to a fundamental premise of the above discussion is that the sentence shown in Figure 9.3, and, indeed, all the

examples listed in Figure 9.1 above, are garden paths only in *written* language; that the garden path effect, at least for these sentences, is an artifact of the written language. If spoken with proper intonation, none of these sentences is a garden path.

This objection is fundamentally correct, I believe, if one accepts the reasonable premise that speech is more fundamental than written language. But this objection, though correct, is in fact no objection at all! (The same holds true for the objection that these sentences are taken out of context, although I believe that for many of these sentences, a reader will take the garden path even if given strong contextual clues.)

As Simon has pointed out [Simon 69], as long as a mechanism works correctly, one can only learn about the structure of the task that it performs from its input/output behavior; one learns little about the internal structure of the mechanism itself. It is only by pushing the mechanism until its performance begins to degrade that one can learn about the structure of the mechanism itself. If a mechanism is a sealed black box, the only way to determine how a mechanism works is to determine how it begins to fail.

The phenomenon I discuss here is thus artificial by necessity; by throwing away prosodic information, the performance of the syntactic analysis mechanism degrades, and some of its internal characteristics may be deduced from the form of this degradation. The garden path phenomenon might be largely artifact, but I believe it is an artifact that yields important clues about the real nature of the process of syntactic recognition.

Diagnostic vs. Non-diagnostic Rules

Before concluding this discussion of structural ambiguity, it is interesting to note that the diagnostic rules developed above and the grammar rules discussed in previous chapters differ in fairly significant ways.

As discussed above, the purpose of diagnostic rules is to explicitly decide between two possible constructions. These two constructions, typically, are entirely unrelated, but just happen to result in terminal strings whose initial segments are identical in special cases. Diagnostic rules recognize and resolve these special cases, and therefore diagnostic rules are typically "special purpose". This is in marked contrast to the grammar rules presented in earlier chapters; each of these earlier rules is written to embody a broad grammatical generalization as independently of other grammatical phenomena as possible.

Another difference between diagnostic rules and the grammar rules presented in previous chapters is that the rules presented earlier can be constrained to utilize a limited range of syntactic operations, while diagnostic rules must, it seems, perform a much wider range of syntactic operations. For example, the diagnostic A-HUNDRED-DIAG must attach a node in the buffer to another node in the buffer, contrary to a constraint which seems to hold for all "general" (i.e. non-diagnostic) grammar rules. (One might speculate that this can be accounted for in terms of a developmental theory of language recognition in which general rules are learned early and are highly constrained in the possible operations they can be composed of. Later,

the need for diagnostic rules is recognized and diagnostic rules are acquired, but now a much richer set of operations is allowed. If this suggestion is correct, it should be possible to find young children who can parse unambiguous cases of each of the two constructions between which a given diagnostic distinguishes, but who always interpret a sentence ambiguous between the two as consistently being one or the other of the two constructions, i.e. who always take the garden path for one or the other of the two constructions.)

In short, the rules which will be presented in this chapter are ugly rules to handle ugly situations. While I do not know whether the complexity of these rules is intrinsic to the process of deterministic syntactic analysis or merely an indication that the approach I take in this chapter is mistaken, it does seem that the residual problems which diagnostic rules attempt to solve are real and highly idiosyncratic.

As one last point, it should be noted that in the initial formulations of the theory of deterministic parsing presented in [Marcus 74], these diagnostic rules were seen to hold a crucial position in the theory. It is now my opinion that these rules are actually at the periphery, in that their role is to "patch" those situations where two highly general rules just happen to overlap in applicability. It is crucial to the theory that such rules can be written which adequately handle such cases, but by their very nature, few generalizations can be stated about such rules themselves.

CHAPTER 10

ON THE NECESSITY OF SOME SEMANTIC/SYNTACTIC INTERACTIONS

Introduction

In the previous chapter, I investigated the use of purely syntactic criteria to resolve local structural ambiguities. As is well known, however, syntactic criteria alone are insufficient to constrain the parsing process; many analyses that are possible on purely syntactic grounds will be ruled out by semantic processing as anomalous. Therefore, if language is to be parsed deterministically, there must be some interaction between syntax and semantics to rule out these anomalous interpretations before they are constructed by the parser. This chapter is intended as an exploratory investigation into the sorts of syntactic/semantic interactions that are necessary to deterministically parse English.

I will show in this chapter that simple semantic well-formedness tests are not sufficient to allow deterministic parsing of grammatical structure. Given the assumption that the determinism hypothesis is true, I will argue that there are necessarily interactions between the syntactic and semantic components of a natural language understanding system, and that such interactions must have specific sorts of properties, many of which are not currently embodied in any existing natural language understanding systems.

It should emphasized at the outset that this approach differs fundamentally from earlier explorations in the use of syntactic/semantic interaction in that the investigations which follow are based on argument from necessity, not argument from efficiency. The use of syntactic/semantic interaction in previous natural language understanding systems (which will be discussed below) has been motivated in terms of the resulting improvement in the efficiency of the parsing process. This chapter will demonstrate the *necessity* for such interactions as consequences of the determinism hypothesis, and will show, based on linguistic data, that it is necessary for such interactions to have certain properties.

In the following pages, I will attempt to demonstrate:

-That tests of *comparative* semantic goodness are necessary, i.e. that a deterministic parser must be able to ask a semantic component not only whether a given constituent is acceptable in a particular syntactic role, but also *how* good it is in that role.

-That the grammar itself must embody, at least implicitly, the same notion of degree of goodness in the syntactic realm.

-That diagnostic rules must take into account the relative strength of both these sorts of bias, i.e. that syntactic decisions are influenced by relatively subtle interactions of syntactic and semantic biases.

This chapter will not present a full theory of semantic/syntactic interaction, but rather a series of observations which I believe lead toward such a theory. While I will argue for general properties that such a theory must have, my observations are based upon the investigation of only a handful of specific syntactic constructions; before a more complete theory can be formulated with any great certainty, a much wider range of such constructions must be investigated. Another limitation of the present

investigation is that there are no existing semantic/pragmatic components capable of answering semantic questions of relative goodness, to the best of my knowledge, nor is it entirely clear how a reasoning component can be implemented that answers these sorts of questions. Until this is done, a full theory of such interactions cannot be formulated. It is interesting that some deep properties of such a theory can be demonstrated before specific theories themselves can be proved or disproved.

As a general framework for the investigations that follow, I should note that this work is based on the general conjecture that syntax and semantics can be characterized as what Simon calls nearly-decomposable subsystems [Simon 69], that there is a fixed small set of possible interactions between the syntactic and semantic components, and that the bandwidth required by these interactions is small. (Note, by the way, that such a suggestion is quite consistent with the so-called "autonomy of syntax" hypothesis assumed by those who study linguistic competence.) Thus, this chapter is the first step of a larger research plan to find syntactic situations where semantic/syntactic interactions *must* occur, given the determinism hypothesis, and then to characterize as small a set of types of interaction as possible. (See Appendix A for a further study of this sort.)

Historical Perspective

For the sake of comparison, it will be useful to contrast the nature of the syntactic/semantic interactions that will be presented below with the type of interactions utilized in two earlier natural language understanding systems of seminal importance. In this section I will briefly present the

mechanisms for syntactic/semantic interaction embodied in Woods' LUNAR system [Woods 72], the first widely known system to use semantics to control the parsing component, and Winograd's SHRDLU [Winograd 71], the first system to interleave syntactic and semantic analysis, allowing much tighter control of the parsing process by the semantic component. I will also separately discuss Woods' Selective Modifier Placement Facility, an addition to the basic ATN framework and still one of the most sophisticated uses of semantic/syntactic interaction in any existing program.

In the LUNAR system, after the parser has generated a complete syntactic analysis for a given input, the syntactic structure is handed over to the semantic analyzer which then attempts to build a semantic interpretation for that structure. The state of the parser, i.e. its stack of untested hypotheses, is not discarded at this time; the stack is saved until semantic analysis is complete. If the semantic component is successful, the resulting interpretation will be given to the "understanding" component, in this system an information retrieval mechanism, and the parser's state will be discarded. If the semantic component cannot build a semantic interpretation for the sentence, i.e. if it determines that the given syntactic analysis is anomalous, it signals this anomaly to the parser by causing a failure to propagate back from the parser's final state. This "failure signal" causes the parser to reject the hypothesis that the parse as it stands is complete, i.e. to backup across the arc that caused the parse to terminate. This, in turn, will cause the parser to continue to search through the remaining set of unattempted arcs until another syntactic analysis is found, as was discussed earlier in Chapter 1.

The LUNAR system's use of semantics as a postfilter to control the production of alternate syntactic analyses was an important advance, but this technique is only a step toward tight semantic/syntactic interaction. While LUNAR's semantics can reject a parse on non-syntactic grounds, it is only able to accept or reject complete analyses, and cannot signal which part of a parse is unacceptable if analysis fails.

Consider, for example, the behavior of this system if an NP near the beginning of an input string is (mis)parsed in a way that is syntactically plausible but semantically anomalous.

One possibility is that the parser will find a consistent syntactic analysis for the rest of the input despite the misanalysis of that NP. In this case, the mistaken analysis will not be ruled out on semantic grounds until the parse has been completed. Furthermore, the only information that is passed back to the parser is a signal that the parse as a whole is unacceptable. Thus the parser can only correct its error by proceeding blindly through the entire space of remaining grammatical possibilities until it chances upon another analysis for the NP. Before it can do this however, it must attempt to reanalyze every structure built later in the parsing process.

The other (more likely) possibility in this situation is that the parser will not find a consistent syntactic analysis for the remainder of the input after having misanalyzed the NP initially. In this case, of course, the

anomaly will never come to the attention of semantic processes. Nevertheless, the parser may struggle on bravely for some time until it backs up and finds a correct analysis for the NP. The parser is forced to rule out the parse on syntactic grounds when there was earlier clear semantic evidence that the analysis must be wrong.

To utilize semantic information more effectively, Winograd's SHRDLU was designed to incrementally call for semantic analysis during the course of syntactic analysis. Winograd's system calls upon semantics to judge the semantic well-formedness of proposed NPs at two separate points during parsing, first after the head noun has been found, then after all modifiers have been assigned. SHRDLU's parser also calls semantics after each subordinate clause has been constructed to confirm that the clause is semantically well-formed. Winograd's system thus uses semantics not only to accept or reject syntactic analyses, but also to actively prune the space of grammatical possibilities.

(Woods makes one counterargument to Winograd's technique that is worthy of note [Woods 73]: Current techniques for semantic analysis are far slower than current techniques for syntactic parsing. Thus, from an engineering standpoint, it is currently less efficient to use semantics as an interactive filter for the syntactic component than it is to simply let syntax filter what it can, calling semantics only after a syntactic analysis has been completed and it must be subjected to semantic interpretation. From an engineering viewpoint, assuming a non-deterministic system, Woods' argument is hard to refute.)

While the bulk of the semantic filtering in both Woods' and Winograd's systems rests on purely semantic restrictions which are expressed in terms of semantic marker systems [Katz & Fodor 64], Winograd's system uses one form of pragmatic filtering as well. If an NP is found to be definite, e.g. "the block which is sitting in the box", then the system tests the corresponding semantic interpretation against its internal world model to see if the interpretation has exactly one referent. If the phrase is found to have more than one referent, discourse heuristics are applied to limit the list of referents; if this is unsuccessful, the syntactic analysis is rejected. While such a filter works in Winograd's toy domain, it should be noted that this filter is in general wrong, especially as a test of syntactic well-formedness. Thus, 10.1 below is both syntactically and semantically acceptable, even if the hearer does not know who the speaker is referring to. Likewise, 10.2 is perfectly acceptable even if the hearer, again, does not know who the speaker is referring to.

10.1 Did you see the guy who wandered in here ten minutes ago?
10.2 The guy who wandered in here ten minutes ago was obviously drunk.

While not used as part of the LUNAR system *per se*, Woods' parsing system includes an experimental facility outside of the basic ATN framework for doing what he calls Selective Modifier Placement (SMP) [Woods 73]. This facility is called by the ATN system whenever a prepositional phrase has been parsed by the grammar. The SMP mechanism computes a list of all possible attachment points for the PP, and then uses a semantic component

based on case frames and semantic markers to decide which of these attachment points *must* take a modifier of the PP's semantic type, which *can optionally* take such a modifier, and which *cannot* take such a modifier. It then chooses the alternative which is "the closest candidate that needs the modifier most".

I will argue below that a mechanism of exactly this sort is more than a good idea; something like Woods' SMP facility is *necessary* to any system that attempts to parse language deterministically. In this sense, the SMP facility is the first syntactic/semantic interaction mechanism of adequate power to handle natural language. However, this mechanism is merely a special case of a much larger class of mechanisms of much the same sort. Thus, what is needed is a natural language understanding system where such strategies can be expressed as an integral part of the parsing mechanism, rather than as special purpose boxes patched onto some other mechanism.

The Problem of Wh-Questions and Indirect Objects

One fairly difficult problem that a deterministic grammar must handle is that of deciding from where in a relative clause or wh-question the wh-head has been "moved". In the current parser, this means deciding where to place a trace in the annotated surface structure that the parser is building if there is an unutilized wh-comp, as explained in Chapter 7. While the placement of traces that result from operations like NP-preposing (i.e passivization and the like) simply falls out from the structure of the grammar, as demonstrated in Chapter 5, the discovery of the "gaps" left by WH-movement cannot be so automatic, as will be thoroughly demonstrated

below.

Often the "WH ungapping" problem is fairly simple as in 10.3.a below; given that "throw" is a transitive verb, it is clear that its object is missing and thus that the wh-comp must serve as object with the analysis as approximately represented in 10.3.b, where "t" represents a trace bound to "what".

Sometimes, however, the problem is much harder; consider, for example, the problem presented by sentences like 10.4.a and 10.5.a. These two superficially identical sentences, it turns out, have quite different analyses. In 10.4.a the wh-comp serves as the direct object (DO), the entity which John gave, with the analysis indicated by 10.4.b, while in 10.5.a it serves as the indirect object (IO), the entity to whom something was given, with the analysis indicated by 10.5.b. From this example, it is clear that for a parser to diagnose the correct structure of a wh-question of this sort, some sort of syntactic/semantic interaction must be involved. The question that must now be answered is exactly what form this interaction must take. (Let me caution the reader about one common misapprehension: the assumption that the word "whom" explicitly flags the *indirect object* of a verb. In fact, "whom" can flag the direct object as well, as in "Whom did the knight feed to the dragon?".)

10.3.a What did John throw to Bill?
10.3.b What did John throw *t* to Bill?

10.4.a What did John give Bill?
10.4.b What did John give Bill *t*?

10.5.a Whom did John give the extra tickets?
10.5.b Whom did John give *t* the extra tickets?

Before beginning this investigation, it is necessary to formulate more precisely the syntactic question that must be answered to resolve the structures of 10.4 and 10.5 above. This question, first of all, arises at the point in the parsing process right after the parser has determined that "give" is the main verb of the WH-question. At this point, it is clear that the verb must take both a direct object (a DO) and an indirect object (an IO), given that both the following constituent and the question phrase, the wh-comp, are NPs. What must be decided is whether the wh-comp or the NP following the verb serves as IO. (There are, of course, alternate ways of putting this same question.) Note, by the way, that this question falls under the definition of structural ambiguity presented in Chapter 9, and therefore that the grammatical rules that resolve this issue can be considered diagnostics. In the following discussion, I will refer to the NP following the verb as NEXT and the question phrase as the WH-COMP.

Some Inadequate Explanations

First of all, let me show that even if one rejects the determinism hypothesis, the types of semantic/syntactic interaction used by Woods' and Wincgrad's systems are inadequate (with the exception of Woods' SMP). In

both systems, it is assumed that the semantic tests of well-formedness are based primarily on the *underlying* grammatical relations between a verb and its associated NPs. This can be shown to be an inadequate theory.

Consider 10.6-10.9 below. The judgements of acceptability shown are shared by many people who were questioned by the author, although there are several other common dialects. (Approximately 20 people were informally questioned about these sentences.) The judgements reflect people's first reactions to these sentences; often an individual's judgements change over several minutes reflection. "*" indicates a sentence that is completely unacceptable, while "?" indicates that the sentence is somewhat unacceptable. "(?)" is used to indicate that a sentence is somewhat questionable for some speakers, and perfectly fine for others. Note that such judgements in themselves make no statement about *why* a sentence is unacceptable. The "b" sentences are the declaratives which correspond to the primary interpretations which people initially give the "a" sentences.

10.6.a *Which boy did the knight give the dragon?
10.6.b The knight gave the dragon a boy.

10.7.a Which dragon did the knight give the boy?
10.7.b The knight gave the boy a dragon.

10.8.a (?)Which boy did the knight give the sword?
10.8.b The knight gave a boy the sword.

10.9.a Which sword did the knight give the boy?
10.9.b The knight gave the boy a sword.

Note that 10.6.a cannot be unacceptable on purely syntactic grounds; the primary (initial) reading of 10.6.a, that parallel to 10.6.b, has the same syntactic structure as 10.7.a, which is perfectly acceptable. Furthermore, 10.6.a cannot be unacceptable on purely semantic grounds involving predicate/argument relations; the primary reading of 10.6.a has the same predicate/argument structure as 10.6.b, which is perfectly fine. If 10.6.a is thus syntactically well formed, and is acceptable in terms of underlying semantic relations, its unacceptability must be due to the fact that constituents filling various semantic roles appear in particular positions in the *surface* syntactic structure of the sentence.

These facts show that the primary form of semantic/syntactic interaction used by Woods and Winograd is necessarily inadequate. Since 10.6.a is syntactically well-formed, it will pass through the syntactic component of either of their systems and since it has a sensible semantic interpretation, it will pass through their semantic components as well. There is simply no facility in either system for directly relating semantic role to surface syntactic position. It should be noted, however, that these observations are consistent with the determinism hypothesis, since some sort of semantic test *must* be performed at a level which is aware of surface structure position to decide whether NEXT or the WH-COMP will serve as the indirect object.

It can now be demonstrated that any filter that attempts to characterize what sentences are unacceptable in terms of the properties of any single NP *in isolation* is inadequate, even if it takes into account the

semantic role, semantic properties, and surface position of the NP. I will show that for each of the two NPs which are NEXT and WH-COMP in 10.6.a, that there is a much better sentence with that NP in the same semantic and syntactic role as in the primary interpretation of 10.6.a. Thus, it cannot be the case that 10.6.a is unacceptable because of some simple filter which declares a sentence to be unacceptable if it includes an NP of given semantic properties in given semantic and surface syntactic roles.

For such a filter to be correct, 10.6.a would have to be unacceptable for one of two reasons: either (i) "boy" cannot serve as DO while appearing as the WH-COMP, as it does in 10.6.a, or (ii) "dragon" cannot serve as IO while appearing as NEXT, as it does in 10.6.a. But consider 10.10 and 10.11 below. (The judgements on these sentences are agreed upon by those whose judgements are reflected in 10.6-10.9. It should be noted that while 10.10 is not perfect, informants judge it to be only slightly funny; it is much better, for instance, than 10.8.a.) While 10.10 is not perfect, it is almost acceptable, and certainly much better than 10.6.a. Yet in 10.10, "which boy" fills exactly the same syntactic and semantic roles as it does in the initial, bad reading of 10.6.a. This rules out (i) above as a possible reason that 10.6.a is bad. Similarly in 10.11, "the dragon" has exactly the same semantic and syntactic role as it does in 10.6.a, yet 10.11 is perfectly acceptable. Thus, (ii) above cannot be the reason that 10.6.a is unacceptable. Furthermore, if either (i) or (ii) were correct, it is not clear why 10.6.a would not then receive an initial interpretation parallel to 10.8.a and be perfectly acceptable.

10.10 (?)Which boy did the knight give the cannibals?
10.11 What did the knight give the dragon?

From this argument, it can be concluded that any theory that tries to account for the unacceptability of 10.6.a by constraining only one of the two NP positions must be wrong, even if such a theory takes into account both surface position and underlying role of the NP. Any adequate theory must thus crucially involve the relationship *between* the two NPs that serve as NEXT and the WH-COMP.

The Notion of Syntactic and Semantic Biases

I will now argue that what is needed are *comparative* tests of some sort between the two NPs. I will show that the judgements given for the sentences discussed above can be accounted for given the related notions of *syntactic* and *semantic biases*. The idea behind the notion of semantic bias is that one possible semantic interpretation of a sentence might be preferable to another, although both are acceptable. Similarly, the notion of syntactic bias captures the idea that one syntactic analysis of a sentence might be preferable to another, even though both are grammatical. Thus, the fact discussed in Chapter 9 that almost all speakers who take a garden path on sentences ambiguous between yes/no-question and imperative interpretations misanalyze imperatives as yes/no-questions, but not vice-versa, can be characterized by saying that speakers have a syntactic bias for yes/no-questions over imperatives. (The notion of syntactic bias was first developed by Bever [Bever 70].) Given these notions, it will be demonstrated that the

extent to which any of these sentences is acceptable depends upon the extent to which these biases agree, or if they disagree the extent to which one bias is strong enough to totally override the other.

It is clear that there is some sort of syntactic preference for NEXT to serve as indirect object in the sorts of sentences discussed above. Thus both 10.6.a and 10.7.a, repeated below as 10.12 and 10.13, receive primary interpretations that assign NEXT as IO, even though 10.12 is judged to be unacceptable, and may then be reinterpreted to have much the same meaning as 10.13. Yet it is not the case that NEXT must *always* serve as indirect object for all speakers; thus among speakers who find 10.12 bad and 10.13 good, 10.8.a, repeated here as 10.14, is acceptable for some, and is almost acceptable for all. Thus it seems necessary to posit a notion of syntactic *bias* (to be differentiated from any notions of syntactic requirements), and more generally a notion of *degree of goodness* along a given dimension. Note that such a notion is different from the notion of degree of grammaticallity; speakers who accept 10.13 as well formed and reject 10.12 clearly exhibit this syntactic bias, yet some such speakers do not judge 10.14 to be ungrammatical, even though the syntactic bias which they demonstrate has been overriden.

10.12 *Which boy did the knight give the dragon?
10.13 Which dragon did the knight give the boy?
10.14 (?)Which boy did the knight give the sword?

Let me once more approach the question of why 10.12 is

unacceptable, beginning with a review of what has been shown above. First, as was demonstrated immediately above, the initial interpretation given to 10.12 assigns the NP "the dragon" as the IO, which is exactly in keeping with the preferred syntactic structure. Thus 10.12 cannot be bad for syntactic reasons. Second, it has been demonstrated that sentences similar to 10.12 which retain exactly one of the two NPs that are NEXT and the WH-COMP in 10.12 in the same syntactic position and semantic role as in that sentence are much better than 10.12. Thus 10.12 cannot be bad because of any restriction on single NPs in isolation. Third, as was shown early in this chapter, the initial reading given to 10.12 (i.e. the reading parallel to the declarative repeated below as 10.15.a) is perfectly acceptable semantically. Thus 10.12 cannot be bad because its semantic interpretation is unacceptable. What factors are left which can account for the unacceptability of 10.12? And why is 10.12 bad while its corresponding declarative is quite acceptable?

10.15.a The knight gave the dragon the boy.
10.15.b The knight gave the boy the dragon.

One possibility is to state that it is the *combination* of the NP "Which boy" *as WH-COMP* serving as DO, and the NP "the dragon" *as NEXT* serving as IO which makes this sentence bad, which seems to be exactly the case. But such a declaration by itself amounts to stating that 10.12 is bad because 10.12 is bad. By extending the notion of bias to include semantic bias as well, an explanation of *why* this combination is bad is is immediately forthcoming, as I will now show.

Let me suggest that while 10.15.a and 10.15.b, the declaratives

corresponding to the two possible semantic interpretations of 10.12, are both semantically acceptable, that 10.15.b is in some sense semantically preferable to 10.15.a. If this is true, then the unacceptability of 10.12 might have something to do with the fact that the the reading which is semantically preferable (that parallel to 10.15.b) is the second choice syntactically. Thus 10.12 will be unacceptable because the syntactic and semantic biases conflict. (In either of the corresponding declaratives 10.15a-b, syntax does not present a bias, but rather dictates that exactly one syntactic structure is acceptable, and thus both are acceptable.)

Given the notion of bias, we are now well on our way to explaining why 10.12 is bad. What remains to be done is (i) to formulate a precise rule which will account for the judgements given for the range of sentences discussed above, and (ii) to determine exactly what it is about the interpretation of 10.15.b that is semantically preferable to 10.15.a. The remainder of this chapter will focus on (i); issues relating to (ii) will be discussed only briefly. This seems to be an interesting area for future investigation.

A Solution to the Problem

There are several possibilities as to what strategy might be used to discover that one reading of a sentence like 10.6-10.10 is preferable to another. While I will mention two such options here, I are not prepared at this time to argue for one as opposed to another. The data discussed above does not distinguish between these possibilities; there are other phenomena that provide some such evidence, as will be discussed in Appendix A, but

that evidence is far from conclusive.

One such strategy would be for the grammar to decide to use either NEXT or the WH-COMP as the IO without using the knowledge that whichever is not chosen as IO will then serve as DO. Such a strategy might ask semantics which of the two it prefers as the IO or it might ask *by how much* semantics prefers one or the other as IO. The semantic decision might be based (for the sake of example) on criteria such as the fact that higher animate entities, i.e. people, are more often given something than merely animate entities, which in turn are given something more often than inanimate entities. This might be thought of as a preference scale which can be expressed graphically as

higher animate > animate > inanimate.

Another possibility, which seems more plausible intuitively, might be to ask semantics which of the two possible orderings of the two NPs as indirect and direct object it prefers. It is this possibility that will be pursued in detail below. For the sake of simplicity, a system of semantic markers like that assumed implicitly in the preceding paragraph will be used as the basis for semantic tests, although it is easy to show that a semantic marker representation of the detail shown above is not adequate for the sorts of judgements that must be made to correctly handle all such sentences. *It must be stressed that I intend to make no claims for the adequacy of semantic markers for such a task.*

Given the assumption that the semantic tests will be done on

possible assignments of both the IO and DO jointly, and also given a set of presumed semantic preferences, I can now present an algorithm that produces the correct judgements on sentences 10.6.a-10.9.a, and 10.10-10.11. Let me first present the algorithm and then state the semantic judgements I assume.

Let us refer to the analysis of these sentences with NEXT as IO and the WH-COMP as DO as the NEXT-as-IO analysis, and the alternate analysis as the WH-COMP-as-IO analysis. Then to decide which analysis of a sentence of the form of 10.6-10.10 is correct, the following algorithm can be executed at the point at which the sentence up to the verb has been parsed and NEXT has been encountered, parsed as an NP and is waiting to be assigned a syntactic role:

1) If exactly one analysis is not semantically acceptable, choose the alternative analysis.

1a) (For dialects that find 10.14 somewhat unacceptable:) If the analysis chosen by rule (1) is not NEXT-as-IO, then the sentence is mildly bad.

2) If the semantic interpretation of the NEXT-as-IO analysis is semantically preferable to the interpretation of the WH-COMP-as-IO analysis, choose the NEXT-as-IO analysis.

3) Otherwise, the sentence is bad, but try the NEXT-as-IO analysis.

Restating this algorithm in terms of semantic and syntactic preferences makes the logic of the algorithm a little more transparent (remember that the NEXT-as-IO analysis is syntactically preferred to the WH-COMP-as-IO analysis):

1) If exactly one analysis is not semantically acceptable, choose the alternative analysis.

1a) (For dialects that find 10.14 somewhat unacceptable:) If the analysis chosen by rule (1) is not the syntactically preferred analysis, then the sentence is mildly bad.

2) If the syntactically preferred analysis is also semantically preferred, choose it.

3) Otherwise, the sentence is bad, but choose the syntactically preferred analysis.

We can now investigate how this algorithm accounts for the judgments of our test cases and what semantic preferences these judgements assume. Each of the sentences 10.6 - 10.10 above will be considered in turn. I will first give each sentence and the assignment of NEXT and the WH-COMP to the IO and DO roles that this algorithm produces, and then give the analysis that results in this assignment:

10.6.a *Which boy did the knight give the dragon?
Assignment: (DO) (IO)

--Assumed semantic forms:

(sf1): X gives animate entity to higher animate entity.

(sf2): X gives higher animate entity to animate entity.

--Assumed result of semantic tests:

(result1): (sf1) is acceptable.

(result2): (sf2) is acceptable.

(result3): (sf1) is preferable to (sf2).

--Analysis: Rule 1 is inapplicable since both (sf1) and (sf2), the two possible interpretations of 10.6.a, are semantically acceptable. Rule 2 is inapplicable since the interpretation corresponding to the NEXT-as-IO analysis, (sf2), is not preferable to (sf1), the interpretation of the WH-COMP-as-IO analysis. Therefore rule 3 applies; the sentence is

bad and NEXT is initially interpreted as IO.

10.7.a Which dragon did the knight give the boy?
Assignment: DO IO

--Assumed semantic forms: (sf1), (sf2)

--Assumed result of semantic tests: (result1), (result2), (result3)

--Analysis: Rule 1 is inapplicable, since (sf1) and (sf2) are both acceptable. Rule 2 applies since the interpretation corresponding to NEXT-as-IO is semantically preferable to that corresponding to WH-COMP-as-IO by test (result3). Thus, the analysis is NEXT-as-IO and the sentence is good.

10.8.a (?) Which boy did the knight give the sword?
Assignment: IO DO

--Assumed semantic forms:

(sf3): X gives inanimate entity to animate entity.

(sf4): X gives animate entity to inanimate entity.

--Assumed result of semantic tests:

(result4): (sf3) is acceptable.

(result5): (sf4) is unacceptable.

--Analysis: Rule 1 applies; the only acceptable semantic analysis is (sf3), which corresponds to the WH-COMP-as-IO syntactic analysis. Thus, the analysis is WH-COMP-as-IO. If the speaker has Rule 1a, the sentence is mildly bad.

10.9.a Which sword did the knight give the boy?
Assignment: DO IO

--Assumed semantic forms: (sf3), (sf4)

--Assumed result of semantic tests: (result4), (result5)

--Analysis: Rule 1 applies; the only acceptable semantic analysis corresponds to NEXT-as-IO.

10.11 What did the knight give the dragon?
Assignment: DO IO

--Same analysis as 10.9.a.

I believe that all of the above semantic assumptions are intuitively quite reasonable.

While the set of rules stated above adequately handles the cases discussed immediately above, it is not subtle enough to pick out the slight unacceptability of 10.10. 10.10 also poses a problem in that the necessary "semantic markers" are clearly *ad hoc*:

10.10 (?) Which boy did the knight give the cannibals?
Assignment: DO IO

--Assumed semantic forms:

(sf5): X gives higher animate entity to higher-animate-eating entity.

(sf6): X gives higher-animate-eating entity to higher animate entity.

--Assumed result of semantic tests:

(result6): (sf5) is acceptable.

(result7): (sf6) is acceptable.

(result8): (sf5) is preferable to (sf6).

Analysis: Rule 1 is inapplicable. Rule 2 applies; the sentence is good, with the analysis being NEXT-as-IO.

There is a bug in the analysis of 10.10 here: The rules stated above do not pick out the fact that this sentence is subtly bad. What seems to be happening is that while (sf5) is preferable to (sf6) (given some representation that adequately represents (sf5) and (sf6)), it is not preferable by much. Thus, one can conjecture that the semantic bias in such cases must have a certain strength before a sentence is judged to be totally acceptable, even if that bias is in support of the syntactic bias. This is a question for future investigation.

Conclusions

It is now time to summarize exactly what has been established. While the analysis presented above is built upon several assumptions, its success in accounting for the range of judgements exhibited hinges on the notions of comparative semantic tests and comparative syntactic bias. If such notions are crucial to explaining these judgements, then any processing model that fails to implement these notions in some way is necessarily an incorrect model of human language processing. As was argued above, any parser based upon the determinism hypothesis must embody mechanisms of this sort, since alternative structural possibilities must be diagnosed by some means or other.

The nature of the *syntactic* interactions involved in WH-ungapping has been previously investigated, it should be noted, by Wanner et al. [Wanner, Kaplan & Shiner 74; Wanner & Maratsos 74]. Wanner's research in this area has been done within the hypothesis-driven ATN framework using example sentences much like those presented above. (It should be noted that using ATNs as psychological models derives from [Kaplan 72].) Wanner

expresses the syntactic bias stipulated above, that NEXT is preferred over the WH-COMP as the IO, by ordering the arc which utilizes the ATN equivalent of the WH-COMP (as explained in Chapter 7) after the arc which attempts to find a new NP. He presents the results of several reaction-time experiments which are consistent with the theory that one first hypothesizes that NEXT is the IO and then, if wrong, backs up and tries the WH-head as the IO.

While a full critique of Wanner's studies is beyond the scope of this document, several comments should be made. (For critiques of Wanner's studies, see [Drescher & Hornstein 76] and [Rich 75].)

Most importantly, Wanner's incorporation of what he terms a "passive" ungapping strategy by using arc order within the ATN is simply another way to stipulate the same syntactic bias captured in the algorithm stated above, with one exception. The one difference between Wanner's expression of the stipulation and the expression of this stipulation in the algorithm above is that Wanner's formulation is more general, and therefore somewhat more interesting, if true. By encoding the choice to utilize the WH-COMP within the NP subnetwork, Wanner claims that if there is a choice between using a following NP and using the WH-COMP *in any situation at all*, that the bias is to use the NP instead.

Second, it should be noted that Wanner's ATN model captures only the syntactic bias; it gives no explanation for why that bias should sometimes be overcome by semantic biases, if the ATN's original analysis is in fact syntactically and semantically acceptable. Wanner admits that semantic

biases interact with syntactic biases to affect the preferred parsing of some sentences, as well as the reaction times to respond to various sorts of tests, but he takes this as a problem in experimental design to be overcome if reaction time confirmation of the syntactic bias is to be obtained. Furthermore, he does not discuss the problem of acceptability judgements at all.

As a concluding point, it is worth repeating the fact that semantic preference is not the sole key to the primary interpretation of the sentences investigated above. This clearly contradicts any theory that would suggest that syntax "throws up its hands" and hands the options off to semantics for resolution when there is not enough purely syntactic information for the parser to make a decision single-handedly. Of course, one can argue that the syntactic bias might be encoded in the information handed to semantics, but such a theory is no more than a shift of the analysis presented above from the status of a "syntactic" processing rule to that of a "semantic" processing rule. Such a shift might in fact be correct, but it is necessary to argue that such a shift is for some reason preferable to the "syntactic" theory presented above.

For the same reasons, these findings also contradict theories that argue that semantic processing is primary, in that syntactic processing is used only when some sort of direct semantic analysis is confused, or else is used only a post-filter for a proposed semantic analysis. *A fortiori*, these findings also contradict theories (e.g. the general framework proposed by Schank and Riesbeck [Schank 75]) which argue that there is no syntactic processing, only

semantic processing. Such theories are untenable in the face of the evidence presented above.

CHAPTER 11

CONCLUSIONS

Many different topics have been discussed in the preceding pages - among them linguistic universals, psychological phenomena, and the nature of syntactic/semantic interaction. All have shared one common property, that significant insights into each topic have followed from the Determinism Hypothesis and the structure of the grammar interpreter. Because of the range of the topics investigated in the preceding pages, it would seem to be useful to include a brief, explicit summary of the arguments presented in earlier chapters so that the reader might be better able to see the forest after having spent much time looking at the trees.

Before presenting this summary, however, I would like to make one point explicit that has been implicit in much of the discussion presented above: that a performance theory of syntactic recognition can have important implications for a competence theory of grammar. I will consider this latter point at some length, and then return to present a summary of this research, followed by suggestions for future research.

Chomsky's Constraints: Competence or Performance Phenomena?

In this section, I will discuss some implications for the theory of generative grammar of the account of the phenomena underlying those of Chomsky's constraints discussed in Chapters 6 and 7.

It is important to realize that the constraints on rules of grammar proposed by Chomsky, like the earlier constraints of Ross and others, are stipulations, plain and simple. These constraints are special in that they are claimed to hold for all possible human languages, and - if these claims of universality are true - are in this sense general principles of language. Thus, these constraints explain the structure of particular languages in that whatever phenomena follow from these constraints in a particular language follow, by definition, from general principles. Nevertheless, these constraints themselves remain stipulations, stipulations which are universal in scope, but stipulations none the less.

This fact, in and of itself, is not necessarily problematic. As Chomsky has stated [Chomsky 75a], the explanation for these constraints themselves might fall out not from any science of language, but rather from biological or evolutionary considerations. Underlying all theories are facts which cannot be explained, but which are taken as axiomatic; they are justified not by appeal to deeper principles, but by appeal to (a) the size of the body of empirical data that the resultant theory explains and (b) the simplicity of the theory which derives from these axioms. Some of these axioms might themselves be explained in terms of some broader theory, but if this reduction does not shed light on the original theory itself, then it is appropriate to take them as axiomatic, relative to that theory.

There is, however, always the possibility that some of these axioms can themselves be derived from broader theories, and that this derivation will

shed light on the original theory as well.

I believe that this research makes this possibility plausible, perhaps for the first time, and furthermore, that the performance model of syntactic recognition presented in this paper already sheds considerable light upon the constraints of generative grammar. It does so by providing explanations for these phenomena within the context of a performance theory, linking them to properties of the grammar interpreter some of which - crucially - are motivated by the Determinism Hypothesis. To the extent that these phenomena follow from the Determinism Hypothesis, they are motivated by a constraint which can itself be motivated by extremely plausible criteria of computational simplicity for which psychological verification can be sought. I hasten to add that the link between the explanations provided and the Determinism Hypothesis is somewhat indirect, and to this extent my claims must be correspondingly modest.

To say the same thing in a different way, this research provides some evidence that the constraints of competence linguistics are epiphenomenal, in a sense, thereby explaining the seemingly arbitrary nature of the constraints. Specifically, this research suggests that Chomsky's constraints are the result of projecting upon the competence domain performance phenomena that are straightforwardly motivated within this latter domain. If this is true (and I have demonstrated only the plausibility of this hypothesis), then an overall theory of language - one which includes both competence and performance - will classify the constraints as performance phenomena; by cutting the pie in this way, the overall theory

is simpler than otherwise.

A generative grammarian might respond by saying that if this is true, then it simply implies that these constraints are not in fact part of the theory of (competence) grammar at all, that a sentence that violates a constraint is not unacceptable grammatically, but rather unacceptable due to performance considerations. This statement is true (given the generative grammarian's restricted use of the word "grammar" to apply only to competence phenomena), but it misses the central point: *that unless the generative grammarian extends his interest to include performance models as well as competence models, he cannot be sure that what he believes to be a competence phenomena cannot be more elegantly explained within the domain of performance.* Without considering the alternatives, there is no way to know *a priori* what formulation of a phenomena is the simplest. (This point is also made by Jackendoff [Jackendoff 74a], in his investigation of sentences like "Herbert placed the blame on John for the accident.", where it would seem that "the blame for the accident" should be a constituent in underlying structure.) Jackendoff points out that though it is possible to formulate a purely grammatical solution to this problem, a solution which accounts for such sentences within the semantic component will be the simplest overall. As he points out, one cannot relegate a phenomenon to one component or another without considering the effect of this decision upon the complexity of the entire system.)

In terms of this argument, it is ultimately irrelevant that I have not conclusively shown that Chomsky's constraints are best explained within the

performance domain; just making this possibility plausible is enough. For once the possibility is raised that what seem to be competence phenomena at first blush might in fact best be accounted for within the domain of performance, then it is necessary to consider both possibilities, for otherwise how will one know? And from this it follows that the generative grammarian cannot focus his attention entirely on competence models; he cannot presuppose that a phenomenon falls within the domain he has defined for himself.

In Summary

I now turn to a summary of the results of this paper.

This summary will focus on those features of the grammar interpreter which are key to capturing the results presented in this paper. It will attempt to convey exactly which aspects of the grammar interpreter lie at the heart of each of the results established in this paper. This point of view should convey exactly which aspects of PARSIFAL are unique to this parser, and exactly why they are important.

The central idea behind this research is the Determinism Hypothesis, the thesis that the syntax of any natural language can be parsed by a mechanism which operates "strictly deterministically" in that it does not simulate a non-deterministic machine. I have attempted to argue for this thesis only indirectly, by assuming that the Determinism Hypothesis is true, and then seeing what follows from this assumption. If significant insights into the nature of natural language follow from this assumption, and I

believe that I have shown that they do, then these insights provide evidence, albeit indirect evidence, for the Determinism Hypothesis.

In fact, most of the arguments presented are one step removed from the hypothesis itself.

In general, I do not show that the results of this paper follow directly from the Determinism Hypothesis, but rather follow from the structure of the grammar interpreter, whose structure in turn is motivated by the Determinism Hypothesis. The structure of PARSIFAL reflects three general principles that follow from the Determinism Hypothesis in conjunction with an examination of the syntax of natural language: that any strictly deterministic parser

-must be at least partially data driven, but

-must reflect expectations that follow from the partial structures built up during the parsing process; and

-must have some sort of restricted "look-ahead" facility.

PARSIFAL reflects these principles rather directly. The grammar is made up of pattern/action rules, allowing the parser to be data directed. These rules themselves are clustered in packets, which can be activated and deactivated to reflect global expectations. The grammar interpreter's constituent buffer gives it a small window through which to view the input string, allowing restricted look-ahead.

The grammar interpreter also includes a stack of active nodes which captures the recursive aspects of the syntax of natural language. This stack

is unusual in that the interpreter can access not only the bottom of the stack but also the cyclic node nearest the bottom. Empirically, this seems to eliminate any need for a primitive that can access the father of a given node; indeed, the lack of such a primitive, which restricts the parser from accessing any node which is not a daughter of the current cyclic node, contributes in part to many of the results derived in this paper. This limitation, which I will call the "Clausemate Limitation" in what follows below, leads directly to one result: it explains the behavior which is stipulated within the framework of competence linguistics by Ross's Complex NP Constraint for the case of relative clauses. (As I pointed out in Chapter 6, this lack of a primitive within the interpreter model is *not* a constraint as the term is typically used in linguistic theory; thus the use of the word "limitation".)

Of the structures that make up the grammar interpreter, however, it is the constituent buffer which is most central to the results that are presented in this document. The most important of these results follow from the sorts of grammar operations that the buffer makes feasible; others follow from the limitations that its fixed length imposes. All however, hinge crucially upon the existence of the buffer. This data structure, I submit, is the primary source of the power of the parser.

For example:

-Because insertion of specific lexical items into the buffer is possible, one four-line rule of grammar can capture the difference between

imperatives and the "unmarked" constituent order of declarative sentences. The imperative rule is formulated so that it inserts the word "you" into the buffer, taking advantage of the fact that a full NP will be produced from the pronoun by the attention shifting mechanism. After this occurs, the buffer will contain an NP followed by a verb, the unmarked declarative order.

-Because the buffer automatically compacts upon the attachment of the constituents that it contains, the parsing of a yes/no question and the related declarative will differ in one rule of grammar, with the key difference restricted to the rule patterns and one line of the rules' actions. The yes/no question rule explicitly states only that the NP in the second buffer cell should be attached as the subject of the clause. Because the buffer will then compact, auxiliary parsing rules that expect the terminals of the verb cluster to be contiguous will then apply without need for modification.

-Because the buffer provides a three-constituent window on the structure of a clause, diagnostic rules can be formulated that allow the differential diagnosis of pairs of constructions that would seem to be indistinguishable by a deterministic parser without such a buffer. Thus, evidence that would seem to imply that natural language must be parsed non-deterministically can be explained away.

-Because the buffer can only contain a limited number of constituents, the power of these diagnostic rules is limited, leading to a principled

distinction between sentences which are perceived as garden paths and those which are not. This explanation for why certain sentences cause garden paths is consistent with the informal experiments presented in Chapter 9. Furthermore, this theory led to the counter-intuitive prediction that as simple a sentence as "Have the packages delivered tomorrow." would cause a garden path, a prediction which was confirmed by informal experiment.

In short, this one mechanism not only allows the formulation of rules of syntax that elegantly capture linguistic generalizations; it also provides the underpinnings for a psychological theory that seems to have high initial plausibility.

Another important source of power is the use of traces, especially in conjunction with the use of the buffer. Especially important is the fact that a trace can be dropped into the buffer, thereby indicating its underlying position in a factorization of the terminal string without specifying its position in the underlying tree. From this follows:

-a simple formulation of passive which accounts for the phenomenon of "raising". The essence of the passive rule - create a trace, bind it to the subject of the current s, drop it into the buffer - is noteworthy in its simplicity. Again, the availability of the buffer yields a very simple solution to a seemingly complex linguistic phenomenon.

-an explanation for the phenomena which underlie Chomsky's Specified

Subject Constraint and Subjacency Principle, in conjunction with the Clausemate Limitation. It is most interesting that the simple formulation of Passive presented here - perhaps the simplest possible formulation of this rule within the framework of PARSIFAL - behaves exactly as if these constraints were true; i.e. that the formulation presented here, by itself and with no extraneous stipulations, leads to the behavior that these constraints attempt to specify. It is also true that it is impossible to formulate a passive rule that would behave otherwise, given the Clausemate Limitation and the Left-to-Right Constraint presented in Chapter 6.

The Determinism Hypothesis itself does figure directly in two of the results presented in this paper. The Determinism Hypothesis is crucial in the argument for semantic/syntactic interaction presented in Chapter 10. Furthermore, in conjunction with the Clausemate Limitation, it leads to behavior in the grammar interpreter which accounts for the phenomenon characterized by the Complex NP Constraint for noun complements.

Directions for Future Work

Throughout this paper, I have tried to point out the limitations of this research, as well as important issues that have remained untouched. Rather than repeat these limitations and issues at length, I will provide a brief list here of those issues that have been raised by this paper that deserve further study; the reader is referred to Chapter 1 for a listing of important questions that this research has not addressed.

Three issues in particular call for further study. Further work is called for:

-to study the entire question of semantic/syntactic interaction. What other sorts of interactions are there? Is there a natural limit to the sorts of interactions necessary? What is the bandwidth of these interactions?

-to extend the grammar. What is the range of syntactic phenomena for which grammar rules can be found that share the simplicity of the rules presented in this paper?

-to carefully test the explanation for garden path sentences proposed in Chapter 9. Will this model account for a wide range of garden paths under careful scrutiny?

BIBLIOGRAPHY

Aho, A. V. and J. Ullman [1972] *The Theory of Parsing, Translation and Compiling*, Vol. 1, Prentice-Hall, Englewood Cliffs, N.J.

Akmajian, A. and F. Heny [1975] *An Introduction to the Principles of Transformational Syntax*, MIT Press, Cambridge, Mass.

Bach, E. and G. Horn [1976] "Remarks on 'Conditions on Transformations'", *Linguistic Inquiry* 7:265.

Bever, T. G. [1970] "The Cognitive Basis for Linguistic Structures", in J. R. Hayes, ed., *Cognition and the Development of Language*, Wiley and Sons, N.Y.

Bresnan, J. W. [1973] "Syntax of the Comparative Clause Construction in English", *Linguistic Inquiry* 4:275.

Bresnan, J. W. [1976] "Evidence for a Theory of Unbounded Transformations", *Linguistic Analysis* 2:353.

Bullwinkle, C. [1977] "The Semantic Component of PAL: The Personal Assistant Language Understanding Program," Working Paper 141, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Chomsky, N. [1957] *Syntactic Structures*, Mouton, The Hague.

Chomsky, N. [1965] *Aspects of the Theory of Syntax*, MIT Press, Cambridge, Mass.

Chomsky, N. [1970] "Remarks on Nominalization", in [Jacobs & Rosenbaum 70]

Chomsky, N. [1971] "Deep Structure, Surface Structure, and Semantic Interpretation", in [Steinberg & Jakobovits 71].

Chomsky, N. [1972] *Studies on Semantics in Generative Grammar*, Mouton, The Hague.

Chomsky, N. [1973] "Conditions on Transformations", in S. Anderson and P. Kiparsky, eds., *A Festschrift for Morris Halle*, Holt, Rinehart and Winston, N.Y.

Chomsky, N. [1975] *Reflections on Language*, Pantheon, N.Y.

Chomsky, N. [1975a] Lectures, 23.755, Linguistic Structures, Fall term, MIT.

Chomsky, N. [1976] "Conditions on Rules of Grammar", *Linguistic Analysis* 2:303.

Chomsky, N. [to appear] "On Wh-Movement", in A. Akmajian, P. Culicover,

and T. Wasow, eds., *Formal Syntax*, Academic Press, N.Y.

Chomsky, N. and H. Lasnik [1977] "Filters and Control", *Linguistic Inquiry* 8:425.

Cowper, E. A. [1976] *Constraints on Sentence Complexity: A Model for Syntactic Processing*, unpublished Ph.D. thesis, Brown University.

Dresher, B. E. and N. Hornstein [1976] "On Some Supposed Contributions of Artificial Intelligence to The Scientific Study of Language", *Cognition* 4:321.

Emonds, J. E. [1970] *Root and Structure Preserving Transformations*, unpublished Ph.D. thesis., MIT.

Emonds, J. E. [1976] "Evidence that Indirect Object Movement is a Structure-Preserving Transformation", *Foundations of Language* 8:547.

Fahlman, S. E. [1973] "A Hypothesis-Frame System for Recognition Problems", Working Paper 57, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Fiengo, R. W. [1974] *Semantic Conditions on Surface Structure*, unpublished Ph.D. thesis, MIT.

Fillmore, C. J. [1967] "The Grammar of Hitting and Breaking" in [Jacobs & Rosenbaum 70]

Fillmore, C. J. [1968] "The Case for Case" in *Universals in Linguistic Theory*, E. Bach and R. T. Harms, eds., Holt, Rinehart, and Winston, N.Y.

Freidin, R. [1977] Notes, Informal Seminar, MIT Linguistics Dept., Spring term.

Goldstein, I. P. [1974] *Understanding Simple Picture Programs*, AI-TR 294, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Goldstein, I. P. and R. B. Roberts [1977] "NUDGE, A Knowledge-Based Scheduling Program," Memo 405, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Gruber, J. S. [1965] *Studies in Lexical Relations*, unpublished Ph.D. thesis, MIT.

Halliday, M. A. K. [1967] "Notes on Transitivity and Theme in English", Part 1, *Journal of Linguistics* 3:37.

Hill, J. M. [1972] *Backup Strategies for Resolving Ambiguity in Natural Language Processing*, unpublished Masters thesis, MIT.

Hudson, R. A. [1971] *English Complex Sentences: An Introduction to*

Systemic Grammar, North-Holland, Amsterdam.

JPL [1973] *Publications of the Jet Propulsion Laboratory: January through December 1972*, Bibliography 39-14, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California.

Jackendoff, R. S. [1972] *Semantic Interpretation in Generative Grammar*, MIT Press, Cambridge, Mass..

Jackendoff, R. S. [1974] *Introduction to the X-bar Convention*, unpublished paper, available from Indiana University Linguistics Club, Bloomington, Indiana.

Jackendoff, R. S. [1974a] "A Deep Structure Projection Rule", *Linguistic Inquiry* 5:481.

Jacobs, R. A. and P. S. Rosenbaum, eds. [1970] *Readings in English Transformational Grammar*, Ginn, Waltham, Mass.

Kaplan, R. M. [1972] "Augmented Transition Networks as Psychological Models of Sentence Comprehension", *Artificial Intelligence* 3:77.

Katz, J. J. and J. A. Fodor [1964] "The Structure of a Semantic Theory", in J. A. Fodor and J. J. Katz, eds., *The Structure of Language*, Prentice-Hall, Englewood Cliffs, N.J.

Kay, M. [1973] "The MIND system", in [Rustin 73].

Knuth, D. E. [1968] *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, Mass.

Kuno, S. and Oettinger, A. G. [1962] "A Multiple Path Syntactic Analyzer", in *Information Processing 1962*, North-Holland, Amsterdam.

Levin, B. [1977] "Mapping Sentences to Case Frames", Working Paper 143, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Maclay, H. [1971] "Overview of Linguistics", in [Steinberg & Jakobovits 71].

Marcus, M. P. [1974] "Wait-and-See Strategies for Parsing Natural Language", Working Paper 75, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Marcus, M. P. [1975] "Diagnosis as a Notion of Grammar", in R. C. Schank and B. Nash-Webber, eds., *Advance Papers of the Workshop on Theoretical Issues in Natural Language Processing*.

Marcus, M. P. [1976] "A Design for a Parser for English" in *Proceedings of the ACM Conference*, 1976.

Moon, D. A. [1974] *The MACLISP Reference Manual*, Project MAC, MIT,

Cambridge, Mass.

Newell, A. and H.A. Simon [1972] *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, N.J.

Peters, P. S. and R. W. Ritchie [1973] "On the Generative Power of Transformational Grammars, *Information Sciences* 6:49.

Petrick, S. R. [1974] "Review of Winograd, 'A Procedural Model of Natural Language Understanding'", *Computing Reviews* 15:272.

Postal, P. M. [1974] *On Raising*, MIT Press, Cambridge, Mass.

Pratt, V. R. [1973] "Top-Down Operator Precedence", in the proceedings of *The SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Boston, Mass.

Pratt, V. R. [1975] "LINGOL - A Progress Report" in *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Vol. 1, available from the MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Pratt, V. R. [1976] "CGOL - An Alternative External Representation For LISP Users", Working Paper 121, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Rich, C. [1975] "On the Psychological Reality of Augmented Transition Network Models of Sentence Comprehension", unpublished paper, MIT Artificial Intelligence Laboratory, Cambridge, Mass.

Ross, J. R. [1967] *Constraints on Variables in Syntax*, unpublished Ph.D. thesis, MIT. Excerpted in G. Harmon, ed. [1974] *On Noam Chomsky*, Anchor, N.Y.

Russell, S. W. [1972] *Semantic Categories of Nominals for Conceptual Dependence Analysis of Natural Language*, Stanford Artificial Intelligence Memo 172, Computer Science Department, Stanford University, Palo Alto, California.

Rustin R., ed. [1973] *Natural Language Processing*, Algorithmics Press, N.Y.

Sager, N. [1973] "The String Parser for Scientific Literature", in [Rustin 73].

Schank, R. C., ed. [1975] *Conceptual Information Processing*, North-Holland, Amsterdam.

Simon, H.A. [1969] *The Sciences of the Artificial*, MIT Press, Cambridge, Mass.

Steinberg, D. D. and L. A. Jakobovits, eds. [1971] *Semantics*, Cambridge University Press, N.Y.

Stockwell, R. P., P. Schachter, and B. H. Partee [1973] *The Syntactic Structures of English*, Holt, Rinehart, and Winston, N.Y.

Sussman, G. J. and A. L. Brown [1974] *Localization of Failures in Radio Circuits - A Study in Causal and Teleological Reasoning*, Memo 319, MIT Artificial Intelligence Laboratory.

Wanner, E. and M. Maratsos [1974] "An Augmented Transition Network Model of Relative Clause Comprehension", unpublished paper, Harvard University, Cambridge, Mass.

Wanner, E., Kaplan R. and S. Shiner [1974] "Garden Paths in Relative Clauses", unpublished paper, Harvard University, Cambridge, Mass.

Waterman, D. A. and A. Newell [1973] "PAS-II: An Interactive Task-Free Version of an Automatic Protocol Analysis System" in *Advance Papers of the Third International Joint Conference on Artificial Intelligence.*

Winograd, T. [1971] *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*, Project MAC-TR 84, MIT, Cambridge, Mass.

Woods, W. A. [1970] "Transition Network Grammars for Natural Language Analysis", *Communications of the ACM* 13:591.

Woods, W. A. [1972] *The Lunar Sciences Natural Language Information System*, BBN Report No. 2378, Bolt, Beranek and Newman, Cambridge, Mass.

Woods, W. A. [1973] "An Experimental Parsing System for Transition Network Grammars", in [Rustin 73].

APPENDIX A

AN APPROACH TO NOUN-NOUN MODIFICATION

This appendix briefly sketches a possible approach for diagnosing the internal structure of noun-noun modifier strings using interacting syntactic and semantic biases, as discussed in Chapter 9. Following Chapter 10, this approach presupposes that a buffer "window" of three constituents is sufficient for analyzing noun-noun modifier strings deterministically.

This discussion is relegated to an appendix because the diagnostic for noun-noun modifiers requires fairly semantic subtle judgements, and therefore the judgements that must be assumed to test this approach on specific examples are necessarily suspect. This is especially true because semantic judgements of this type seem to vary widely between individuals. Because of this, it is somewhat difficult to assess the extent to which this approach is adequate and in exactly what ways it fails to be a general solution.

Despite these problems, I include this discussion because of the intrinsic interest of the problem of parsing noun-noun modifiers. While the semantics of noun-noun modifier strings has been previously investigated (see, for example, [Russell 72]), no parser that I know of has attempted to resolve the internal structure of noun-noun modifier strings. Typically, where this phenomenon has been handled by a natural language parser, each noun in the string of modifiers has been parsed as an independent daughter

of some common node. This ignores the fact that there is much internal structure to a noun-noun modifier string. For example, the noun string shown in Figure A.1.a, taken from Winograd [Winograd 71], has the internal structure shown in A.1.b, where each parenthesized entity, each listed in A.1.c, can be considered to be a "complex noun".

(a) water meter cover adjustment screw
(b) [[[water meter] cover] [adjustment screw]]
(c) 1) water meter
(c) 2) water meter cover
(c) 3) adjustment screw

Figure A.1 - The internal structure of a noun-noun modifier string.

In the discussion below, I will ignore such issues as the fact that different sorts of noun-noun modification seem to have different syntactic properties, as evidenced by the difference in intonation given to the two noun-noun pairs "maternity dress" and "paternity suit". I also ignore the fact that not all contiguous nouns are part of one noun-noun modifier string, as in "The cat was hungry, so I gave the cat food.". Clearly, these issues must be dealt with, but the problem attacked here is a prerequisite for any solution to these further problems.

A Rule to Parse Noun-Noun Modifier Strings

Given the caveats stated above, I will now present a diagnostic procedure that will analyze arbitrarily long noun-noun modifier strings, present a few examples of its operation, and then show a few ways in which it is deficient. (Because I know of no technique which can to answer the necessary semantic questions, this procedure has not been

implemented.)

The procedure is based upon two crucial assumptions, the first of which is that a semantic component can decide upon the relative "goodness" of two possible noun-noun modifier pairs. Thus, it is assumed that a semantic component can decide, for example, that "adjustment screw" in the string shown in A.1 is a better noun-noun pair than "cover adjustment", although both pairs are acceptable.

It is not clear, I should admit, whether this semantic is adequate for the diagnostic presented here, or whether it must further take into account the remaining noun, somehow incorporating the knowledge that this noun may modify or be modified by whatever noun-noun pair is constructed first. I will assume the simpler form in what follows below, although there is some evidence that this assumption is wrong. How any of these judgements can be made, of course, is very much an open question.

The second assumption is that an arbitrarily long modifier string can be analyzed by iteratively examining only the three leftmost nouns in the modifier string, i.e. that local decisions are sufficient to accurately determine the global structure of the modifier string. The diagnostic rule will operate by reducing the leftmost three nouns in a modifier string, N_1, N_2, and N_3, to a string of two nouns either by creating a new complex noun from N_1 and N_2 (which I will denote as $[N_1\ N_2]$) or from N_2 and N_3 (the complex noun $[N_2\ N_3]$). If there are at least three nouns remaining, including the newly formed complex noun, the diagnostic rule will be triggered again. Otherwise,

an additional rule will construct the remaining two nouns into a complex noun if the resulting complex noun is semantically acceptable.

The essence of the diagnostic is as follows: Given a string of three consecutive nouns N_1, N_2, and N_3, if both noun-noun pairs [N_1 N_2] and [N_2 N_3] are semantically acceptable, there is a syntactic bias for the first two nouns to form a complex noun that may modify the third noun, yielding the structure shown in Figure A.2.a below, rather than for the second and third to form a complex noun that may be modified by the first, giving the structure shown in A.2.b. This syntactic bias can be overcome, however, if the noun-noun pair [N_2 N_3] is semantically preferred to the pair [N_1 N_2]. In this later case, the analysis is as shown in A.2.b. In short, the syntactic bias is the default, but it is easily overriden by semantic preference. (If both pairs are semantically unacceptable, then no complex noun is formed.)

(a) [N_1 N_2] N_3
(b) N_1 [N_2 N_3]

Figure A.2 - There is a syntactic bias for (a) over (b).

A careful statement of the algorithm is given in Figure A.3 below.

Given two nouns N_1 and N_2 in the first two buffer cells and a non-noun in the third buffer cell:

If $[N_1\ N_2]$ is semantically acceptable, then build $[N_1\ N_2]$.

Given three nouns N_1, N_2, N_3 in the first three buffer cells:

1) If either $[N_1\ N_2]$ or $[N_2\ N_3]$ is not semantically acceptable, then build the alternative structure; else...

2) If $[N_2\ N_3]$ is semantically preferable to $[N_1\ N_2]$, then build $[N_2\ N_3]$;

3) Otherwise, build $[N_1\ N_2]$.

Figure A.3 - The algorithm for constructing noun-noun pairs.

Let us see how this algorithm analyses the examples presented in Figure A.4 below. (The first example is Winograd's; the second and third were found in JPL's bibliography of 1972 publications [JPL 73].) The semantic judgements in these examples are strictly my own; other people's judgements may differ. If the reader's judgements differ from the judgements given here, the reader should apply the analysis algorithm for himself, substituting his own judgements, and see if the resulting analysis is consistent with his own intuition of what the underlying structure should be. What is crucial here is not the specific judgements given, nor the particular analyses that result from those judgements, but rather the relationship between the relevant semantic judgements and the resulting analysis. (The symbol ">" should be read "is semantically better than".

The Buffer	$[N_2\ N_3] > [N_1\ N_2]$?
1) water | meter | cover |	no
2) [water meter] | cover | adjustment	no
3) [[water meter] cover] | adjustment |screw	yes
4) [[water meter] cover] | [adjustment screw]	(Rule for 2 nouns applies)
5) [[[water meter] cover][adjustment screw]]	(Finished)

(a) - Water Meter Cover Adjustment Screw

The Buffer	$[N_2\ N_3] > [N_1\ N_2]$?
1) ion | thruster | performance	no
2) [ion thruster] | performance | calibration	yes
3) [ion thruster] | [performance calibration]	(2-noun rule applies)
4) [[ion thruster][performance calibration]	(Finished)

(b) - Ion Thruster Performance Calibration

The Buffer	$[N_2\ N_3] > [N_1\ N_2]$?
1) boron | epoxy | rocket	no
2) [boron epoxy] | rocket | motor	yes
3) [boron epoxy] | [rocket motor] | chambers	yes
4) [boron epoxy] | [[rocket motor] chambers]	(2-noun rule applies)
5) [[boron epoxy][[rocket motor] chambers]]	(Finished)

(c) - Boron Epoxy Rocket Motor Chambers

Figure A.4 - Three example analyses.

The first example, shown in A.4.a, demonstrates the analysis of the noun-noun string "water meter cover adjustment screw". The analysis begins with the leftmost three nouns entering the buffer, as shown in A.4.a1. Both $[N_1\ N_2]$, "water meter", and $[N_2\ N_3]$, "meter cover", are semantically acceptable, so step (1) in the algorithm does not apply. The pair "meter cover" is not semantically preferable to the pair "water meter", so (2) does not apply. Step (3) thus applies by default, building the complex noun [water meter]. The noun "adjustment" now enters the buffer, as shown in

A.4.a2, and the algorithm is retriggered. Again, [N_1 N_2], "water meter cover", and [N_2 N_3], "water meter adjustment" are acceptable, and [N_2 N_3] is not preferable to [N_1 N_2], so step (3) applies again, building [[water meter] cover]. The noun "screw" now enters the buffer, and the algorithm is triggered again. This time [N_2 N_3], "adjustment screw", *is* semantically preferable to [N_1 N_2], "water meter cover adjustment", so step (2) of the algorithm applies, constructing [N_2 N_3]. There are now two complex nouns in the buffer, as shown in Figure A.4.a4, with no following noun, so the 2-noun rule applies, and because [N_1 N_2] is semantically acceptable, the structure shown in A.4.a5 results, concluding the example.

Because the analyses of A.4.b and A.4.c are both similar to the above analysis and should be self-explanatory, the reader should be able to work through these last two examples without much trouble. Again, some of the semantic judgements given may be questionable for the reader, in which case the reader should complete the analysis using his own judgements.

Coverage

The noun-noun algorithm was tested by hand-simulating it on the list of examples presented below in Figure A.1; all the examples were found in the JPL 1972 bibliography. While I will not present the analyses here, the algorithm seems to correctly analyze all these examples included in A.1.a.

a) ion thruster performance calibration
b) ground communications facility operations chief
c) two-station interferometer analog input channel
d) spacecraft ion beam noise effects
e) water flow vizualization studies
f) telecommunications systems design techniques handbook
g) antenna drive system performance evaluation
h) subcarrier recording equipment implementation
i) boron epoxy rocket motor chambers
j) filament winding fabrication difficulties

Figure A.1 - Some modifier strings parsable by the noun-noun algorithm.

The algorithm correctly analyzes these examples as well as all the others I could find in the JPL bibliography, with one exception; the following noun-noun string is not correctly analyzed by this algorithm:

1970 balloon flight solar cell standardization program

The essence of the difficulty is that the initial noun "1970" must modify the entire remaining string; the correct analysis of this string is

[1970 [[balloon flight][[solar-cell standardization] program]]]

where "1970" modifies the rest of the noun-noun string. (I take "solar cell" to be a single noun here.) If the algorithm is to successfully analyze such a string, the first noun must be held in the buffer until the remaining structure has been completed. This will leave only two cells available for succeeding nouns, which means that the remainder of the string should be strictly left-branching, since at each application of the algorithm, the complex noun pair [N_2 N_3] must be constructed if another noun is to enter the buffer. While it is possible that "1970" is not part of the noun-noun modifier string, my intuition is that it is, and that this is a counterexample to the algorithm proposed above. I will not attempt to patch the algorithm, however, until more counterexamples can be found and examined.

APPENDIX B

THE GRAMMAR LANGUAGE

This appendix provides a description of PIDGIN, PARSIFAL's grammar input notation.

PIDGIN is a formal language that reads much like English, although its syntax is very restrictive and completely artificial. There are several advantages to using an English-like formal language as the grammar language. By putting the grammar into an English-like notation, the grammar can be intrinsically self-documenting. Because it does read much like English, PIDGIN code should be highly perspicuous both to those with computational backgrounds and to those linguists and psychologists not familiar with LISP syntax; in short, it attempts to be a Perspicuous, Intrinsically Documented Grammar Input Notation. The language also conveniently enforces a strict set of constraints on the form of legal grammar rules. By the simple expedient of not including the standard LISP primitives of CAR, CDR, CONS, etc. in the grammar language, we can strictly limit the sorts of operations that can be used within the grammar.

PIDGIN is internally converted to LISP by a parser which is based on Pratt's notion of top-down operator precedence parsing [Pratt 73], which provides a powerful framework for constructing grammars for simple token-based deterministic computer languages. Indeed, much of the code which underlies the PIDGIN parser derives from Pratt's implementation of his CGOL

language [Pratt 76]. It is worthy of mention that the implementation of CGOL, a language designed as an alternate external representation for general LISP code, was very easily adapted for PIDGIN, a language with a very different purpose and external appearance.

(It should also be noted that the cost of using this non-LISP syntax is not very high. The top-down operator precedence parser is very fast and efficient, and by using the read macro facility of MACLISP [Moon 74], one can read in PIDGIN code through the normal LISP reader with little difficulty. Indeed, it is easy to get the MACLISP compiler to directly compile PIDGIN code into machine language instructions.)

Each grammar rule in PIDGIN is of the general form:

{Rule <name> (Priority: <numeric priority>) in <packet list>
 <pattern> --> <action>}

Each rule is delimited by "{ }", by curly brackets, and includes a rule name, an optional priority, a list of the packets the rule is part of, and a pattern and an action. Rule and packet names can be any legal LISP atom containing at least one non-numeric character, but not spaces, tabs, carriage returns or any of the special characters "| [ ] / { } ( ) ' ; , = | % ! .". These special characters delimit names, and need not be separated from them. The priority can be any positive integer, where 1 is the highest priority and 1000 the lowest priority. If no priority is specified for a rule, it is assigned a default priority of 10. The packet list consists of either one name or else a series of names separated by commas. Note that a rule can be in more than one packet; it will be active if any packet it is in is active.

The following sections present the core of PIDGIN syntax. I will first present the syntax of patterns and the subset of PIDGIN commands legal in patterns, then the remainder of basic PIDGIN command which are of use primarily in rule actions.

Patterns

Patterns are written in PIDGIN as a list of descriptions, where each description is enclosed in square brackets. The first three descriptions in a pattern are taken by default to refer to the constituents in the first, second, and third buffer positions, referred to in PIDGIN as 1st, 2nd, and 3rd respectively. The following pattern is a typical example of a simple pattern, which looks for an auxiliary verb followed by an NP:

[* is verb, auxverb] [* is NP]

This pattern will be satisfied if 1st, the constituent in the first buffer position, has the features *verb* and *auxverb*, and 2nd, the constituent in the second buffer position, has the feature *NP*. This pattern does not impose any conditions on 3rd; indeed, there need be no constituent in the third position at all for this pattern to match. The predicate "is" tests that the node named by its first argument is labelled with all the features in the feature list that follows the predicate. "*" within a description refers to the node that the description refers to.

Since descriptions so often test for the positive presence of sets of features, PIDGIN allows the notation "= <feature list>" in patterns as a shorthand for "* is <feature list>", where <feature list> is either a single

feature, or a list of features separated by commas.

A description that refers to C, the current active node, begins with the label "** C" followed by a semi-colon. The following pattern will match if C has the features *S* and *ynquest* (yes-no-question) and 1st has the feature *NP*:

```
[=NP] [**C;  =s, ynquest]
```

PIDGIN provides several other predicates besides "is" which test for the presence or absence of sets of features of a node. These predicates are:

<node> is (all of) <feature list>

<node> is any of <feature list>

<node> is not (all of) <feature list>

<node> is none of <feature list>
OR
<node> is not any of <feature list>

Note that the last two predicates above are synonymous constructs; PIDGIN tries to provide synonymous constructs whenever there seems to be more than one simple natural way to express an operation.

In this figure and below, I will use angle brackets to indicate meta-syntactic strings. Words surrounded by parentheses are optional; they make be used or not, at the option of the grammar writer. Also, PIDGIN totally ignores the tokens "the", "a" and "an"; these can be used freely in the grammar to increase readability and will be used in examples below without further comment.

These predicates can be combined with logical operators to form arbitrary boolean feature tests. A fairly complex test might look like:

B.1 1st is not an auxverb or 3rd is a verb.

The logical operations allowed, and their corresponding PIDGIN expressions are:

conjunction: <pred1> and <pred2> and ... and <predn>

OR (in patterns)

<pred1>; <pred2>; ....; <predn>

disjunction: <pred1> or <pred2> or ... or <predn>

negation: it isn't true that <pred>

A precedence ordering is imposed on these operators to allow proper scoping in most cases without parentheses. The precedence order is:

"it isn't true that" > "and" > "or" > ";".

Thus, for example B.2a below has as its meaning B.2b:

B.2a <p1> or <p2> and it isn't true that <p3>; <p5> or <p6>

B.2a {<p1> $\vee$ [<p2> $\wedge$ $\neg$<p3>]} $\wedge$ {<p5> $\vee$ <p6>}

The idea behind the semi-colon conjunctions is that semi-colons separate the set of predications that make up each pattern description; all of these predicates must be true to satisfy the description. Note that the "it isn't true that" operator takes narrow scope. Thus B.3 is equivalent to B.1 above:

B.3 it isn't true that 1st is an auxverb or 3rd is a verb.

If the other sense is desired, parentheses can be used to specify the proper scopes. Thus, to get the wide scope negation sense of B.3 one would say

it isn't true that (1st is an auxverb or 3rd is a verb).

While parentheses *can* be used to change the "natural" scopes of these

operators, there really shouldn't be much need to do so. This is especially true given that there are two operators for logical conjunction in patterns, and the wide range of feature testing predicates.

To complete the subset of PIDGIN used in pattern descriptions, the phrase below is a tree traversing operator which can be used to name the sons of a node:

<type> of <fathernode>

This phrase returns as its value the node attached to <fathernode> as <type>, or the rightmost node attached as <type> if there is more than one. This construct is right associative, e.g. the two phrases below are equivalent:

the NOUN of the NP of the S above ^

(the NOUN of (the NP of (the S above ^)))

These expressions go up to the current S node and then down through the NP attached to the S, i.e. through its subject, to the NOUN under the NP node.

One last primitive that must be mentioned is the primitive "t", which can be thought of as a predicate which is always true. A description made up of only this primitive will match any node, but such a description in a pattern will force the matcher to wait until some constituent fills the corresponding buffer position before the pattern will match. This is important because each of the parameters 1st, 2nd and 3rd has a value within the action of a rule only if the pattern of that rule has a description

for the corresponding buffer position.

A few example patterns should help to make the use of all of this syntax clear.

Here is a pattern from a rule intended to do "passive-shift", to make the subject of a passive sentence that sentence's functional object:

[** c; * is a VP; the AUX of the S above * is passive; * is not np-preposed]

This pattern looks at the current active node (i.e. C), the current cyclic node (the S above *), and some of their daughters. It will match if C is a verb phrase, if the auxiliary attached to the current S has the feature *passive*, and C itself does not have the feature *np-preposed* (which is used to indicate that "passive-shift" has already taken place). Note that PIDGIN completely ignores the "a" in "* is a VP".

The following pattern will match against the first node in the buffer if it has the feature *pp*, i.e. it is a prepositional phrase, and its preposition is marked with the feature **of*, indicating that the root of the word under the prep node is "of":

[=pp; the prep of * is *of] [t]

This pattern will match any node in the second buffer position, but it will not match until there is some node in the second buffer position. Unless 2nd was referenced by this pattern, the action of the rule of which this is the pattern would be unable to examine this constituent.

The rule which starts infinitive complements without overt subjects

(phrases like "to buy a car") after verbs which take such a complement, e.g. the verb "want" as in "I want to buy a car", has the following pattern:

[="to] [=tnsless] [** c; the verb of * is subj-less-inf-comp]

This pattern will match if 1st is the word "to", 2nd is a tenseless verb, and C, implicitly assumed here to be a VP, has the feature *subj-less-inf-comp*, indicating that it takes subject-less infinitive complements.

The Basic Pidgin Primitives - Actions

In this section, we present the central core of PIDGIN. This section will include the PIDGIN primitives for creating and attaching nodes, and the primitives for labelling parse nodes with features. It will also include parser control primitives and PIDGIN conditional expressions. Pointers to PIDGIN commands described at length in earlier chapters will be provided.

Actions On Nodes

The basic action commands in PIDGIN operate on nodes; these commands create them, attach them, and pop them from the active node stack.

The basic command for creating new parse nodes is:

Create a (new) <type> node (labelled <feature set>)

which creates a new node of type <type>, makes it the current active node, and optionally labels it with the features in <feature set>. An alternative way of saying the same thing is:

a new <type> node (labelled <feature set>)

This construct for node creation returns the name of the newly created node

as its value. It can thus be used in composition with other primitive operations such as attachment. Note that here the token "new" is obligatory here.

One very useful grammatical operation is replacing "deleted" lexical items. For example, the same general parsing rules can be made to apply in both cases if "all the boys" can be transformed into "all of the boys" by simply putting the "of" back after the word "all". To facilitate this sort of operation, PIDGIN provides a composite command which inserts a node dominating a particular lexical item into the buffer in a specified position:

Insert the word '<lexical item>' (into the buffer) (before <position>)

This command inserts a node dominating <lexical item> into the buffer before <position>, which can be any of 1st, 2nd, or 3rd. If <position> is not specified, then the node is inserted into the buffer before 1st. Thus, the PIDGIN code to convert "all the" into "all of the", assuming all is 1st and "of" 2nd would be:

Insert the word 'of' into the buffer before 2nd.

Later rules will then process the input as if 'of' appeared in the original input string.

To attach <node> to <fathernode>, PIDGIN provides the command:

Attach <node> to <fathernode> as <type>

For the time being, <type> can be considered to be a redundant specification of the type of the new daughter node.

As a special case, the semantics of the form

Attach a new <type> node ... to C as <type>

are defined to mean: Create a new <type> node, attach it to the old current active node, and *then* make the new node the current active node. This definition solves by *fiat* the problem of whether "C" refers to the newly created node or the old current active node.

Once the parser is finished with a current active node, it must pop the active node stack. The "drop" command does this:

Drop c (into the buffer).

If C is attached to some other node, this command simply pops C from the active node stack. If C is not attached, this command pops it off the active node stack and drops it into the buffer in the 1st buffer position, shifting the other buffered constituents one to the right. Note that the optional phrase "into the buffer" carries no significance whatever to the PIDGIN reader. It can be used for the sake of perspicuity and self-documentation, if the grammar writer knows that the node about to be dropped is not attached to another node.

One last operation on nodes that is most useful is the ability to test if a node has a daughter of a given type. For instance, a grammar of English must check to see if a NP has a determiner when checking the person and number of the NP. If there is a determiner and it is "a" or "an", then the head noun must also be singular if the NP is grammatical. In PIDGIN, given that the NP in question is C, the predicate of this test will be simply

there is a DET of C and it is *a.

The "there is" predicate is of form

there is <node>

where <node> is most likely to be a complex specification for a node. The predicate is true if <node> exists and false otherwise. Since <node> is most likely a long expression, PIDGIN provides the convenience that the token "it" refers to the node found by the last "there is" predicate evaluated. If the last "there is" test failed, then "it" has as its value the LISP atom NIL. All predicates applied to NIL are false. (This device was borrowed from CGOL.)

Feature Manipulating Primitives

PIDGIN provides many primitives for manipulating features. The most basic of these add, and, for the moment, remove given features.

The command

Label <node> with (the feature(s)) <feature set>

adds the features in <feature set> to the features of <node>. Note, by the way, that the two clauses below are exactly equivalent:

Label a new S node with the features decl, major.

Create a new S node labelled decl, major.

Although I believe that the constraint that features never be removed can be enforced, PIDGIN provides the command

Remove (the feature(s)) <feature set> from <node>

as a utility for grammar formulation.

A very common operation is "transferring" some general class of features from a daughter node to its parent, labelling the parent with those

features of the daughter node that are in a given set. To facilitate this operation, PIDGIN provides the construct:

B.4 Transfer (the feature(s)) <feature set> from <node1> to <node2>.

This construct adds to the features of <node2> the intersection of the features of <node1> and <feature set>. Thus, for example, one can shift the singular-plural features *ns*, noun singular, and *npl*, noun plural, and the 1st, 2nd, and 3rd person features *n1p*, *n2p*, and *n3p* from a NOUN to its parent NP (which we'll assume is 1st) with the command:

Transfer the features ns, npl, n1p, n2p, n3p from the NOUN of 1st to 1st.

Besides specifying feature lists literally, several operations exist for manipulating feature lists themselves. These operations allow the grammar to access the feature set of a particular node, and to take the meet of two separate feature lists.

To directly access the features of a node, PIDGIN provides the phrase:

the features of <node>.

which returns the feature set of <node>. To set the features of a node, one uses the command:

Set the features of <node> to <feature set>.

To take the intersection of two feature sets, one uses the phrase:

the meet of <feature set> and <feature set>.

PIDGIN allows the use of "meet" and "feature" phrases everywhere it allows literal feature sets. Thus, B.4 above can be seen as an abbreviation for

Label <node2> with the meet of the features of <node1> and <feature set>.

Conditionals

While PIDGIN provides no control primitives allowing iteration within rules, or calls to subroutines, it does allow conditional expressions. The basic form of a conditional is

If <boolean expression> then <complex action> (else <complex action>).

The <boolean expression> can be made up of any logical composition of predicates. The <complex action>s can be either 1) primitive actions, 2) conditionals (allowing for nested conditionals) or 3) a sequence of complex actions separated by semi-colons i.e.

<complex action1>; <complex action2>;.....;<complex actionn>.

Such a complex action sequence (a LISP "PROGN") is executed, as would be expected, from left to right (or, if you prefer, top to bottom).

It is important to note that semi-colons have a *different meaning* in actions than in patterns. Semi-colons in patterns have the sense of logical conjunction. In actions, they separate the primitive actions of complex action sequences and *do not* have this logical sense. They cannot be used in actions to make up complex predicates.

Given that there can be nested conditionals, i.e. conditionals of the form

If <pred1> then if <pred2> then <action1> else <action2>,

there must be rules for scoping the nested conditionals. As is commonly done, "else" clauses are scoped by the nearest "if", thus the phrase above is

equivalent to:

If <pred1> then (if <pred2> then <action1> else <action2>))|.

Similarly, complex action sequences are scoped to the nearest conditional. Thus, B.5.a below means the same as B.5.b but not B.5.c.

B.5.a If <p1> then <act1>; if <p2> then <act3>; <act4>.

B.5.b If <p1> then <act1>; (if <p2> then <act3>; <act4>).

B.5.c If <p1> then <act1>; (if <p2> then <act3>); <act4>.

Parentheses, as used in B.5.a-c, are legal here as elsewhere to specify scope. However, their use is discouraged; the philosophy of PIDGIN is to encourage the user to write code that reads like English.

Parser Control

In this section I will present the PIDGIN primitives which can be used in grammar rules to control the parser itself. Two of these primitives activate and deactivate packets, a third is used to indicate that the parse is finished, and a fourth provides a very limited inter-rule control facility.

The two PIDGIN commands to activate and deactivate packets are, as one would expect,

Activate <packet list>

Deactivate <packet list>.

In each of these two commands <packet list> may be a single packet name or a list of packet names separated by commas.

When a grammar rule action concludes that the parse is complete upon hitting some sort of final punctuation character, it signals the grammar

interpreter with the command

The parse is finished.

Executing this command sets a flag which the grammar interpreter checks after each rule action is finished. If the flag is set, the interpreter returns control to whatever higher level process originally called the grammar interpreter. Note that the parse does not terminate immediately upon execution of this command, but rather after the rule it is part of has finished.

A rule can determine its own successor, circumventing the pattern matching process, with the command

Run <rule> next.

This command sets another flag to the name of the rule to be run next. The grammar interpreter checks this flag after the completion of each rule. If the flag is set to a rule name, the grammar interpreter runs that rule instead of pattern matching to find out what rule to run next. Only those parameters of 1st, 2nd, and 3rd which had values within the action of the calling rule have values in the action of the called rule (remembering that each of these parameters normally only has a value within a rule if there is a corresponding description in the pattern of that rule).

Note that this mechanism is essentially a device for abbreviating grammar rules and allowing the grammar writer to capture generalizations. This mechanism allows the grammar writer to abbreviate several rules that end with the same code by making the common code a separate rule and having each of the original rules run it after they are done. Empirically, it

seems that there is usually independent motivation for the common code to be a rule in its own right, so forcing the common code to be a separate rule rather than using some sort of macro expansion mechanism seems to be appropriate. In either case, since this mechanism is purely abbreviatory, it adds no real power to the form of grammar rules.

Comments

PIDGIN allows grammar writers to add comments anywhere within rule actions and patterns. Comments in PIDGIN consist of any text surrounded by "%"s. (To include a "%" in a comment, type two of them.) It is perfectly legal for a single comment to run on for several lines.

Actions

Grammar rule actions are made up of a sequence of instructions, where each instruction is either one of the primitive commands discussed above or an "if..then...else..." conditional expression. Each instruction in an action ends with a period. These instructions are executed sequentially from first to last; each action is thus equivalent to a LISP "PROGN".

How to Throw In the Kitchen Sink

In a natural language front end for a large application- oriented system, it is often convenient to cross the line between syntax and semantics and embed some low level semantic processing in the grammar itself. To allow this to be done where it may be useful, PIDGIN includes some primitives to set and fetch registers, and to allow arbitrary LISP code to be

embedded in PIDGIN rules. It should be stressed that this facility should *not* be used anywhere in the grammar in any way that affects the resulting parse. It should be possible to isolate all PIDGIN code that is "semantic" and remove it from the grammar without affecting the overall behavior of the parser for any input in any way. All uses of the following primitives should be restricted to such "semantic" code.

One can embed pure LISP code inside a grammar rule by simply prefixing any S-expression with "!". To include PIDGIN code inside the LISP code, simply surround the PIDGIN code with "{}". An example of such mixed code is:

```
If 1st is a vp then !(PRINT {the verb of 1st}).
```

PIDGIN also provides a means for calling arbitrary LISP functions without leaving the PIDGIN reader. To call a LISP function, simply use the "FORTRAN" function syntax. Thus, the LISP code

```
(plus 3 4 5)
```

is equivalent to the PIDGIN code

```
plus(3, 4, 5)
```

Registers

While the grammar model does not formally include such a mechanism, it is possible to place and retrieve registers on parse nodes. PIDGIN provides primitives for setting and retrieving the values of registers. The command to set a register is

```
Set the <register> of <node> to <value>.
```

The phrase whose value is the contents of a register is

the <register> register of <node>.

Traces

See Chapter 5 for a description of the following commands:

Set the binding of <node> to <controlling node>.

the binding of <node>

Attention Shifting

See Chapter 8 for a description of the following command:

Restore the buffer.

APPENDIX C

GRAMMAR RULES REFERENCED IN THE TEXT

This appendix is a compendium of all the rules which are invoked in the examples of Chapters 4 and 5. The rules are organized by packets, and the packets are organized approximately in the order that they are activated during the processing of a sentence.

```
{rule INITIAL-RULE in nowhere
[t] -->
Create a new s node.
!(setq s c).
Activate  cpool, ss-start.}
```

SS-START

```
{rule MAJOR-DECL-S in ss-start
[=np] [=verb] -->
Label c s, decl, major.
Deactivate ss-start. Activate parse-subj.}

{rule YES-NO-Q in ss-start
[=auxverb] [=np] -->
Label c s, quest, ynquest, major.
Deactivate ss-start. Activate parse-subj.}

{rule IMPERATIVE in ss-start
[=tnsless] -->
Label c s, imper, major.
Insert the word 'you' into the buffer.
Deactivate ss-start. Activate parse-subj.}
```

PARSE-SUBJ

{rule UNMARKED-ORDER in parse-subj
[=np] [=verb] -->
Attach 1st to c as np.
Deactivate parse-subj.
Activate parse-aux.}

{rule AUX-INVERSION in parse-subj
[=auxverb] [=np] -->
Attach 2nd to c as np.
Deactivate parse-subj. Activate parse-aux.}

PARSE-AUX

{rule START-AUX priority: 10. in parse-aux
[=verb] -->
Create a new aux node.
Label C with the meet of the features of 1st and vspl, v1s,
v+13s, vpl+2s, v-3s, v3s.
%(The above features are "person/number codes", e.g. "vpl+2s"
means that this verb goes with any plural or 2nd person singular
np as subject. The verb "are" has this feature.)%
Label C with the meet of the features of 1st and pres,
past, future, tnsless.
Activate build-aux.}

{rule TO-INFINITIVE priority: 10. in parse-aux
[="to, auxverb] [=tnsless] -->
Label a new aux node inf.
Attach 1st to c as to.
Activate build-aux.}

{rule AUX-ATTACH priority: 10. in parse-aux
[=aux] -->
Attach 1st to c as aux.
Activate parse-vp. Deactivate parse-aux.}

BUILD-AUX

{rule PERFECTIVE priority: 10. in build-aux
[=*have] [=en] --> Attach 1st to c as perf. Label c perf.}

{rule PROGRESSIVE priority: 10. in build-aux
[=*be] [=ing] --> Attach 1st to c as prog. Label c prog.}

{rule PASSIVE-AUX priority: 10. in build-aux
[=*be] [=en] --> Attach 1st to c as passive. Label c passive.}

{rule MODAL priority: 10. in build-aux
[=modal] [=tnsless] --> Attach 1st to c as modal. Label c modal.}

{rule DO-SUPPORT priority: 10. in build-aux
[=*do] [=tnsless] --> Attach 1st to c as do.}

{rule AUX-COMPLETE priority: 15. in build-aux
[t] --> Drop c into the buffer.}

PARSE-VP

{rule MVB in parse-vp
[=verb] -->
Deactivate parse-vp.
If c is major then activate ss-final else
If c is sec then activate emb-s-final.
Attach a new vp node to c as vp.
Attach 1st to c %which is now the vp% as verb.
Activate subj-verb, cpool.}

SUBJ-VERB

```
{rule SUBJ-VERB priority: 15. in subj-verb
[t] -->
%The next line really belongs in the case frame demons.
It appears here for reasons having to do with implementation
peculiarities and will not be discussed in the text.%
If c is not np-preposed
        and there is a np of the s above c then
        it fills the subj slot of the cf of c.
%Activate packets to parse objects and complements.%
If the verb of c is inf-obj then activate inf-comp.
If the verb of c is to-less-inf-obj then activate to-less-inf-comp.
If the verb of c is that-obj then activate that-comp.
If there is a wh-comp of the s above c
        and it is not utilized then activate wh-vp
        else If the s above c is major then activate ss-vp
        else activate embedded-s-vp.
Deactivate subj-verb.}

{rule PASSIVE in subj-verb
[** c; the aux of the s above * is passive;
        the s above * is not np-preposed] -->
Label the s above c np-preposed.
Create a new np node labelled trace.
Set the binding of c to the np of the s above c.
Drop c.}

{rule SEEMS in subj-verb
[=*to] [=tnsless]
[** c; the verb of * is no-subj;
        the s above * is not np-preposed] -->
%This simple form won't handle "John seems happy."%
Run passive next.}
```

SS-VP

```
{rule OBJECTS in ss-vp
[=np] -->
Attach 1st to c as np.}

{rule VP-DONE priority: 20 in ss-vp
[t] --> Drop c.}

{rule PP-UNDER-VP-1 in ss-vp
[=pp] -->
If 1st fits a pp slot of the cf of c
        then attach 1st to c as pp
else run vp-done next.}
```

INF-COMP

```
{rule INF-S-START1 priority: 5. in inf-comp
[=np] [=*to,auxverb] [=tnsless] -->
Label a new s node sec, inf-s.
Attach 1st to c as np.
Activate cpool, parse-aux.}
```

EMBEDDED-S-VP

```
{rule OBJ-IN-EMBEDDED-S in embedded-s-vp
[=np] -->
If 1st fits an obj slot of the cf of c
        then attach 1st to c as np
      else run embedded-vp-done next.}
```

SS-FINAL

```
{rule S-DONE  in ss-final
[=finalpunc] -->
Attach 1st to c as finalpunc.
%The next line is really part of the cf mechanism.%
Finalize the cf of s.
Parse is finished.}
```

APPENDIX D

THE CURRENT GRAMMAR

This appendix documents the current grammar written for PARSIFAL. This grammar has served two purposes; it has served as a testbed for grammar rule development, and it is also the grammar used by the natural language understanding system which serves as front end for the MIT AI Lab Personal Assistant Project. The NP-level rules in this grammar typically were written for the particular needs of the Personal Assistant, and for these rules I claim little generality; the clause-level rules, on the other hand, typically attempt to capture significant linguistic generalizations. Indeed, wherever a generalization has been missed, the grammar will typically so note.

I make no claims for the size of this grammar; less than two months work has gone into the coding of this particular set of rules. It should be noted, however, that much of this grammar derives from an earlier grammar written for an earlier parser of the same sort. This grammar was the product of approximately four months effort. Much of the grammar for this earlier parser has yet to be recoded; it has not been relevant to the PA project.

Special effort has gone into assuring that the clause-level grammar is robust; I have attempted to test the grammar on fairly complex cases involving the interaction of many different rules. The first section of this appendix includes an example set of sentences that the parser can handle; it

should be clear to the reader exactly what phenomena each sentence is attempting to test.

A note about the running time of this parser is in order. For this grammar, running in MACLISP [Moon 74] on the KA-10 version of the PDP-10 computer, these sentences take less then .1 sec/word to parse, including the time taken by the case frame interpreter documented in Appendix D. The reader should note that because the parser operates strictly deterministically, the time taken to parse any given sentence handled by this grammar is approximately a linear function of the number of words in the sentence.

Some Example Sentences

These sentences were all successfully parsed by the grammar included in the next section. The emphasis in this set of examples is to show that a few complex linguistic phenomena are handled robustly by the parser.

i told that boy that boys should do it.
the jar seems broken.
there seems to be a jar broken.
i wanted john to do it.
i want to do it.
i persuaded john to do it.
there seems to have been a meeting scheduled for friday.
schedule a meeting for friday.
is there a meeting scheduled for friday?
does there seem to be a meeting scheduled for friday?
a meeting seems to have been scheduled for Friday.
i told the boy that i saw sue.
i told sue you would schedule the meeting.
i told the girl that you would schedule the meeting.
the boy who wanted to meet you scheduled the meeting.
the boy who met you scheduled the meeting.
the boy who you met scheduled the meeting.
the boy you met scheduled the meeting.
who did john see?
who broke the jar?
what did bob give to sue?
who did bob give the book?
who did bob give the book to?
what did bob give to sue?
what did bob give sue?
i promised john to do it.
who did you say that bill told?
you promised to give the book to john.
who did you promise to give the book to?
who did you promise to schedule the meeting?
who did you say scheduled the meeting?
who did you persuade to do it?
what did you give sue yesterday?
who did you give the book yesterday?
who did you ask to schedule the meeting?
who do you want to give a book to tomorrow?
who did you want to give a book to sue?
i gave the boy who you wanted to give the books to three books.
who did you promise to give the book to tomorrow?
who did you promise to give the book to sue tomorrow?

The Grammar as of 7/77

```
;;/THE PARSER, BY CONVENTION, INITIALLY CALLS THE RULE INITAL-RULE.%

!RULE INITIAL-RULE IN NOWHERE
[t] -->
Create a new s node.
'(setq s c).
Activate cpool, ss-start.!

;;/A RULE TO GRAB SINGLE NPS AS UTTERANCES%

!RULE NP-UTTERANCE IN SS-START
[=np] [=finalpunc] -->
Label c np-utterance.
Attach 1st to c as np.
Attach 2nd to c as finalpunc.
The parse is finished.!

!RULE PP-UTTERANCE IN SS-START
[=pp] [=finalpunc] -->
Label c pp-utterance.
Attach 1st to c as pp.
Attach 2nd to c as finalpunc.
The parse is finished.!

;;/RULES TO INITIATE MAJOR CLAUSES.%
(Comment Initiate major clauses)
!RULE MAJOR-DECL-S IN SS-START
[=np] [=verb] -->
Label c decl, major.
Deactivate ss-start. Activate parse-subj.!

!RULE YES-NO-Q IN SS-START
[=auxverb] [=np] -->
Label c quest, ynquest, major.
Deactivate ss-start. Activate parse-subj.!

!RULE IMPERATIVE IN SS-START
[=tnsless] -->
Label c imper, major.
Insert the word 'you' into the buffer.
Deactivate ss-start. Activate parse-subj.!

!RULE HAVE-DIAG PRIORITY: 5 IN SS-START
[=*have, tnsless] [=np] [t] -->
/If can't prove it's not a yes-no-q then assume it is.%
If 2nd is ns, n3p or 3rd is not verb or 3rd is tnsless
          then run imperative next else
          run yes-no-q next.!

!RULE WH-QUEST PRIORITY: 5 IN SS-START
[=wh] [=verb] -->
Label c major, quest, wh-quest.
Attach 1st to c as whcomp.
If 1st is a pp then label c pp-quest else
If 1st is a np then label c np-quest.
Deactivate ss-start. Activate parse-subj, wh-pool.!
```

```
(comment Subject parsing)
!RULE UNMARKED-ORDER IN PARSE-SUBJ
[=np] [=verb] -->
Attach 1st to c as np.
Deactivate parse-subj. Activate parse-aux.!

!RULE AUX-INVERSION IN PARSE-SUBJ
[=auxverb] [=np] -->
Attach 2nd to c as np.
Deactivate parse-subj. Activate parse-aux.!

!RULE SUBJ-QUEST? PRIORITY: 5. IN PARSE-SUBJ
[=verb]  [== c; = is np-quest] [=np] [t] -->
If 1st is not auxverb or 3rd is not verb
    then create a new np node labelled trace, not-modifiable;
           set the binding of c to wh-comp;
           drop c;
           label wh-comp utilized
    else run aux-inversion next.!
```

```
(COMMENT RULES FOR BUILDING AUXILIARIES)
;;/RULES FOR BUILDING AUXILIARIES/
;;/predicate adjectives and prepositions should be parsed as verbs%

!RULE STARTAUX IN PARSE-AUX
[=verb] -->
Create a new aux node.
Transfer vsp1, v1s, v+13s, vp1+2s, v-3s, v3s from 1st to c.
%(The above features are "person/number codes", e.g. "vp1+2s"
means that this verb goes with any plural or 2nd person singular
np as subject.  The verb "are" has this feature.)%
Transfer pres, past, future, tnsless from 1st to c.
Activate build-aux, cpool.!

!RULE TO-INFINITIVE IN PARSE-AUX
[=+to, auxverb] [=tnsless] -->
Label a new aux node inf.
Attach 1st to c as to.
Activate build-aux, cpool.!

!RULE AUX-ATTACH IN PARSE-AUX
[=aux] -->
Attach 1st to c as aux.
Activate parse-vp. Deactivate parse-aux.!

!RULE PERFECTIVE IN BUILD-AUX
[=+have] [=en] --> Attach 1st to c as perf.  Label c perf.!

!RULE PROGRESSIVE IN BUILD-AUX
[=+be] [=ing] --> Attach 1st to c as prog.  Label c prog.!

!RULE PASSIVE-AUX IN BUILD-AUX
[=+be] [=en] --> Attach 1st to c as passive.  Label 2nd passive.!

!RULE MODAL IN BUILD-AUX
[=modal] [=tnsless] --> Attach 1st to c as modal.  Label c modal.!

!RULE FUTURE IN BUILD-AUX
[=+will] [=tnsless] --> Attach 1st to c as will. Label c future.!

!RULE DO-SUPPORT IN BUILD-AUX
[=+do] [=tnsless] --> Attach 1st to c as do.!

!RULE BE-PRED IN BUILD-AUX
[=+be] [= is any of prep, adj; * is not part] -->
Attach 1st to c as copula.
Label c copula.
Label 2nd verb, pred-verb.!

!RULE AUX-COMPLETE PRIORITY: 15. IN BUILD-AUX
[t] --> Drop c into the buffer.!

!RULE THERE IN BUILD-AUX
[=+be] [=np]
[== c; the noun of the nbar of the binding of the np of the current s is ethere] -->
Label the current s existential.
Attach 2nd to the current s as np.!
```

```
(Comment Verb processing)

(RULE main-verb IN PARSE-VP
[=verb] -->

%Set up state of S node%
Deactivate parse-vp.
If c is major then activate ss-final else
If c is sec then activate emb-s-final.

%Attach VP and V.%
Attach a new vp node to c as vp.
Attach 1st to c /which is now the vp% as verb.
Activate cpool.

%Activate packets to parse objects and complements.%
If there is a verb of c and it is passive
    then activate passive; run passive next.
%The following is close to an explicit system of verb types.%
If it is inf-obj then
    if it is to-less-inf-obj then activate to-less-inf-comp andthen
    if it is to-be-less-inf-obj then activate to-be-less-inf-comp andthen
    if it is 2-obj-inf-obj then activate 2-obj-inf-comp
          else activate inf-comp;
          if it is subj-less-inf-obj then activate subj-less-inf-comp else
          if it is no-subj then activate no-subj.
If it is that-obj then activate that-comp.

%Activate correct packet for regular objects.%
If there is a wh-comp and it is not utilized
    then activate wh-vp else
If the current s is major then activate ss-vp else
Activate embedded-s-vp.)

(ATTACHMENT CRULE VP-VERB  VP OVER VERB
Associate a new case frame with the upper node.
Associate the case frame of the upper node with the s above upper.
The aux of the s above upper fills the spec slot of  upper.
The lower node fills the pred slot of  upper.
If the lower node is none of no-subj, passive
     and there is a binding of the np of the s above upper
     then it fills the subj slot of upper.)

(RULE PREOP IN parse-verb
[= is any of pp /more to come%] -->
Deactivate parse-verb.
If c is major then activate ss-final else
If c is sec then activate emb-s-final.
Label 1st predp.
Attach 1st to c as predp.)

(RULE PASSIVE PRIORITY: 5 IN PASSIVE
%Activated by MVB if verb is labelled passive.%
t -->
Label the current s np-preposed.
Create a new np node labelled trace, not-modifiable.
Set the binding of c to the np of the current s.
Drop c.
Deactivate passive.)

(RULE SEEMS IN NO-SUBJ
[seeto] [=tnsless] -->
Deactivate no-subj.
Run passive next.)
```

```
(Comment Parse simple objects)
;;/Parsing the objects of simple sentences%

!RULE OBJECTS IN SS-VP
[=np] -->
Attach 1st to c as np.!

!RULE VP-DONE PRIORITY: 20 IN SS-VP
[t] --> Drop c.!

!RULE S-DONE  IN SS-FINAL
[=finalpunc] -->
Attach 1st to c as finalpunc.
Finalize the cf of c.
Parse is finished.!

;;/Parsing the objects of embedded clauses.%

!RULE OBJ-IN-EMBEDDED-S IN EMBEDDED-S-VP
[=np] -->
If 1st fits an obj slot of the cf of c
           then attach 1st to c as np
        else run embedded-vp-done next.!

!ATTACHMENT CRULE VP-NP VP OVER NP
The lower node fills an obj slot of the upper.
/With no-subj verbs, can't know that the subject really is the subject until
we see the complement./
If the verb of the upper node is no-subj
      and the s above the upper node is not np-preposed
      then the np of the s above the upper node
           fills the subj slot of upper.
/We can bind delta subjects and finalize the case frames of infinitive
clauses only after they are attached to the clause, allowing all the
wh-placement stuff to have done whatever it will do.
See the rules on "Infinites with delta subjects for more on this.%
If there is an s of lower and the np of it is delta then
      if the verb of the upper node is obj-binds-delta then
           set the binding of the np of the s of lower
                to the indirect object of upper
           else
     if the verb of the upper node is subj-binds-delta
          or
          the verb of the upper node is subj-less-inf-obj
          and it isn't true that
               there is a binding of the np of the s of the lower node
          then set the binding of the np of the s of lower
               to the np of the current s.
If there is an s of lower and the np of it is delta then
      the np of the s of lower fills the subj slot of the s of lower;
      finalize the cf of the s of lower.!

!RULE EMBEDDED-VP-DONE PRIORITY: 15. IN EMBEDDED-S-VP
[t] -->
Drop c.
Activate embedded-s-final.!

!RULE EMBEDDED-S-DONE PRIORITY: 20 IN EMBEDDED-S-FINAL
[t] -->
Finalize the cf of c.
Drop c.!
```

```
;;Rules to start relative clauses
(Comment Relative clauses)

!RULE WH-RELATIVE-CLAUSE IN NP-COMPLETE
[=relpron-np] [t] -->
Label c modified.
Attach a new s node labelled sec, relative to c as s.
Activate cpool, parse-subj, wh-pool.
/Note that the node is attached as "whcomp"; "wh-comp" refers to
the special wh register./
Attach 1st to c as whcomp.
Set the binding of 1st to the np above c.
If 2nd is a verb then
          create a new np node labelled trace, not-modifiable;
          Set the binding of c to wh-comp;
          Label the binding of c utilized;
          Drop c into the buffer.!

!RULE REDUCED-RELATIVE IN NP-COMPLETE
[=np; e is not relpron-np][=verb] -->
Insert the word 'wh-' into the buffer before 1st.!

!RULE WHICH-DIAGN IN CPOOL
[=*which; * is not any of  quant, relpron] -->
If the np above c is not modified then
          label 1st pronoun, relpron, wh
          else label 1st quant, ngstart, ns, npl, wh.!

!RULE WHAT-DIAG priority: 15 IN npool
[=*what] [t] -->
%This won't work for cleft sentences: what little fish eat is little worms
(but are these garden paths?)%
If 2nd is ngstart and 2nd is not det
     then label 1st det, ns, npl, n3p, wh;
     activate parse-det
     else label 1st pronoun, relpron, wh.!
```

```
(comment The wh-comp mechanism of the grammar interpreter)
;This impliments a mechanism that is different in efficiency, but otherwise
;exactly equivalent to the mechanism documented in chapter 7. I use a
;register to cache the wh-comp. It can be considered an implimentation of
;that mechanism.

[ATTACHMENT CRULE S-WHCOMP S OVER WHCOMP
/This crule and the next should really be a part of the grammar
interpreter, hardwired and universal.  I put them here so that
they are clearly visible.  The effect is exactly equivalent.
This rule primes the wh-comp mechanism./
Set the :wh-comp of the upper node to the lower node.]

[CREATION CRULE S-CREATE S
/This is ugly and adhoc.  It simply embodies a form of
the complex NP constraint, which is adhoc and ugly.
That this rule should be a putative universal doesn't help much.
The rule: scan up the active node stack until you come to either an NP
or an S.  If an NP, that's it.  If an S, copy its wh-comp.  This IS
really a piece of the interpreter, and must be written in LISP./

Set the :wh-comp of c to find-wh-comp (the node above c)]


(defun find-wh-comp (node-variable)
;Now into PIDGIN
[if there is not a node-variable or
     the node-variable is an np then nil else
if the node-variable is an S
     then the :wh-comp register of the node-variable else find-wh-comp
(the node above node-variable)])
```

```
(comment WH placement)

;These rules differ somewhat from those discussed in chapter 10, mainly
;because the semantic mechanism that is actually implemented is weaker
;than that assumed in chapter 10.  For an explanation of how the "prefer"
;test really works, see appendix E.

!RULE WH-WITH-END-NEXT PRIORITY: 15. IN wh-vp
 t -->
If the greatest possible number of objects of c is equal to 0
          or
     it isn't true that the wh-comp fits an obj slot of the cf of the current s
     and the verb of c is not comp-obj
          then run too-many-nps next
          else run create-wh-trace next.!

!RULE WH-WITH-NP-NEXT IN WH-VP
[=np] -->
/This rule decides whether or not to use the wh-comp if there is an NP
next.  Note that if the NP is followed by a PP, then a different rule
will run.  Running an object rule next indicates that the wh-comp is not
to be used, while running create-wh-trace indicates the wh-comp is to be
used./

If the greatest possible number of objects of c is less than 2
     /i.e. the current verb can only take 1 object/
     then if the current s is major then run objects next
               else run obj-in-embedded-s next
     else
/There are 2 object slots available; we'll use both./
The number of objects of c will be 2;
If semantics prefers 1st filling an obj slot of c somewhat
     better than wh-comp filling an obj slot of c then
          run objects next else
If semantics prefers the wh-comp filling an obj slot of c much
     better than 1st filling an obj slot of c then
     label the current s slightly-bad;
     run create-wh-trace next else
If semantics prefers 1st filling an obj slot of c no
     better than the wh-comp filling an obj slot of c then
     label the current s slightly-bad;
     run objects next else
     label the current s very-bad;
     run create-wh-trace next.!

!RULE WH-WITH-NP-PP-NEXT PRIORITY: 7 IN WH-VP
[=np] [=prep] -->
If the greatest possible number of objects of c is greater than 1
     and a prepositional phrase of 2nd and the wh-comp fits a pp slot of c
     or
     the greatest possible number of objects of c is equal to 1
     and a prepositional phrase of 2nd and the wh-comp fits a pp slot of the current s
     then run objects next else
If the greatest possible number of objects of c is greater than 1
     then run wh-with-np-next next else
Run too-many-nps next.!
```

```
(RULE WH-WITH-PP-NEXT PRIORITY: 5 IN WH-VP
[=prep] [=np] -->
If a prepositional phrase of 1st and 2nd fits a pp slot of c
%The syntactic preference is for simple pps.%
      then run pp next else
If it isn't true that
      a prepositional phrase of 1st and the wh-comp fits a pp slot of c
      /i.e it's true that the wh-comp can't serve as the object of the prep%
      then if greatest possible number of objects of c is greater than 0
            /i.e. if the wh-comp can serve as an object then%
            then run create-wh-trace next
            else /Hopefully the trace is a time word%
                  run too-many-nps next
      else
If the lowest possible number of objects of c is greater than 0
      /i.e. this verb MUST take an object%
      then run create-wh-trace next else
      run wh-pp-build next!


(RULE WH-PP-BUILD IN CPOOL
[=prep] [= is not np]
[== c; there is a wh-comp and it is not utilized] -->
Attach 1st to a new pp node as prep.
Attach a new np node labelled trace to c as np.
Set the binding of c to the wh-comp.
Label the wh-comp utilized.
Drop c /i.e. the trace/.
Drop c /i.e. the pp/.!

(RULE WH-WITH-PP-2 IN CPOOL
[=prep] [=np]
[== c; there is a wh-comp and it is not utilized] -->
If c is any of nbar, np and
       a prepositional phrase of 1st and 2nd fits a pp slot of c
       then run pp next /building the simple pp%
       else
If a prepositional phrase of 1st and the wh-comp fits a pp slot of c
      then run wh-pp-build next /building a pp with the wh-comp% else
%the current constituent is done%
If c is an nbar then run nbar-done next else
If c is an np then run np-done next else
run pp next.!


(RULE WH-RESOLVED PRIORITY: 5 IN wh-vp
[== c; the wh-comp is utilized ] -->
Deactivate wh-vp.
If the current s is major
            then activate ss-vp
            else activate embedded-s-vp.!

(RULE TOO-MANY-NPS PRIORITY: 15 IN WP-VP
[=np]
[== c; the greatest possible number of objects of c is less than 2] -->
%This is just a start, clearly%
if there is not a whcomp of the current s
      /i.e the wh-comp originated from higher up%
      then run wh-resolved next
      else !(warn 1. too-many-nps loses).!

(RULE CREATE-WH-TRACE PRIORITY: 14 IN WH-POOL
T -->
Create a new np node labelled trace, not-modifiable.
Set the binding of c to the wh-comp.
Label the wh-comp utilized.
```

```
Drop c into the buffer.!

(RULE WH-RESOLVED-1 PRIORITY: 5 IN WH-POOL
(## c; the wh-comp is utilized) -->
Deactivate wh-pool.!
```

```
;;Rules for that clauses
(comment That clauses)

(RULE THAT-S-START PRIORITY: 5 IN CPOOL
[=comp, =that] [=np] [=verb] -->
/Handles the marked case;always active.%
Label a new s node sec, comp-s, that-s.
Attach 1st to c as comp.
Attach 2nd to c as np.
Activate cpool, parse-aux.)

(RULE THAT-S-START-1 PRIORITY: 5. IN THAT-COMP
[=np] [=verb] -->
Label a new s node sec, comp-s, that-s.
Attach 1st to c as np.
Activate cpool, parse-aux.)

(RULE COMP-TO-NP IN CPOOL
[=COMP-S] -->
Attach 1st to a new NP node labelled comp-np, not-modifiable as s.
Drop c into the buffer.)

(RULE that-diag-1 in cpool
[=+that; + is none of comp, det, pronoun] [=np] -->
/This rule diagnoses whether "that" is a comp or a determiner or a
relative pronoun.  This rule will misdiagnose
"That deer ate everything in my garden surprised me."
but then again so did you.
The diagnostic is: if "that" must be a det, it is.
If c is nbar, i.e. we're under an np, assume "that" starts a relative clause.%
If there is not a det of 2nd
           and there is not a qp of 2nd
           and the nbar of 2nd is none of npl, massn
           and 2nd is not not-modifiable
 then   attach 1st to 2nd as det;
           label 1st det, ns
 else   if c is a nbar then label 1st pronoun, relpron
 else   label 1st comp.)

(RULE THAT-DIAG-2 PRIORITY: 13 IN CPOOL
/If there's nothing else to do with that, it's a pronoun.%
[=+that; + is not pronoun] -->
Label 1st pronoun.)


(RULE THAT-DIAG-3 PRIORITY: 5 IN NBAR-COMPLETE
[=+that; + is none of pronoun, comp] [=np]
[=+ c; the verb of the vp of the current s is that-obj;
the lowest possible number of objects of the current s is equal to 2]
-->
Label 1st comp.)
```

```
;;%Rules to do infinitive clauses.%
(Comment Infinitive clauses)

!RULE TO-DIAG PRIORITY: 5. IN INF-COMP, 2-OBJ-INF-COMP
[=*to, auxverb, prep] [=tnsless]
[=* c; there is a wh-comp and it is not utilized] -->
%Does this really belong in this packet or in cpool?
Should the fact that an inf-comp is expected serve as evidence?%
if a pp of 1st and wh-comp
          fits a pp slot of the cf of the current s
then remove the feature auxverb from 1st
else remove the feature prep from 1st.!

!RULE INF-S-START PRIORITY: 5. IN CPOOL
[=*for] [=np] [=*to] -->
%Handles the marked case; always active%
Label a new s node sec, comp-s, inf-s.
Attach 1st to c as comp.
Attach 2nd to c as np.
Activate cpool, parse-aux.!

!RULE INF-S-START1 PRIORITY: 5. IN INF-COMP
%The COMP can only be dropped if the complement is expected.%
[=np] [=*to,auxverb] [=tnsless] -->
Label a new s node sec, comp-s, inf-s.
Attach 1st to c as np.
Activate cpool, parse-aux.!

!RULE INSERT-TO IN TO-LESS-INF-COMP
%This rule and INSERT-TO-BE must be in different packets
- consider "help" vs. "seems"%
[=np] [=tnsless] -->
Insert the word 'to' into the buffer before 2nd.!

!RULE INSERT-TO-1 IN TO-LESS-INF-COMP
%I've clearly missed a generalization somewhere.%
[=tnsless] -->
Insert the word 'to' into the buffer before 1st.!

!RULE INSERT-TO-BE IN TO-BE-LESS-INF-COMP
[=np] [= is any of en, adj] -->
%won't work for adjectives until interpreter changed%
Insert the word 'be' into the buffer before 2nd.
Insert the word 'to' into the buffer before 2nd.!

!RULE INSERT-TO-BE-1 IN TO-BE-LESS-INF-COMP
[= is any of en, adj] -->
%Again, I've missed the generalization%
Insert the word 'be' into the buffer before 1st.
Insert the word 'to' into the buffer before 1st.!
```

```
(comment Infinitives with delta subjects)

!RULE CREATE-DELTA-SUBJ IN 2-OBJ-INF-COMP
[=*to, auxverb] [=tnsless] -->
%This rule handles verbs like promise and persuade.  The analysis
I'm adopting assumes that the subject of the complement is a delta
(i.e. a "base-generated" trace), which is not bound by syntactic rules,
but by later semantic processing.  The binding is done in the crule
vp-np%
Create an np node labelled trace, not-modifiable, delta.
Drop c into the buffer.
Deactivate 2-obj-inf-comp. Activate inf-comp.!

!RULE create-delta-subj-1 IN SUBJ-LESS-INF-COMP
[=*to, auxverb] [=tnsless] -->
%Handles verbs like "want" that may have an explicit subject
or may have a delta subject.  These cases are hard
if they interact with wh-movement.%
Create an np node labelled trace, not-modifiable.
Drop c into the buffer.!

!RULE SUBJECT-IS-DELTA-DIAG PRIORITY: 15. IN EMBEDDED-S-FINAL
[=* c; the np of c is trace; the np of c is not delta;
        there is not a binding of the np of c] -->
%This rule finishes the analysis of embedded clauses of verbs like
"want", deciding whether the subject is a bound trace or a delta.%
If there is a wh-comp and it is not utilized
      then set the binding of the np of c to wh-comp;
      %This next belongs in the case mechanism, but its simpler here.
      Life would be simpler if "want" took 2 objects.%
      the np of c fills the subj slot of the cf of c;
      label the wh-comp utilized
      else label the np of c delta.!

!RULE DELTA-SUBJ-S-DONE PRIORITY: 15. IN EMBEDDED-S-FINAL
[=* C; THE NP OF C IS DELTA] -->
%This rule overrides embedded-s-done; and does not finalize the cf.%
Drop c into the buffer.!

!ATTACHMENT CRULE NP-S  NP OVER S
If lower is inf-s then set the markers register of upper to !'(inf-comp).
If lower is that-s then set the markers of upper to !'(that-comp).
If lower is relative
           then lower fills a mod slot of the upper node
           else associate the case frame of lower with the upper node.!
```

```
;;/NP CREATING RULES/
(Comment Mainline np parsing)

!AS RULE STARTNP IN CPOOL
[=ngstart] -->
Create a new np node.
If 1st is det then activate parse-det
          else activate parse-qp-1.
Activate npool.!

!CREATION CRULE NP-START NP
if c is none of propn-np, pron-np, name, complex-noun-np, trace, *,
             comp
             then associate a new case frame with c.!

;;this should be a finalize rule.
;;!ATTACHMENT CRULE NP-QP  NP OVER QP
;;The lower node fills the spec slot of upper.!

!RULE DETERMINER IN PARSE-DET
[=det] -->
Attach 1st to c as det.
Label c det.
Transfer the features indef, def, wh from 1st to c.
Deactivate parse-det. Activate parse-qp-2.!

!RULE A-HUNDRED-DIAG PRIORITY: 5. IN PARSE-DET
[=a]
[=num, bignumg; the num1 of the num1 of * is any of shundred, bignum] [t] -->
If 3rd is noun, ns, measure
     then run determiner next
     else attach 1st to the num1 of 2nd as num8;
          deactivate parse-det;
          activate parse-qp-1.!

!RULE ALL-THE PRIORITY: 5 IN PARSE-QP-1
[=*all] [=det,def] -->
/Another similar rule needed to cover "half the boys", "half A bottle"
Also there may be a timing problem with "all G-d's children".X
Insert the word 'of' into the buffer before 2nd.!

!RULE QUANT IN PARSE-QP-1
[=quant] -->
Attach a new qp node to c as qp.
Attach 1st to c as quant.
If 1st is num then label c numqp.
Transfer ns, npl from 1st to c.
Transfer the feature wh from 1st to the np above c.
Drop c.
Run quant-done next.!

!RULE QUANT-DONE PRIORITY: 15 IN PARSE-QP-1
[t] -->
Deactivate parse-qp-1. Activate parse-adj.!

!RULE ORDINAL IN PARSE-QP-2
[=ord, complete-num] [t] -->
Attach a new qp node labelled ordqp to c as qp.
Attach 1st to c as ord.
If 2nd is num then attach 2nd to c as num;
          label c numqp;
            transfer ns, npl from 1st to c
       else label c ns, npl.
Drop c.
Deactivate parse-qp-2. activate parse-adj.!
```

```
!RULE DET-QUANT IN PARSE-QP-2
[=quant; ≠ is any of detq, num] -->
/There should be a less ad-hoc way than the feature "detq" to distinguish
quantifiers that can follow determiners (like "many") from those which don't
(like "some).../
Attach a new qp node to c as qp.
Attach 1st to c as quant.
If 1st is num then label c numqp.
Transfer ns, npl from 1st to c.
Drop c.
Run det-quant-done next.!

!RULE DET-QUANT-DONE PRIORITY: 15 IN PARSE-QP-2
[t] -->
Deactivate parse-qp-2. Activate parse-adj.!

;;will need a rule that does "next <time> " when np stuff is fixed.
```

```
(RULE ADJ IN PARSE-ADJ
[t] -->
If lst is adj then attach lst to c as adj
        else if lst is =/, then attach lst to c as comma
           else deactivate parse-adj; activate parse-noun.)

(ATTACHMENT CRULE NP-ADJ NP OVER ADJ
/Note that this does not handle non-attributive adjectives like "main"%
Associate a new mod case frame with the lower node.
The lower node fills a mod slot of the upper node.
The lower node fills the pred slot of lower.
Create a new = node labelled np.
Set the binding of c to the upper node.
C fills the subj slot of lower.
Finalize the cf of lower.
Drop c.)

(RULE NOUN IN PARSE-NOUN
[=noun] -->
Attach lst to a new nbar node as noun.
Transfer the features massn, time, ns, npl,
          n1p, n2p, n3p from lst to c.
Drop c.)

(RULE NBAR IN PARSE-NOUN
[=nbar] -->
Attach lst to c as nbar.
If the noun of lst is any of propnoun, pseudopropnoun then label c not-modifiable.
Transfer the features time, place, n1p, n2p, n3p from lst to c.
Deactivate npool, parse-noun.
/The next line has to do with the Node reactivation mechanism, about which
see below./
Drop c into the buffer.
Restore the buffer.)

(ATTACHMENT CRULE NP-NBAR NP OVER NBAR
Associate the case frame of the upper node with the lower node.
If there is a case frame of upper and
  there is a noun of the lower node
          then it fills the pred slot of upper.)
```

```
(comment Incomplete nps)
!RULE INCOMPLETE-NP PRIORITY: 15. IN PARSE-NOUN
[t] -->
Attach a new nbar node to c as nbar.
Attach a new noun node labelled trace, not-modifiable, delta to c as noun.
Drop c.
Drop c.
Deactivate npool, parse-noun.
If there is a qp of c
        then label c quant-np; activate quant-np-complete, cpool
          else drop c;
          restore the buffer.!

!RULE ALL-OF-THE-BOYS IN QUANT-NP-COMPLETE
[=pp; the prep of = is =of] -->
Attach 1st to c as pp.
Set the binding of the noun of the nbar of c to
         the noun of the nbar of the np of 1st.
Run quant-np-done next.!

!RULE QUANT-NP-DONE PRIORITY: 15. IN QUANT-NP-COMPLETE
t -->
Deactivate quant-np-complete.
Activate np-complete.!


(comment NBAR-reactivation)

!NR RULE NBAR-COMPLETE IN CPOOL
/This is an experimental type of rule not discussed in the thesis.
The experiment is still in progress, and I suspect this type of rule is not
motivated.  This type of rule, called a Node-Reactivation rule, is similar
to an Attention Shifting rule.  The mechanism is briefly discussed in <ref.
my ACM paper.>/
[=nbar] -->
Activate cpool, nbar-complete.!

!RULE NBAR-DONE PRIORITY: 15. IN NBAR-COMPLETE
t -->
Drop c.
Restore the buffer.!

(comment Np-reactivation)
!NR RULE NP-COMPLETE IN CPOOL
[=np; = is none of modified, not-modifiable] -->
/NUMBER AGREEMENT/
If there is not a det of c and there is not a qp of c then
           if the nbar of c is npl then label c npl else
           if the noun of the nbar of c is  propnoun then label c prop-np else
           if the nbar of c is massn, ns then label c massnp else
           if the nbar of c is time and the ord of the qp of c is general-ord
                    then label c next-time-np else
        if the noun of the nbar of c is none of pseudopropnoun then
           label c bad.
If there is a det of c or there is a qp of c then
        transfer the meet of
                                (if there is a det of c then the features of it else ns,npl)
                      and (if there is a qp of c then the features of it else ns,npl)
           from the nbar of c to c;
           if c is none of ns,npl then label c bad.
Activate cpool, np-complete.!

!RULE NP-DONE PRIORITY: 15. IN NP-COMPLETE
t -->
If there is a case frame of c then finalize the cf of c.
Drop c.
```

```
Restore the buffer.|
```

```
(Comment Numbers)
;;/Rules to parse numbers/

!AS RULE NUMBER IN NPOOL
[=num ; ≠ is not complete-num] -->
Deactivate npool.
Activate build-number.!

!RULE NUMBER-DONE PRIORITY: 12. IN BUILD-NUMBER
[t] -->
Label 1st complete-num.
If 1st is not ord then label 1st quant.
If 1st is none of ns,np1 then label 1st np1.
Deactivate build-number.
Activate npool.
Restore the buffer.!

!RULE ORDINAL-DONE PRIORITY: 5. IN BUILD-NUMBER
[=ord] [≠≠ c; ≠ is not num] -->
Run number-done next.!


!RULE NINETY-NINE IN BUILD-NUMBER
[=tens] [=ones] -->
Label a new num node 99s.
Attach 1st to c as num1.
Attach 2nd to c as num2.
Set the quant of c to
          plus(the quant register of 1st, the quant register of 2nd).
Transfer ord from 2nd to c.
Drop c.!

!RULE TWO-HUNDRED IN BUILD-NUMBER
[ ≠ is any of ones, ≠ten, ≠10, 99s; ≠ is not ord] [=shundred] -->
Label a new num node hundred+.
Attach 1st to c as num1.
Attach 2nd to c as num2.
Set the quant of c to
          times(the quant register of 1st, the quant register of 2nd).
Transfer ord from 2nd to c.
Drop c.!

!RULE HUNDRED IN BUILD-NUMBER
[=shundred] -->
Label a new num node hundred+.
Attach 1st to c as num1.
Set the quant register of c to the quant register of 1st.
Transfer ord from 1st to c.
Drop c.!


!RULE HUNDREDS-STARTS-BIGNUMG IN BUILD-NUMBER
[ ≠ is any of hundred+, bignum+; ≠ is not ord] -->
Create a new num node labelled bignumg.
Attach 1st to c as num1.
Activate build-number.!

!RULE 99S-ATTACH IN BUILD-NUMBER
[t] [≠≠ c; ≠ bignumg] -->
If there is a conj of c or 1st is any of 99s, tens, ones then
          Attach 1st to c as num2;
          Transfer ord from 1st to c;
          Set the quant of c to
                    plus (the quant register of num1 of c,
```

```
                                the quant register of 1st)
            else set the quant register of c to the quant register of numl of c.
Drop c.!

(RULE HUNDRED-AND IN BUILD-NUMBER
[=eand] [== c; =bignumg] -->
Attach 1st to c as conj.!
```

```
!RULE BIGNUM IN BUILD-NUMBER
[=num; ≠ is not bignumg, ord] [=bignum; ≠ is not *hundred] -->
Create a new num node labelled bignum*.
Attach 1st to c as num1.
Attach 2nd to c as num2.
Set the quant  of c to
          times(the quant register of 1st, the quant register of 2nd).
Transfer ord from  1st to c.
Drop c.!

!RULE TEN-TWENTY IN BUILD-NUMBER
[≠ is any of ones, 99s, *ten]
[≠ is any of 99s, tens; ≠ is not ord] -->
Create a new num node labelled listnum.
Attach 1st to c as num1.
Activate build-number.!

!RULE TEN-TWENTY-FINISH IN BUILD-NUMBER
[=* c; ≠ listnum]
[≠ is any of 99s, tens; ≠ is not ord] -->
Attach 1st to c as num2.
Set the quant register of c to
          plus (times (100., the quant register of num1 of c),
                  the quant register of 1st).
Drop c.!
```

```
;;/Special time np constructions/
(Comment Time)

!RULE TWO-OCLOCK PRIORITY: 5 IN NPOOL
[=num] [ c is any of =o/'clock, =a/.m/., =p/.m/. ] -->
Create a new noun node labelled time, hour, complex-noun, propnoun.
Attach a new np node labelled time, complex-noun-np to c as np.
Attach 1st to c /which is now the np/ as hour.
Activate build-hour.!

!RULE TWO-THIRTY PRIORITY: 5 IN NPOOL
[=num] [=:] [=num] -->
Create a new noun node labelled time, hour, complex-noun, propnoun.
Attach a new np node labelled time, complex-noun-np to c as np.
Attach 1st to c /which is now the np/ as hour.
Attach 2nd to c as colon.
Attach 3rd to c as minute.
Activate build-hour.!

!RULE HOUR-COMPLETE IN BUILD-HOUR
[t] -->
If 1st is any of =o/'clock, =a/.m/., =p/.m/.
        then attach 1st to c as oclock.
Drop c /i.e. the complex-noun-np / .
/For the sake of semantics, all the following:/
Set the time register of c to 'hour'.
Label c hour.
Set the hours register of c
          to the quant register of the hour of the np of c.
Set the minutes register of c to
        (if there is a minute register of the np of c
             then quant register of it else 0).
If there is an oclock register of the np of c
        then set the time-of-day register of c to it
        else set the time-of-day register of c to the word 'o/'clock'.
Set the markers register of c to !'(time).
Finalize the cf of c.
Drop c.!

!RULE MONTH PRIORITY: 5. IN NPOOL
[=month; c is not complex-np] -->
Create a new noun node labelled time, complex-np, propnoun.
Attach a new np node labelled time, complex-noun-np to c as np.
Attach 1st to c as noun.
Activate npool, month-build.!

!RULE JUNE-1ST IN MONTH-BUILD
[=complete-num] -->
/Again, using fact that ORDs have NUM feature./
Attach a new qp node to c as qp.
Attach 1st to c /i.e. the qp/ as ord.
If 1st is not ord then label c ord.
Drop c.
Run month-complete next.!


!RULE JUNE-THE-1ST IN MONTH-BUILD
[=the] [=ord] -->
Attach a new qp node to c as qp.
Attach 1st to c as det.
Attach 2nd to c as ord.
Drop c.
Run month-complete next.!

!RULE THE-FIRST-OF-JUNE PRIORITY: 5 IN PARSE-QP-2
[=ord] [=of] [=month] -->
```

```
Create a new noun node labelled time, complex-np, propnoun, ns.
Attach a new np node labelled time, complex-noun-np to c as np.
Attach a new qp node to c as qp;
          Attach 1st to c as ord;
          Drop c.
Attach 2nd to c as of.
Attach 3rd to c as noun.
Run month-complete next.!
```

```
!RULE MONTH-COMPLETE PRIORITY: 15. IN MONTH-BUILD
[t] -->
Drop c /. i.e. the time np , so c is now the noun/ .
Set the month register of c to the noun of the np of c.
Set the markers register of c to !'(time).
If there is not a qp of the np of c
          then set the time register of c to 'month'; label c month
          else set the time register of c to 'date'; label c date;
          set the day register of c to
                    the quant register of the ord of the qp of the np of c.
Finalize the cf of c.
Drop c.!

!RULE NUMBER-TO-YEAR IN QUANT-NP-COMPLETE
[== c; = is quant-np; = is not time; the quant of the qp of = is listnum]
 -->
Label c time,year.
Set the markers register of c to !'(time).
Set the time register of c to 'year'.
Set the year register of c to the quant register of the quant of the qp of c.!

!AS RULE MONDAY PRIORITY: 5 IN CPOOL
[=noun, day-of-week] -->
Attach 1st to a new np node labelled time, dow as noun.
Set the time register of 1st to 'dow'.
Set the dow register of 1st to 1st.
Set the markers register of c to !'(time).
Drop c.
Restore the buffer.!

!AS RULE MONDAY-REGISTERS PRIORITY: 5 IN NPOOL
[=noun, day-of-week; it isn't true that there is a time of =] -->
/Sets some registers for semantics/
Set the time register of 1st to 'dow'.
Set the dow register of 1st to 1st.
Restore the buffer.!

!NR RULE TIME-FINISH IN CPOOL
[=np, time] -->
Activate time-complete, cpool.!

!RULE MONDAY-THE-FIRST IN TIME-COMPLETE
[== c; = is dow] [==/,]
     [=np,time; the noun of = is date] [= is not verb] -->
Attach 1st to c as comma.
Attach 2nd to c as date.
If 3rd is =/, then attach 3rd to c as comma.
Remove the feature dow from c.
Label c date.
Set the time register of the noun of c to 'date'.
Set the month register of the noun of c to
          the month register of the noun of 2nd.
Set the year register of the noun of c to
          the year register of the noun of 2nd.
Set the day register of the noun of c to
          the day register of the noun of 2nd.!
```

```
  !RULE JUNE-FIRST-1976 IN TIME-COMPLETE
[=* c; the noun of = is date] [= =/,] [=np, the noun of = is year]
      [= is not verb] -->
Attach 1st to c as comma.
Attach 2nd to c as year.
If 3rd is =/, then attach 3rd to c as comma.
Set the year register of the noun of c to
           the year register of the noun of 2nd.!


!RULE TIME-DONE PRIORITY: 15. IN TIME-COMPLETE
t -->
Finalize the cf of c.
Drop c.
Restore the buffer.!

!RULE NUMBER-TO-TIME PRIORITY: 12. IN QUANT-NP-COMPLETE
[== c; :cheap-hacks; = is quant-np; c is not time] -->
/An ellision hack. Clearly special purpose for the PA domain.
A better approximation would be only to use this hack if the
incomplete np followed a prep that marked time, e.g. "on the first",
but even that, obviously, could be wrong.  Recovering ellisions, as
opposed to detecting them, really shouldn't be in the grammar.%
If there is a qp of c and it is ordqp /e.g. "the first"%
           then run month-complete next else
If it is numqp and there is not a det of the np above it
           then (if lessp (the quant register of the quant of it, 13.) %e.g. "ten"%
                     then run hour-complete next else
                if lessp (1900., the quant register of the quant of it, 20000.) %e.g. "1976"%
                     then run number-to-year next
                else run np-done next)
else run np-done next.!

!RULE TIME-NP-TO-PP PRIORITY: 5 IN CPOOL
/This rule needs to be controlled, but right zeroeth approx%
[=np, time] -->
Insert the word 'during' into the buffer before 1st.!
```

```
;;/Pronouns, proper names, etc.%
(Comment Pronouns, proper names, etc.)

!RULE PRONOUN IN npool
[=pronoun] -->
Attach 1st to c as pronoun.
Label c pron-np, not-modifiable.
Transfer ns, npl, n1p, n2p, n3p, wh from 1st to c.
If 1st is relpron then label c relpron-np.
Drop c.
Restore the buffer.!

!AS RULE POSS-PRONOUN PRIORITY: 5 IN CPOOL
[=poss-pronoun] -->
Create a new det node labelled def, poss-det, ngstart, ns, npl, n3p.
Attach a new np node labelled pron-np, poss-np to c as np.
Attach 1st to c as pronoun.
Transfer ns, npl, n1p, n2p, n3p            from 1st to c.
Drop c. %the np node%
Drop c. %the det node%
Restore the buffer.!


!AS RULE PROPNAME PRIORITY: 5 IN CPOOL
[=name] -->
Create a np node labelled name, ns ,n3p, not-modifiable.
Activate build-name.!

!AS RULE TITLE PRIORITY: 5 IN CPOOL
[=title] -->
Create a np node labelled name, ns, n3p.
Attach 1st to c as title.
Set the title register of c to 1st. %set register for PA semantics%
Activate build-name.!


!RULE NAME IN BUILD-NAME
[=name] -->
Attach 1st to c as noun.
Set the names register of c to
          !(cons 1st !the names register of c1). %for pa semantics%!

!RULE END-OF-NAME PRIORITY: 15. IN BUILD-NAME
[t] -->
If !(> (length !the names register of c1) 1)
              or there is a title of c
          then set the last-name register of c to
                    !(car !the names register of c1);
               set the names register of c to
                    !(cdr !the names register of c1). %for pa semantics%
Set the names register of c to
          !(nreverse !the names register of c1).
Drop c.
Restore the buffer.!

!AS RULE PROPNOUN PRIORITY: 5 IN CPOOL
[=propnoun; = is not name] -->
Attach 1st to a new np node labelled propn-np, ns, n3p, not-modifiable as noun.
Drop c.
Restore the buffer.!
```

```
(Comment Pp attachment rules)
;;/PP rules/
!RULE PP IN CPOOL
[=prep; = is not pred-verb] [=np]
[== c; there is not a wh-comp or the wh-comp is utilized]
%for WH-placement PP rules, see WH-placement rules%
 -->
Attach 1st to a new pp node as prep.
Attach 2nd to c as np.
Drop c.!

!RULE NBAR-PP-UNDER-S IN NBAR-COMPLETE
[=pp] [== c: the node above the np above = is s] -->
If 1st fits a pp slot of the cf of c
          then attach 1st to c as pp
          else run nbar-done next.!

!RULE OF-PP IN NBAR-COMPLETE
[=pp; the prep of = is =of] -->
Attach 1st to c as pp.!

!RULE NBAR-PP-UNDER-VP IN NBAR-COMPLETE
[=pp] [== c; the node above the np above = is vp] -->
If 1st fits a pp slot of the cf of the node above the np above c
          then run nbar-done next
          else run nbar-pp-under-s next.!

!ATTACHMENT CRULE NBAR-PP NBAR OVER PP
If the prep of the lower node is =of
          then the np of the lower node fills a obj slot of the upper node
          else the lower node fills a pp slot of the upper node.!


!RULE NP-PP-UNDER-VP IN NP-COMPLETE
[=pp] [== c; the node above = is vp] -->
if it isn't true that 1st fits a pp slot of the cf of c
          or 1st fits a pp slot of the cf of the vp above c
then run np-done next
else attach 1st to c as pp.!

!RULE NP-PP-UNDER-PP IN NP-COMPLETE
[=pp] [== c ; the node above = is pp] -->
/For now, NPs only get a PP if the above VP or S does't want it.
which is wrong, but a zeroeth order approximation.
Will get wrong "Can we meet on the friday after the party"
Unless there is a notion of "mustmod"%

/For now,/ if the node above the pp above c is vp
          then run np-pp-under-vp next else
if the node above the pp above c is s
          and 1st fits a pp slot of the cf of the s above the pp above c
then run np-done next
else attach 1st to c as pp.!


!RULE NP-PP-DEFAULT PRIORITY: 15. IN NP-COMPLETE
[=pp] -->
%This module is intended to run only when the NP is clause initial.%
Attach 1st to c as pp.!

!ATTACHMENT CRULE NP-PP NP OVER PP
Associate a new mod case frame with the lower node.
The lower node fills a mod slot of the upper node.
The prep of the lower node fills the pred slot of lower.
The np of lower fills an obj slot of lower.
Create a new = node labelled np.
```

Set the binding of c to the upper node.
C fills the subj slot of lower.
Finalize the cf of lower.
Drop c.1

```
!RULE PP-UNDER-VP-1 IN SS-VP
[=pp] -->
If 1st fits a pp slot of the cf of c
          then attach 1st to c as pp
else run vp-done next.!

!RULE PP-UNDER-S-1 IN SS-FINAL
[=pp] -->
If 1st fits a pp slot of the cf of c
        then attach 1st to c as pp
else /what else can we do?/ attach 1st to c as pp.!

!RULE PP-UNDER-S-2 IN EMBEDDED-S-FINAL
[=pp] -->
If 1st fits a pp slot of the cf of c
        then attach 1st to c as pp
else run embedded-s-done next.!

!ATTACHMENT CRULE S-PREDP S OVER PREDP
If the lower node is pp then
          Associate a new case frame with the lower node;
          The prep of the lower node fills the pred slot of lower;
          The np of lower fills an obj slot of lower.
Associate the case frame of lower with the s above lower.
The np of the s above lower fills the subj slot of lower.!

!ATTACHMENT CRULE S-PP  S OVER PP
Associate a new mod case frame with the lower node.
The lower node fills a mod slot of the upper node.
The prep of the lower node fills the pred slot of lower.
The np of lower fills an obj slot of lower.
Create a new = node labelled s.
Set the binding of c to the upper node.
C fills the subj slot of lower.
Finalize the cf of lower.
Drop c.!

!CREATION CRULE =-DUMMY =
/All = nodes should disappear when dropped, so we make them look attached.X
Set the father of c to t.!

!RULE PP-UNDER-VP-2 IN EMBEDDED-S-VP
[=pp] -->
If 1st fits a pp slot of the cf of c
          then attach 1st to c as pp
else run embedded-vp-done next.!

!ATTACHMENT CRULE VP-PP  VP OVER PP
The lower node fills a pp slot of the upper node.!
```

APPENDIX E

THE CASE FRAME INTERPRETER

This appendix describes a case frame interpreter which monitors the construction of the Annotated Surface Structures described in this paper and constructs a Predicate/Argument representation of the utterance from the parser's output.

The discussion of this mechanism demonstrates three things: First it should demonstrate to readers who are unfamiliar with the Annotated Surface Structures discussed in this document that, given these structures, it is easy to compute a "deeper" representation with which the reader is perhaps more familiar. Second, it also demonstrates the simple semantic mechanism that was used to test preliminary versions of the theory of semantic/syntactic interaction for Wh-ungapping described in Chapter 10. Finally, the discussion will briefly touch on how this representation can be converted to a deeper "knowledge" representation for use in the Personal Assistant Project of the MIT AI Lab [Goldstein & Roberts 77].

I should stress at the outset that no linguistic claims are made for this mechanism, although I believe the mechanism to be of interest from an engineering viewpoint. In particular, this mechanism does *not* attempt to be deterministic; the algorithm for case mapping that it uses was directly taken from the approach of Stockwell, Schacter, and Partee [Stockwell, Schacter & Partee 73]. The semantic mechanism is more elaborate is some ways than

the semantic marker systems commonly used in many natural language understanding systems, but it was implemented primarily to provide a convenient test bed for answering semantic inquiries of the sort investigated in Chapter 10.

This appendix will first discuss the case frame mechanism itself. It will then turn to a discussion of the mechanism which monitors the construction of Annotated Surface Structures, and actually constructs case frames. The rules which drive this mechanism are interspersed with the grammar rules of Appendix D. It will then discuss the semantic marker component and the semantic preference mechanism. And finally, it will present an annotated trace of the case frame interpreter running on a simple example.

The Case Frame Interpreter

(The following discussion is taken from [Levin 77] which discusses and critiques the case frame interpreter and the algorithm which it implements. I should note that this paper also discusses and critiques alternative formulations of case frame notions, and presents an original set of rules for case frame interpretation, none of which is included here. I also include the general introduction to case frames from this paper; I agree with the position taken there. This text is used here with Levin's permission.

I should also note that the current version of the case frame interpreter was programmed by Kurt VanLehn; the original version was

programmed by the author.)

Section 1: Introduction

A case frame makes predicate-argument relations in a sentence explicit, but how are these relations extracted from an English sentence? And given a case frame, what arrangements of its cases can be found in sentences?

Describing these processes is not a trivial task. The information in a case frame may be expressed in a variety of ways in an English sentence. The verb *present* allows the following choices:

(1) The judge presented the prize to the boy.
(2) The judge presented the boy with the prize.
(3) The judge presented the boy the prize.

The example above also shows that the position of noun phrases with respect to a verb is not usually sufficient to uniquely determine what case it fills. Active-passive sentence pairs are another example of this: the grammatical subject of an active sentence and of its corresponding passive form do not fill the same case. Even prepositions, which are supposed to signal cases, can mark more than one case. *With* can mark the neutral, comitative, instrument, and manner cases. Before a case frame can be filled, the proper case frame must be chosen since some words have several:

(4) The committee met with the visitor.
(5) The proposal met with disapproval.

In (4), *meet* has an agent as subject, while in (5) it takes a neutral. *With* marks the neutral and manner cases respectively, in (4) and (5). Choosing

the right case frame is a part of the word sense problem that must be solved at the case frame level. *[This is not done by the current case frame interpreter; the case frames used are in some sense the union of the relevant word senses - MM]*

...
...
...

Section 2: Case Frames: A Semantic or Syntactic Representation?

The case representation discussed here is only intended to capture predicate-argument relations and word-sense disambiguation, but no deeper semantic generalizations. Case frames are an intermediate level in the mapping from an English sentence to its deep semantic representation. Although the case frame builder in a natural language system may interact with the semantic component to resolve word sense questions, case frames allow the semantic component to remain unaware of how predicate-argument relations are expressed in the sentence.

The purpose of a case frame representation as an intermediate step in the mapping from a sentence to its semantic representation is similar to that of functional decomposition. The case frame component can use a limited number of cases, enough to capture the different behaviors present in English sentences. Each case should embody a particular type of behavior in the case frame-grammatical relation mapping. Cases are not expected to reflect semantic roles; the slot names of the deep semantic frame may be chosen according to the frame's function. The process of lexical decomposition can be done, if desired, in the mapping from case frames to deep frames.

This theory of the place of case frames in a natural language system has been embedded in the natural language understander for the Personal Assistant project [Bullwinkle 1977, Goldstein and Roberts 1977]. The example below compares the case frame and deep frames for the verb *schedule* in sentence (1).

(1) I want to schedule a meeting at 3 p.m. Tuesday.

The case frame for *schedule* is on the left. The resulting deep frames are on the right.

```
schedule1                          schedule36
        agt     I                          actor       I
        neut    meeting                    activity    meeting37
        time    3 p.m. Tuesday
                                   meeting37
                                           when    3 p.m. Tuesday
```

The case frame for *schedule* fills slots of the deep frames of both *schedule* and *meeting*. In the PA domain, the mapping is done by means of simple functions such as the one below:

```
(SET-MAP (SCHEDULE1 SCHEDULE)
         (=> AGT (FILL ACTOR))
         (=> NEUT (FILL ACTIVITY))
         (=> TIME (INSERT-INTO NEUT WHEN))
         (=> PLACE (INSERT-INTO NEUT WHERE)))
```

This function maps the SCHEDULE1 word sense of *schedule* onto an instance of the deep SCHEDULE frame. The function FILL fills the actor slot of the deep frame with the agent of the case frame. The INSERT-INTO function puts the time and place cases into the when and where slots respectively of a frame created for the neutral case.

Section 3: Filling a Case Frame

Mitchell Marcus has implemented a case frame builder to convert the annotated surface structure produced by his parser [Marcus 1976] to a case frame representation. Marcus' case frames consist of four components:

1. Predicate: the root of the word whose case frame it is.
2. Specializers: added information about the predicate such as the auxiliaries preceding a verb or the determiner preceding a noun.
3. Cases: the filled cases present in this use of the predicate.
4. Modifiers: phrases which are case frames themselves used to modify an entire case frame rather than to specify a case. Modifiers are optional sentence level comments such as time or location.

The parser communicates with the case frame builder via messages informing it to fill in any of the four components, check that the obligatory slots of a case frame are filled, and check if a node of the annotated surface structure fits in a case frame. The problem of determining whether a prepositional phrase is a case or a modifier, a decision which may require semantic interaction, will not be discussed here. The concern is the means employed to fill the case slots of a case frame.

When Marcus' case frame builder is asked to fill a case slot of a case frame, the parser specifies the grammatical role of the node which is to be inserted: subject, object, or prepositional phrase. The case frame builder must be able to generate all possible cases that can have the specified grammatical role from the predicate's case frame. The interdependence of cases and grammatical roles means that each candidate must be paired with the cases which remain to be filled if it is chosen.

This results in a fifth component of a case frame which is used during the case filling process: a list of hypotheses describing the different ways to fill the case frame. Each hypothesis has two parts: the cases filled so far and the cases which remain to be filled. Initially, there is only one hypothesis consisting of no filled cases and the case frame from the predicate's lexical entry. Each time the case frame builder is asked to fill a slot with a certain grammatical role, the remaining cases in each hypothesis are examined for cases that can fill the role. Each such case results in a new hypothesis in which the chosen case is added to the hypothesis' filled case list. If no cases remain in the hypothesis or none of the cases remaining can fill that grammatical role, the hypothesis is discarded. This can also happen if an obligatory slot is left unfilled after the subject, objects, and prepositional phrases associated with the predicate have been found. Choices between certain hypotheses will have to be made according to semantic criteria, for example, the decision whether *the rock* is agent or instrument in "The rock broke the window." This ability is not part of the case frame builder, but the decision will be made by asking questions of the semantics component. *[See the next section - MM]*

The case frame builder must be able to generate all possible candidates for a grammatical role from a case frame; this information can be extracted from the results of the process which maps an underlying case frame to the alternative sequences of cases appearing in sentences.

...
...
...

Section 5: B. Stockwell, Schacter, and Partee's Approach

SS&P [1973] use a transformational framework to formalize their solution to the mapping problem. Their mechanism is based on a set of rules for finding the grammatical relations of subject and object and prepositional phrases from an ordered list of cases. ... In Marcus' case frame builder, the three most fundamental of SS&P's rules are incorporated into the functions that generate the cases that a grammatical role may fill.

The case frame in a verb's lexical entry consists of a subset of the ordered list of cases below:

(neutral) (dative) (locative) (instrument) (agent)

Each case present in the case frame is marked optional or obligatory (parentheses around the name of a case will indicate that it is optional, none indicate that it is obligatory). To turn the case frame into the possible sequences appearing in English sentences, the following rules, which were expressed in transformational terms by SS&P, are used:

(R1) Finding the Subject: the rightmost case must become the subject if it is obligatory. If it is optional, it may be discarded and the rule applied to the remaining cases.

(R2) Finding the Objects: the objects are found by reading from left to right until the number of objects is used up. The objects occur with no preposition.

(R3) Prepositional Phrases: the remaining cases occur marked by prepositions. Each case has a default marking preposition associated with it. If a verb requires some other preposition, it must be specified in the verb's lexical entry.

As an example of the use of the rules, consider the verb *break*:

(4) The boy (A) broke the window (N) with a rock (I).
(5) The rock (I) broke the window (N).
(6) The window (N) broke.

SS&P's case frame for *break* is: N (I) (A) . The neutral is obligatory, but the other cases are optional. *[This is exactly the notation used for cases by the case frame interpreter - MM]* None of *break*'s cases are marked by unusual prepositions. Applying (R1), the agent, the rightmost case, can become the subject. Then by (R2), the neutral, the leftmost case, will be the object, and by (R3) the instrument will be marked by *with*. This gives the structure in (7) which is that of (4):

(7) *A* break *N* with *I*.

Alternatively, since the agent is optional, it could have been discarded by (R1) leaving the instrument case as the rightmost case, and therefore, a candidate for subject. Once again by (R2) the neutral will be the object resulting in the pattern underlying (5):

(8) *I* break *N*.

(R1) could have been applied in a third way: both the agent and instrument, which are optional, could have been omitted leaving the neutral as the subject and no other cases as in (6). Sentence (9) cannot occur since by (R1) the instrument must be deleted from the case frame if the neutral occurs as subject.

(9) * The window (N) broke with a rock (I).

The first obligatory case must be chosen as the subject; this prevents *break* from occuring with no subject. It also allows verbs to have neutrals which do not occur in subject position by having an obligatory case to the right of the neutral in the case frame.

The rules described so far are independent of the verb and the cases present in the case frame; as a result they are inadequate. There is no way

of allowing the neutral to follow a dative or locative object as in *give* or *smear*, and no way for a neutral subject to precede a dative or locative as with *swarm*, *familiar*, and *drop*. Verbs like *hang* allow a neutral subject and a locative marked by a prepositional phrase, but not a locative subject followed by a neutral marked by a prepositional phrase. The rules will generate the latter sequence, but not the former. These problems can be handled in several ways:

1. allow verbs to have more than one case frame
2. have two underlying case orders
3. formulate rules that allow the case frame to be reordered
   a. these rules can depend on grammatical relations
   b. these rules can depend on the cases

The first two possibilities preserve the independence of the rules from the verb and case configuration, but are unsatisfactory for other reasons.... The method adopted by SS&P to overcome the inadequacies is that of (3a): rules (transformations in their framework) that allow certain cases to become subjects or objects, overriding (R1)-(R3).

SS&P's subjectivalization and objectivalization transformations can be considered general functions that move a case into subject or object position. The lexical entry of a verb must indicate whether either of the transformations apply to it, as well as specifying the case which is to be moved and the preposition which is to mark the case that would have occupied that position. *Swarm* allows its locative to subjectivalize while *smear*'s locative objectivalizes. In both verbs, the neutral will be marked by *with*. In Dative Shift verbs such as *give*, the dative objectivalizes. Subjectivalization is not used for ergative verbs like *drop*, instead SS&P modify (R1): if the next choice for subject is a marked locative, then the

first choice for object becomes the choice for subject. This treatment is inadequate because ergative verbs show the same behavior with datives, for example the verb *ring*:

(10) He (A) rang the bell (N) for class (D).
(11) The bell (N) rang for class (D).

The underlying case order and rules (R1)-(R3) give special properties to the neutral and agent since they occur at either end of the list. The subjectivalization and objectivalization transformations which SS&P propose allow the behavior of any verb to be duplicated. The mechanism is so general that any cases could be put in subject or object position even if they are never found there. The use of subjectivalization and objectivalization on verbs with neutrals that shift could be combined into one process if a transformation formulated in terms of cases were used. SS&P are concerned with mapping cases to the grammatical relations of subject and object, so their transformations are formulated in these terms.

[end of quotation from [Levin 77]]

.....

The Annotated Surface Structure Monitors

The case frame builder runs in parallel with the parser by means of a set of programs that monitor some of the types of actions the parser can take; a monitor program can be written so that it will be executed whenever a node of a given type is created or whenever a node of type A is attached to a node of type B. (There is also need for monitors which fire when nodes are completed, but this is currently not implemented.) These monitors are written in an extension of PIDGIN that amounts to a command

language for an independent case frame interpreter.

Several such monitors are exhibitted below in Figure E.1. Rather than explain the monitor mechanism in general, I will explain each of these monitors in particular; the examples chosen are sufficiently comprehensive, and the mechanism is simple enough, that this should serve as an adequate explanation of the monitor mechanism.

```
{ATTACHMENT CRULE VP-VERB  VP OVER VERB
Associate a new case frame with the upper node.
Associate the case frame of the upper node with the s above upper.
The aux of the s above upper fills the spec slot of  upper.
The lower node fills the pred slot of  upper.
If the lower node is none of no-subj, passive
     and there is a binding of the np of the s above upper
     then it fills the subj slot of upper.}

{CREATION CRULE NP-START NP
if c is none of propn-np, pron-np, name, complex-noun-np, trace, *,
               comp
               then associate a new case frame with c.}

{ATTACHMENT CRULE NP-ADJ NP OVER ADJ
%Note that this does not handle non-attributive adjectives like "main"%
Associate a new mod case frame with the lower node.
The lower node fills a mod slot of the upper node.
The lower node fills the pred slot of lower.
Create a new * node labelled np.
Set the binding of c to the upper node.
C fills the subj slot of lower.
Finalize the cf of lower.
Drop c.}
```

Figure E.1 - Some example surface structure monitors.

The first of these monitors, the attachment monitor VP-VERB, is triggered whenever a VP is attached over a VERB, as its "pattern" indicates. This rule begins the case frame for a clause, fills the *specifier* of the case frame with the auxiliary of the clause, the *predicate* of the case frame with

the verb of the clause, and then tells the case frame interpreter that the surface subject of the clause serves as "deep" subject as long as the verb has neither of the features *passive* or *no-subj* (i.e. it is not a verb like "seem") and the subject itself has been determined.

Note that case frames (cfs) themselves are associated with specific parse nodes (although the mapping is many-to-one, i.e. many parse nodes can be associated with the same cf), and a cf is referred to by a node that it is associated with. The extension of PIDGIN which serves as the case frame interpreter's command language refers to the slots a constituent fills in terms of the grammatical function of the constituent. It is the case frame interpreter's job to map this function into possible case roles. The PIDGIN to inform the case frame interpreter of a constituent's role is thus

<node1> fills the <role> slot of the cf of <node2>.

The second monitor in Figure E.1 is a creation monitor, in this case a monitor which is executed immediately after an NP node is created. This monitor, called NP-START, will associate a cf with an NP node if the NP is not initially labelled with a feature indicating that it will not have a lexical noun as a head. (I should note that it is purely gratuitous that the name of this rule is the same as the attention shifting grammar rule discussed in Chapter 8.)

The third monitor, the attachment monitor NP-ADJ, indicates how *modifiers* are handled by the case frame interpreter. Modifiers are the case frame equivalent of constituents like adjectives and relative clauses with

respect to NPs, and sentential adverbials and prepositional phrases directly dominated by an S (as opposed to those directly dominated by a VP.) A cf is created to represent the modifier, in this case an adjective, and this cf is associated with the adjective. This case frame then fills a special *modifier* slot of the NP itself. Finally, a special type of non-grammatical node called a "* node" is created, bound to the NP node and used to fill the appropriate slot in the case frame of the adjective. (For reasons which I will not pursue here, the cf interpreter considers this to be a subject slot. The usage is not overly important.)

Note that the mechanisms of the parser are used to create and manipulate this "*" node; this is done mainly for the sake of convenience. Because a node was created, however, the line "Drop C." must be executed to pop this node from the active node stack. The "Drop" mechanism has been programmed to handle "*" nodes specially, and will not insert them into the buffer. Thus, the parser itself will never be aware that its mechanisms were usurped for the purposes of the case frame interpreter.

The Semantic Marker Component

To decide whether or not a given NP can fill a given case slot, there is need for some form of semantic test. The case frame interpreter, like many other natural language understanding systems, uses a semantic filter based upon a set of Fodor-Katz semantic markers [Katz & Fodor 64]. Associated with each noun in the lexicon is a set of semantic markers; for example, associated with "boy" are the markers HANIM (higher animate) ANIM (animate) and PHYSOB (physical object). Associated with each case is a

similar set of markers; associated with the AGENT case, for instance, are the markers HANIM and ANIM. If the intersection of the marker set of the head noun of a given NP and the marker set of a given case is nonempty, then that NP can fill that case. Because any given case may be semantically constrained with respect to a particular verb, a set of markers can be associated with any case with respect to that verb. The case frame interpreter, when deciding whether an NP can fill a given case role for a particular verb, first checks to see whether that case has a marker set with respect to that verb; if there is a marker set, the case frame interpreter will use it, if not, it will use the general set of markers for that case. All of this is typical of many natural language understanding systems, and is quite unremarkable.

While this approach is sufficient to decide the yes/no question of whether a given NP can fill a given case role for a particular verb, it is not sufficient to answer the preference questions of Chapter 10. In that chapter, I argued that a deterministic parser must ask questions of comparative semantic goodness; i.e. it must be able to ask not only whether a given constituent is acceptable in a particular syntactic role, but also how good it is in that role. (Of course, the case frame interpreter translates questions about syntactic roles into questions about case roles, using the SS&P algorithm discussed above.) The simple semantic marker representation discussed above will not suffice for such tests.

To allow comparative tests to be performed while still using a fairly simple semantic marker representation, the following scheme was used:

Instead of simply providing an undifferentiated set of markers for each case, as indicated above, each marker set was divided into two "bins", the first a set of markers that signify a good semantic "fit", and the second a set of markers signify an "ok" semantic fit. Figure E.2 below indicates the marker sets for a representative sample of the cases used in the system; each set is of the form

(<"good" markers> # <"ok" markers>).

It should be noted that I make no claims for the correctness of these divisions, nor for the particular features chosen.

AGENT:	(HANIM # ANIM)
DATIVE:	(HANIM # ANIM)
EXCHANGE:	(MONEY # PHYSOB ANIM INANIM HANIM)
BENEFACTIVE:	(HANIM # ANIM INANIM PHYSOB)
LOCATIVE:	(PLACE # INANIM)

Figure E.2 - Marker "bins" for some example cases.

Given this division, tests of comparative goodness can be performed as follows (note that this is not quite the method outlined in Chapter 10):

First of all, to determine how well a given NP fills a given case slot, its marker set is intersected with both the "good" and the "ok" marker sets. If the intersection with the "good" set is nonempty, the fit is good, otherwise if the intersection with the "ok" set is nonempty, the fit is ok, otherwise the fit is bad. Now to perform a comparative test of the semantic preference for NP_1 in Semantic-$Role_1$ over NP_2 in Semantic-$Role_2$, the system first computes the relative goodness of each NP in its respective semantic role, and compares the results according to the following rule:

-If the acceptability of both NP_1 and NP_2 is identical, there is *no preference*.

-If NP_2 is bad, and NP_1 is not, NP_1 is *strongly preferred* to NP_2.

-If NP_1 is good, and NP_2 is ok, then NP_1 is *mildly preferred* to NP_2.

While this test is not truly comparative, it has sufficed as a test bed for semantic rules like those discussed in Chapter 10, and will suffice until a more adequate scheme for comparative tests can be devised.

An Example

This section includes an annotated trace of the functioning of the case frame interpreter. The trace shows (1) what monitors are run, (2) the messages that the case frame interpreter sends to deeper semantics, passing on its analysis, and (3) the current state of the cf interpreter's internal hypotheses. The annotation is enclosed in square brackets. Messages concerning the internal structure of the NPs have been deleted. The possible mappings from subject to the case frame considered by the case frame interpreter for the verb "break" are exactly those enumerated by Levin in her discussion of the SS&P algorithm.

```
> the rock broke the window .
PARSING
 Running create-rule S-CREATE
 Running attach-rule VP-VERB

 To semantics: NEWFRAME CASEFR26 (VP10 . 1) NORMAL
        [a newcaseframe, CASEFR6, is created and associated with the node
                VP10.  Note that these are messages sent from the cf
                interpreter to deeper semantics, not from the monitors to
                the cf interpreter.]

 To semantics: HEAD (WORD55 . 1) CASEFR26
        [its head is WORD55, the word "broke"]

 New frames for (VP10 . 1)
        [after the cf is told NP16, "the rock" is its subject]
        PRED        BREAK

        HYPO-SLOTS  ((((NEUT OBLIG)) ((INS (NP16 . 3) SUBJ)))
                            [either NP16 is an INSTRUMENT, with the NEUTRAL unfill

                           (NIL ((NEUT (NP16 . 3) SUBJ))))
                            [or it is the NEUTRAL, with no more cases to come,
                                    as in "The rock broke."]

        CASES       NIL [No cases are known for sure]
-----------
 Running attach-rule VP-NP
        [Now the cf is told NP17, "the window" is its object.  This rules
                out the possibility that NP16 can by NEUTRAL, because this
                would leave no more case to fill]

 To semantics: CASE (INS (NP16 . 3) SUBJ) CASEFR26
        [It's sure that NP16 is INSTRUMENT]

 To semantics: CASE (NEUT (NP17 . 1) OBJ) CASEFR26
        [And that NP17 is NEUTRAL]

 New frames for (VP10 . 1)
        PRED        BREAK
        HYPO-SLOTS  (((NIL) ((NEUT (NP17 . 1) OBJ) (INS (NP16 . 3) SUBJ))))
                [Now there's only one hypothesis, with no cases unfilled]

        CASES       ((INS (NP16 . 3) SUBJ) (NEUT (NP17 . 1) OBJ))
                [These cases it's sure of]
-----------
 To semantics: FINALIZE-THE-FRAME CASEFR26
        [Later code signals the end of the sentence, and semantics is told
                of this]
 DONE
```